



PARALLEL COMPUTER ARCHITECTURE

A Hardware/Software Approach

David E. Culler
Jaswinder Pal Singh
with Anoop Gupta



PRAISE FOR PARALLEL COMPUTER ARCHITECTURE A Hardware/Software Approach

Parallel computing is entering the mainstream, thanks to complementary advances in VLSI, architecture, system software, and programming support. Culler, Singh, and Gupta—outstanding researchers and outstanding educators—have helped to drive this dramatic convergence and now have brought it all together in this authoritative and readable text. Once again, Morgan Kaufmann has delivered a landmark book in computer systems.—**Edward D. Lazowska, Professor and Chair, Department of Computer Science & Engineering, University of Washington**

This book covers the latest machine architectures and techniques at a time when there is explosive growth in parallel machine use, but many of the older machine organizations are running out of gas. The book provides useful techniques for understanding the performance of parallel machines and gives examples of how they apply to the various architectural options. Students, researchers, and industry practitioners will find this book extremely valuable. I know I have.—**Dan Lenoski, SGI**

Parallel Computer Architecture offers a wealth of information about the features of actual machines, which simply cannot be found in any other book. The selection of topics strikes a good balance between research results and real life implementations.—**Michel Dubois, Professor, Department of Electrical Engineering-Systems, University of Southern California**

This book provides a fresh look at an important topic in computer architecture, including an in-depth examination of several of the thorny problems in parallel computing.—**David Patterson, Pardee Professor of Computer Science, University of California, Berkeley**

This book thoroughly presents the state of the art—delving into issues only glossed over in many research papers.—**Mark Hill, Professor & Romnes Fellow, Computer Sciences Department, University of Wisconsin-Madison**

Finally, the Hennessy & Patterson of parallel architectures has arrived. Written by foremost experts in the field, it is full of case studies, quantitative data, and architectural insights. Integrated hardware/software approach is the only way to understand parallel computing, and this book is an existence proof.—**Arvind, Charles W. & Jennifer C. Johnson Professor of Computer Science & Engineering, Massachusetts Institute of Technology**

RELATED TITLES FROM MORGAN KAUFMANN PUBLISHERS

Computer Architecture: A Quantitative Approach, Second Edition, by John L. Hennessy and David A. Patterson

Computer Organization and Design: The Hardware/Software Interface, Second Edition, by David A. Patterson and John L. Hennessy

Scalable Shared-Memory Multiprocessing, by Daniel E. Lenoski and Wolf-Dietrich Weber

The Grid: Blueprint for a New Computing Infrastructure, edited by Ian Foster and Carl Kesselman

Parallel Programming with MPI, by Peter S. Pacheco

Distributed Algorithms, by Nancy A. Lynch

Introduction to Parallel Algorithms and Architectures: Arrays, Trees, and Hypercubes, by F. Thomson Leighton

Parallel Computing Works! by Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina

Parallel Processing from Applications to Systems, by Dan I. Moldovan

Synthesis of Parallel Algorithms, edited by John H. Reif

VLSI and Parallel Computation, edited by Robert Suaya and Graham Birtwistle

FORTHCOMING

Readings in Computer Architecture, edited by Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi

Industrial Strength Parallel Computing, edited by Alice E. Koniges

Parallel Computer Architecture

Parallel Computer

A Hardware/Software Approach

Frank Cazier Award for Outstanding and Innovative Research (BECAS) and a Sloan Research Fellowships.

Frank Cazier Award for Outstanding and Innovative Research (BECAS) and a Sloan Research Fellowships.

David E. Culler, Professor of Computer Science at University of California, Berkeley, has been selected as the recipient of the Frank Cazier Award for Outstanding and Innovative Research (BECAS) and a Sloan Research Fellowships.

The Frank Cazier Award for Outstanding and Innovative Research (BECAS) and a Sloan Research Fellowships.

About the Authors

David E. Culler, Professor of Computer Science at University of California, Berkeley. Dr. Culler works in the areas of computer architecture, communication, programming languages, operating systems, and performance analysis. He led the Berkeley Network of Workstations (NOW) project, which sparked the current commercial revolution in high-performance clusters. He is internationally known for his work on Active Messages for fast communication, the LogP parallel performance model, the Split-C parallel language, the TAM threaded abstract machine, and for his work on dataflow architectures. He received the Presidential Faculty Fellowship Award and the Presidential Young Investigator Award from the National Science Foundation. He received the PhD from MIT in 1989. Currently he is vice-chair of computing and networking for the Department of Electrical Engineering and Computer Sciences at UC Berkeley and leads the Millennium Project, investigating campus-wide clusters.

Jaswinder Pal Singh, Assistant Professor in the Computer Science Department at Princeton University. Dr. Singh works at the boundary of parallel applications and multiprocessor systems, including architecture, software, and performance evaluation. He has led the development and distribution of the SPLASH and SPLASH-2 suites of parallel programs, which are very widely used in parallel systems research. While at Stanford, where he obtained his MS and PhD degrees, he participated in the DASH and FLASH multiproces-

sor projects, leading the applications efforts there. The technology developed in the DASH project is becoming widely available in commercial products. At Princeton, he heads PRISM, an application-driven research group that investigates supporting programming models on a variety of communication architectures and applies parallel computing to a variety of application domains. He is a recipient of the Presidential Early Career Award for Scientists and Engineers (PECASE) and a Sloan Research Fellowship.

Anoop Gupta, Associate Professor of Computer Science and Electrical Engineering at Stanford University and Senior Researcher at Microsoft. Dr. Gupta has worked in the areas of computer architecture, operating systems, programming languages, performance debugging tools, and parallel applications. With John Hennessy, he co-led the design and construction of the Stanford DASH machine, one of the first scalable distributed shared memory multiprocessors, and has worked on the follow-up FLASH project. The technology developed in the DASH project is now becoming widely available in commercial products. Professor Gupta has published close to 100 papers in major conferences and journals, including several award papers. Professor Gupta received the NSF Presidential Young Investigator Award and held the Robert Noyce faculty scholar chair at Stanford. He obtained the PhD from Carnegie Mellon University in 1986.

To Sara, Silvia, and our families

Parallel Computer Architecture

A Hardware/Software approach

Digital ISBN: 0-12-2286-03-3-1

David E. Culler Jaswinder Pal Singh

with Anoop Gupta

Front Cover Rating 2000

Back Cover 2000

Rating 2000

Rating 2000 (Micro)

Rating 2000

Rating 2000

Rating 2000

Digital ISBN: 0-12-2286-03-3-2



ELSEVIER

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW DELHI • NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO

Morgan Kaufmann Publishers is an imprint of Elsevier



MORGAN
KAUFMANN

Parallel Computer Architecture

Culler & Singh

Morgan Kaufmann Publishers

An imprint of Elsevier

340 Pine Street, Sixth Floor
San Francisco, CA 94104-3205, USA

© 1999 by Morgan Kaufmann Publishers, Inc.

Original ISBN : 978-1-5586-0343-1

All rights reserved.

No part of this publication may be reproduced or transmitted in any form or by any means—electronic or mechanical, including photocopy, recording, or any information storage and retrieval system—with permission in writing from the publisher.

First Indian Reprint 2000

Reprinted 2002

Reprinted 2003

Reprinted 2004 (Twice)

Reprinted 2008, 2009, 2010

Reprinted 2011

Indian Reprint ISBN : 978-81-8147-189-5

This edition has been authorized by Elsevier for sale in the following countries: India, Pakistan, Nepal, Sri Lanka and Bangladesh. Sale and purchase of this book outside these countries is not authorized and is illegal.

Published by Elsevier, a division of Reed Elsevier India Private Limited.

Registered Office: 622, Indraprakash Building, 21 Barakhamba Road,
New Delhi-110 001.

Corporate Office: 14th Floor, Building No. 10B, DLF Cyber City, Phase-II,
Gurgaon-122 002, Haryana, India.

Printed and bound at Saurabh Printers Pvt. Ltd., A-15-16, Sector IV, Noida-201 301.

To Sara, Silvia, and our families

Foreword

John L. Hennessy

Formerly Exutive Dean of Engineering, Stanford University

In memory of Susanne Kreith Culler

I am delighted to be able to write the foreword for this exciting and timely new book on parallel computing. The insightful approach taken by the authors combined with a systematic and quantitative examination of different architectures distinguishes this book from all previous books on parallel architecture. The approach, which is developed in the first four chapters, has three major innovations: it builds on the recent convergence of parallel architectures, it uses applications as a driver for evaluating and analyzing architectures, and it is grounded in a solid methodology for performance evaluation.

The recent convergence among the shared memory and message-passing paradigms, which is described in Chapter 1, provides new opportunities for characterizing and analyzing architectures in a common framework. Relying on this convergence, the authors describe four fundamental design issues: communication abstraction, programming model, communication and replication, and performance. They create a framework for talking about a wide variety of architectures and organizations. Within this framework, different architectural approaches are compared and evaluated critically.

One cannot understand the design trade-offs or performance of multiprocessors without understanding the interaction of applications and architecture. Accordingly, Chapters 2 and 3 describe a set of parallel programs as well as how the applications are parallelized and organized for performance. These chapters illuminate both the parallel programming process and its challenges in addition to laying a foundation for quantitative evaluation of architectural approaches and implementations. These chapters are key to understanding the performance of multiprocessors, and Chapter 4 illustrates this by showing how to evaluate an architecture using a parallel workload. The authors also describe the complexities of evaluating parallel machines, including issues ranging from the scaling of machine sizes and workloads. Together these three chapters form the foundation on which the remaining chapters build.

Small-to-medium-sized shared-memory multiprocessors are the dominant form of parallel architecture seen today, and understanding the principles and design trade-offs of these machines is critical to anyone interested in parallel computing.

Foreword

John L. Hennessy

Frederick Emmons Terman Dean of Engineering, Stanford University

I am delighted to be able to write the foreword for this exciting and timely new book on parallel computing. The insightful approach taken by the authors combined with a systematic and quantitative examination of different architectures distinguishes this book from all previous books on parallel architecture. The approach, which is developed in the first four chapters, has three major innovations: it builds on the recent convergence of parallel architectures, it uses applications as a driver for evaluating and analyzing architectures, and it is grounded in a solid methodology for performance evaluation.

The recent convergence among the shared memory and message-passing paradigms, which is described in Chapter 1, provides new opportunities for characterizing and analyzing architectures in a common framework. Relying on this convergence, the authors describe four fundamental design issues (communication abstraction, programming model, communication and replication, and performance) that create a framework for talking about a wide variety of architectures and implementations. Within this framework, different architectural approaches are compared and examined critically.

One cannot understand the design trade-offs or performance of multiprocessors without understanding the interaction of applications and architecture. Accordingly, Chapters 2 and 3 describe a set of parallel programs as well as how the applications are parallelized and organized for performance. These chapters illuminate both the parallel programming process and its challenges in addition to laying a foundation for quantitative evaluation of architectural approaches and implementations. These chapters are key to understanding the performance of multiprocessors, and Chapter 4 illustrates this by showing how to evaluate an architecture using a parallel workload. The authors also describe the complexities of evaluating parallel machines, including issues arising from the scaling of machine sizes and workloads. Together these three chapters form the foundation on which the remaining chapters build.

Small-to-medium-sized shared memory multiprocessors are the dominant form of parallel architecture seen today, and understanding the principles and design trade-offs of these machines is critical to anyone interested in parallel computing.

Chapter 5 describes the key concepts underlying shared memory multiprocessing: cache coherency, memory consistency, and synchronization. The authors then describe the detailed design of snoop-based shared memory multiprocessors, including two detailed case studies, in Chapter 6.

Designing multiprocessors that scale to larger numbers of processing nodes remains one of the most challenging and controversial aspects of multiprocessor architecture. Chapter 7 devotes itself to such machines, spanning the design space from message passing to shared memory. Chapter 8 extends this discussion by examining the use of directory schemes, which allow cache coherency to scale to larger numbers of processing nodes. The basics of directory-based coherence are discussed, and two detailed case studies form the core of the chapter. These case studies are the first detailed and quantitative examinations of commercial implementations of directory-based cache coherence.

Some of the most important hardware and software technologies used in multiprocessors are largely independent of the details of the architectural approach. Hence, the authors explore these key technologies in a set of three chapters. Chapter 9 describes the software implications, hardware requirements, and performance trade-offs that arise in memory systems, including both consistency issues and the extended use of caching. Chapter 10 examines interconnection technology, a key constituent of any multiprocessor. Finally, Chapter 11 examines techniques for tolerating latency, in many ways the key “universal” design problem for parallel computers.

The book concludes with an insightful discussion of future hardware and software challenges. First, the authors discuss likely evolutionary scenarios in the hardware and software domain. Then they turn to the potential hurdles in a pair of sections entitled “Hitting a wall.” Finally, they examine potential breakthroughs! I found the final chapter both stimulating and thought provoking. The different backgrounds and complementary strengths of the authors help make this chapter both perspicacious and provocative.

In summary, this is an exciting and dynamic new exploration of the multiprocessor design space. The convergence in architectural approaches combined with the authors’ framework has made it possible to establish a common ground on which to examine the diversity of modern parallel architectures. A few years ago, it would have been impossible to write this book because the architectural approaches were too divergent. Similarly, without the attention to quantitative measures of performance and the interaction between applications and architectures, this book would be much less distinctive. Instead, the authors have taken advantage of the convergence and the focus on an applications-driven and performance-based analysis to produce a unique and insightful exploration of parallel architectures. This approach, combined with the unique strengths and experiences of the authors, yields a treatise that is far more perceptive than any other book in parallel architecture. I congratulate the authors and commend this book to all readers interested in both the practice and concepts of parallel processing and the future of these technologies.

Contents

Foreword ix

Preface xxi

1 Introduction 1

- 1.1 Why Parallel Architecture 4
 - 1.1.1 Application Trends 6
 - 1.1.2 Technology Trends 12
 - 1.1.3 Architectural Trends 14
 - 1.1.4 Supercomputers 21
 - 1.1.5 Summary 23
- 1.2 Convergence of Parallel Architectures 25
 - 1.2.1 Communication Architecture 25
 - 1.2.2 Shared Address Space 28
 - 1.2.3 Message Passing 37
 - 1.2.4 Convergence 42
 - 1.2.5 Data Parallel Processing 44
 - 1.2.6 Other Parallel Architectures 47
 - 1.2.7 A Generic Parallel Architecture 50
- 1.3 Fundamental Design Issues 52
 - 1.3.1 Communication Abstraction 53
 - 1.3.2 Programming Model Requirements 53
 - 1.3.3 Communication and Replication 58
 - 1.3.4 Performance 59
 - 1.3.5 Summary 63
- 1.4 Concluding Remarks 63
- 1.5 Historical References 66
- 1.6 Exercises 70

2 Parallel Programs 75

- 2.1 Parallel Application Case Studies 76
 - 2.1.1 Simulating Ocean Currents 77
 - 2.1.2 Simulating the Evolution of Galaxies 78
 - 2.1.3 Visualizing Complex Scenes Using Ray Tracing 79
 - 2.1.4 Mining Data for Associations 80
- 2.2 The Parallelization Process 81
 - 2.2.1 Steps in the Process 82
 - 2.2.2 Parallelizing Computation versus Data 90
 - 2.2.3 Goals of the Parallelization Process 91
- 2.3 Parallelization of an Example Program 92
 - 2.3.1 The Equation Solver Kernel 92
 - 2.3.2 Decomposition 93
 - 2.3.3 Assignment 98
 - 2.3.4 Orchestration under the Data Parallel Model 99
 - 2.3.5 Orchestration under the Shared Address Space Model 101
 - 2.3.6 Orchestration under the Message-Passing Model 108
- 2.4 Concluding Remarks 116
- 2.5 Exercises 117

3 Programming for Performance 121

- 3.1 Partitioning for Performance 123
 - 3.1.1 Load Balance and Synchronization Wait Time 123
 - 3.1.2 Reducing Inherent Communication 131
 - 3.1.3 Reducing the Extra Work 135
 - 3.1.4 Summary 136
- 3.2 Data Access and Communication in a Multimemory System 137
 - 3.2.1 A Multiprocessor as an Extended Memory Hierarchy 138
 - 3.2.2 Artifactual Communication in the Extended Memory Hierarchy 139
 - 3.2.3 Artifactual Communication and Replication: The Working Set Perspective 140
- 3.3 Orchestration for Performance 142
 - 3.3.1 Reducing Artifactual Communication 142
 - 3.3.2 Structuring Communication to Reduce Cost 150
- 3.4 Performance Factors from the Processor's Perspective 156
- 3.5 The Parallel Application Case Studies: An In-Depth Look 160
 - 3.5.1 Ocean 161
 - 3.5.2 Barnes-Hut 166
 - 3.5.3 Raytrace 174
 - 3.5.4 Data Mining 178

3.6	Implications for Programming Models	182
3.6.1	Naming	184
3.6.2	Replication	184
3.6.3	Overhead and Granularity of Communication	186
3.6.4	Block Data Transfer	187
3.6.5	Synchronization	188
3.6.6	Hardware Cost and Design Complexity	188
3.6.7	Performance Model	189
3.6.8	Summary	189
3.7	Concluding Remarks	190
3.8	Exercises	192

4 Workload-Driven Evaluation 199

4.1	Scaling Workloads and Machines	202
4.1.1	Basic Measures of Multiprocessor Performance	202
4.1.2	Why Worry about Scaling?	204
4.1.3	Key Issues in Scaling	206
4.1.4	Scaling Models and Speedup Measures	207
4.1.5	Impact of Scaling Models on the Equation Solver Kernel	211
4.1.6	Scaling Workload Parameters	213
4.2	Evaluating a Real Machine	215
4.2.1	Performance Isolation Using Microbenchmarks	215
4.2.2	Choosing Workloads	216
4.2.3	Evaluating a Fixed-Size Machine	221
4.2.4	Varying Machine Size	226
4.2.5	Choosing Performance Metrics	228
4.3	Evaluating an Architectural Idea or Trade-off	231
4.3.1	Multiprocessor Simulation	233
4.3.2	Scaling Down Problem and Machine Parameters for Simulation	234
4.3.3	Dealing with the Parameter Space: An Example Evaluation	238
4.3.4	Summary	243
4.4	Illustrating Workload Characterization	243
4.4.1	Workload Case Studies	244
4.4.2	Workload Characteristics	253
4.5	Concluding Remarks	262
4.6	Exercises	263

5 Shared Memory Multiprocessors 269

5.1	Cache Coherence	273
5.1.1	The Cache Coherence Problem	273
5.1.2	Cache Coherence through Bus Snooping	277

5.2	Memory Consistency	283
5.2.1	Sequential Consistency	286
5.2.2	Sufficient Conditions for Preserving Sequential Consistency	289
5.3	Design Space for Snooping Protocols	291
5.3.1	A Three-State (MSI) Write-Back Invalidation Protocol	293
5.3.2	A Four-State (MESI) Write-Back Invalidation Protocol	299
5.3.3	A Four-State (Dragon) Write-Back Update Protocol	301
5.4	Assessing Protocol Design Trade-offs	305
5.4.1	Methodology	306
5.4.2	Bandwidth Requirement under the MESI Protocol	307
5.4.3	Impact of Protocol Optimizations	311
5.4.4	Trade-Offs in Cache Block Size	313
5.4.5	Update-Based versus Invalidation-Based Protocols	329
5.5	Synchronization	334
5.5.1	Components of a Synchronization Event	335
5.5.2	Role of the User and System	336
5.5.3	Mutual Exclusion	337
5.5.4	Point-to-Point Event Synchronization	352
5.5.5	Global (Barrier) Event Synchronization	353
5.5.6	Synchronization Summary	358
5.6	Implications for Software	359
5.7	Concluding Remarks	366
5.8	Exercises	367

6 Snoop-Based Multiprocessor Design 377

6.1	Correctness Requirements	378
6.2	Base Design: Single-Level Caches with an Atomic Bus	380
6.2.1	Cache Controller and Tag Design	381
6.2.2	Reporting Snoop Results	382
6.2.3	Dealing with Write Backs	384
6.2.4	Base Organization	385
6.2.5	Nonatomic State Transitions	385
6.2.6	Serialization	388
6.2.7	Deadlock	390
6.2.8	Livelock and Starvation	390
6.2.9	Implementing Atomic Operations	391
6.3	Multilevel Cache Hierarchies	393
6.3.1	Maintaining Inclusion	394
6.3.2	Propagating Transactions for Coherence in the Hierarchy	396
6.4	Split-Transaction Bus	398
6.4.1	An Example Split-Transaction Design	400
6.4.2	Bus Design and Request-Response Matching	400

6.4.3	Snoop Results and Conflicting Requests	402
6.4.4	Flow Control	404
6.4.5	Path of a Cache Miss	404
6.4.6	Serialization and Sequential Consistency	406
6.4.7	Alternative Design Choices	409
6.4.8	Split-Transaction Bus with Multilevel Caches	410
6.4.9	Supporting Multiple Outstanding Misses from a Processor	413
6.5	Case Studies: SGI Challenge and Sun Enterprise 6000	415
6.5.1	SGI Powerpath-2 System Bus	417
6.5.2	SGI Processor and Memory Subsystems	420
6.5.3	SGI I/O Subsystem	422
6.5.4	SGI Challenge Memory System Performance	424
6.5.5	Sun Gigaplane System Bus	424
6.5.6	Sun Processor and Memory Subsystem	427
6.5.7	Sun I/O Subsystem	429
6.5.8	Sun Enterprise Memory System Performance	429
6.5.9	Application Performance	429
6.6	Extending Cache Coherence	433
6.6.1	Shared Cache Designs	434
6.6.2	Coherence for Virtually Indexed Caches	437
6.6.3	Translation Lookaside Buffer Coherence	439
6.6.4	Snoop-Based Cache Coherence on-Rings	441
6.6.5	Scaling Data and Snoop Bandwidth in Bus-Based Systems	445
6.7	Concluding Remarks	446
6.8	Exercises	446

7 Scalable Multiprocessors 453

7.1	Scalability	456
7.1.1	Bandwidth Scaling	457
7.1.2	Latency Scaling	460
7.1.3	Cost Scaling	461
7.1.4	Physical Scaling	462
7.1.5	Scaling in a Generic Parallel Architecture	467
7.2	Realizing Programming Models	468
7.2.1	Primitive Network Transactions	470
7.2.2	Shared Address Space	473
7.2.3	Message Passing	476
7.2.4	Active Messages	481
7.2.5	Common Challenges	482
7.2.6	Communication Architecture Design Space	485
7.3	Physical DMA	486
7.3.1	Node-to-Network Interface	486
7.3.2	Implementing Communication Abstractions	488

7.3.3	A Case Study: nCUBE/2	488
7.3.4	Typical LAN Interfaces	490
7.4	User-Level Access	491
7.4.1	Node-to-Network Interface	491
7.4.2	Case Study: Thinking Machines CM-5	493
7.4.3	User-Level Handlers	494
7.5	Dedicated Message Processing	496
7.5.1	Case Study: Intel Paragon	499
7.5.2	Case Study: Meiko CS-2	503
7.6	Shared Physical Address Space	506
7.6.1	Case Study: CRAY T3D	508
7.6.2	Case Study: CRAY T3E	512
7.6.3	Summary	513
7.7	Clusters and Networks of Workstations	513
7.7.1	Case Study: Myrinet SBUS Lanai	516
7.7.2	Case Study: PCI Memory Channel	518
7.8	Implications for Parallel Software	522
7.8.1	Network Transaction Performance	522
7.8.2	Shared Address Space Operations	527
7.8.3	Message-Passing Operations	528
7.8.4	Application-Level Performance	531
7.9	Synchronization	538
7.9.1	Algorithms for Locks	538
7.9.2	Algorithms for Barriers	542
7.10	Concluding Remarks	548
7.11	Exercises	548

8 Directory-Based Cache Coherence 553

8.1	Scalable Cache Coherence	558
8.2	Overview of Directory-Based Approaches	559
8.2.1	Operation of a Simple Directory Scheme	560
8.2.2	Scaling	564
8.2.3	Alternatives for Organizing Directories	565
8.3	Assessing Directory Protocols and Trade-Offs	571
8.3.1	Data Sharing Patterns for Directory Schemes	571
8.3.2	Local versus Remote Traffic	578
8.3.3	Cache Block Size Effects	579
8.4	Design Challenges for Directory Protocols	579
8.4.1	Performance	584
8.4.2	Correctness	589

- 8.5 Memory-Based Directory Protocols: The SGI Origin System 596
 - 8.5.1 Cache Coherence Protocol 597
 - 8.5.2 Dealing with Correctness Issues 604
 - 8.5.3 Details of Directory Structure 609
 - 8.5.4 Protocol Extensions 610
 - 8.5.5 Overview of the Origin2000 Hardware 612
 - 8.5.6 Hub Implementation 614
 - 8.5.7 Performance Characteristics 618
- 8.6 Cache-Based Directory Protocols: The Sequent NUMA-Q 622
 - 8.6.1 Cache Coherence Protocol 624
 - 8.6.2 Dealing with Correctness Issues 632
 - 8.6.3 Protocol Extensions 634
 - 8.6.4 Overview of NUMA-Q Hardware 635
 - 8.6.5 Protocol Interactions with SMP Node 637
 - 8.6.6 IQ-Link Implementation 639
 - 8.6.7 Performance Characteristics 641
 - 8.6.8 Comparison Case Study: The HAL S1 Multiprocessor 643
- 8.7 Performance Parameters and Protocol Performance 645
- 8.8 Synchronization 648
 - 8.8.1 Performance of Synchronization Algorithms 649
 - 8.8.2 Implementing Atomic Primitives 651
- 8.9 Implications for Parallel Software 652
- 8.10 Advanced Topics 655
 - 8.10.1 Reducing Directory Storage Overhead 655
 - 8.10.2 Hierarchical Coherence 659
- 8.11 Concluding Remarks 669
- 8.12 Exercises 672

9 Hardware/Software Trade-Offs 679

- 9.1 Relaxed Memory Consistency Models 681
 - 9.1.1 The System Specification 686
 - 9.1.2 The Programmer's Interface 694
 - 9.1.3 The Translation Mechanism 698
 - 9.1.4 Consistency Models in Real Multiprocessor Systems 698
- 9.2 Overcoming Capacity Limitations 700
 - 9.2.1 Tertiary Caches 700
 - 9.2.2 Cache-Only Memory Architectures (COMA) 701
- 9.3 Reducing Hardware Cost 705
 - 9.3.1 Hardware Access Control with a Decoupled Assist 707
 - 9.3.2 Access Control through Code Instrumentation 707
 - 9.3.3 Page-Based Access Control: Shared Virtual Memory 709
 - 9.3.4 Access Control through Language and Compiler Support 721

9.4	Putting It All Together: A Taxonomy and Simple COMA	724
9.4.1	Putting It All Together: Simple COMA and Stache	726
9.5	Implications for Parallel Software	729
9.6	Advanced Topics	730
9.6.1	Flexibility and Address Constraints in CC-NUMA Systems	730
9.6.2	Implementing Relaxed Memory Consistency in Software	732
9.7	Concluding Remarks	739
9.8	Exercises	740

10 Interconnection Network Design 749

10.1	Basic Definitions	750
10.2	Basic Communication Performance	755
10.2.1	Latency	755
10.2.2	Bandwidth	761
10.3	Organizational Structure	764
10.3.1	Links	764
10.3.2	Switches	767
10.3.3	Network Interfaces	768
10.4	Interconnection Topologies	768
10.4.1	Fully Connected Network	768
10.4.2	Linear Arrays and Rings	769
10.4.3	Multidimensional Meshes and Tori	769
10.4.4	Trees	772
10.4.5	Butterflies	774
10.4.6	Hypercubes	778
10.5	Evaluating Design Trade-Offs in Network Topology	779
10.5.1	Unloaded Latency	780
10.5.2	Latency under Load	785
10.6	Routing	789
10.6.1	Routing Mechanisms	789
10.6.2	Deterministic Routing	790
10.6.3	Deadlock Freedom	791
10.6.4	Virtual Channels	795
10.6.5	Up*-Down* Routing	796
10.6.6	Turn-Model Routing	797
10.6.7	Adaptive Routing	799
10.7	Switch Design	801
10.7.1	Ports	802
10.7.2	Internal Datapath	802
10.7.3	Channel Buffers	804

10.7.4	Output Scheduling	808
10.7.5	Stacked Dimension Switches	810
10.8	Flow Control	811
10.8.1	Parallel Computer Networks versus LANs and WANs	811
10.8.2	Link-Level Flow Control	813
10.8.3	End-to-End Flow Control	816
10.9	Case Studies	818
10.9.1	CRAY T3D Network	818
10.9.2	IBM SP-1, SP-2 Network	820
10.9.3	Scalable Coherent Interface	822
10.9.4	SGI Origin Network	825
10.9.5	Myricom Network	826
10.10	Concluding Remarks	827
10.11	Exercises	828

11 Latency Tolerance 831

11.1	Overview of Latency Tolerance	834
11.1.1	Latency Tolerance and the Communication Pipeline	836
11.1.2	Approaches	837
11.1.3	Fundamental Requirements, Benefits, and Limitations	840
11.2	Latency Tolerance in Explicit Message Passing	847
11.2.1	Structure of Communication	848
11.2.2	Block Data Transfer	848
11.2.3	Precommunication	848
11.2.4	Proceeding Past Communication in the Same Thread	850
11.2.5	Multithreading	850
11.3	Latency Tolerance in a Shared Address Space	851
11.3.1	Structure of Communication	852
11.4	Block Data Transfer in a Shared Address Space	853
11.4.1	Techniques and Mechanisms	853
11.4.2	Policy Issues and Trade-Offs	854
11.4.3	Performance Benefits	856
11.5	Proceeding Past Long-Latency Events	863
11.5.1	Proceeding Past Writes	864
11.5.2	Proceeding Past Reads	868
11.5.3	Summary	876
11.6	Precommunication in a Shared Address Space	877
11.6.1	Shared Address Space without Caching of Shared Data	877
11.6.2	Cache-Coherent Shared Address Space	879
11.6.3	Performance Benefits	891
11.6.4	Summary	896

11.7	Multithreading in a Shared Address Space	896
11.7.1	Techniques and Mechanisms	898
11.7.2	Performance Benefits	910
11.7.3	Implementation Issues for the Blocked Scheme	914
11.7.4	Implementation Issues for the Interleaved Scheme	917
11.7.5	Integrating Multithreading with Multiple-Issue Processors	920
11.8	Lockup-Free Cache Design	922
11.9	Concluding Remarks	926
11.10	Exercises	927

12 Future Directions 935

12.1	Technology and Architecture	936
12.1.1	Evolutionary Scenario	937
12.1.2	Hitting a Wall	940
12.1.3	Potential Breakthroughs	944
12.2	Applications and System Software	955
12.2.1	Evolutionary Scenario	955
12.2.2	Hitting a Wall	960
12.2.3	Potential Breakthroughs	961

Appendix: Parallel Benchmark Suites 963

A.1	ScaLapack	963
A.2	TPC	963
A.3	SPLASH	965
A.4	NAS Parallel Benchmarks	966
A.5	PARKBENCH	967
A.6	Other Ongoing Efforts	968

References 969

Index 993

Preface

Parallel computing has become a critical component of the computing technology of the 1990s, and it is likely to have as much impact over the next 20 years as microprocessors have had over the past 20. Indeed, the two technologies are deeply linked, as the evolution of highly integrated microprocessors and memory chips makes multiprocessor systems increasingly attractive. Multiprocessors already represent the high-performance end of almost every segment of the computing market, from the fastest supercomputers and largest data centers to departmental servers to the individual desktop. Tightly integrated clusters of PCs, workstations, or even multiprocessors are emerging as scalable Internet servers. In the past, computer vendors employed a range of technologies and processor architectures to provide increasing performance across their product line. Today, the same state-of-the-art microprocessor is used throughout. To obtain a significant range of performance, the primary approach is to increase the number of processors, and the economies of scale make this extremely attractive. Very soon, several processors will fit on a single chip and multiprocessors will be even more widespread than they are today.

Although parallel computing has a long and rich academic history, the close coupling with commodity technology has fundamentally changed the discipline. The emphasis on radical architectures and exotic technology has given way to quantitative analysis, the realization of different programming models on the same underlying processing nodes, and careful engineering trade-offs. Our goal in writing this book is to equip designers of the emerging class of multiprocessor systems—from modestly parallel desktop computers to highly parallel information servers and supercomputers—with an understanding of the fundamental architectural and software issues and the available techniques for addressing design trade-offs. At the same time, we hope to provide designers of software systems and applications with an understanding of the likely directions of architectural evolution, the forces that will determine the specific path that hardware designs will follow, and the impact of these developments on performance-oriented programming.

The most exciting recent development in parallel computer architecture is the convergence of traditionally disparate approaches—namely, shared memory, message-passing, data parallel, and data-driven computing—on a common machine structure. This convergence is driven partly by common technological and economic forces and partly by a better understanding of parallel software. It allows us to develop a common framework in which to understand and evaluate architectural trade-offs rather than to focus on exotic designs and taxonomies. Moreover, popular

parallel programming models are available on a wide range of machines, making parallel programming more portable and allowing meaningful benchmarks and evaluation methodologies to flourish. This maturing of the field makes it possible to undertake a quantitative as well as qualitative study of hardware/software interactions. In fact, it demands such an approach. The book follows a set of issues that are critical to all parallel architectures—data access, communication performance, coordination of cooperative work, and correct implementation of useful semantics—across the full range of modern designs. It describes the set of techniques available in hardware and in software to address each issue and explores how the various techniques interact. Carefully chosen, in-depth case studies provide a concrete illustration of the general principles and demonstrate specific interactions between mechanisms.

One of the motivations for writing this book is the lack of an adequate textbook for our own courses at Berkeley, Princeton, and Stanford. Several existing texts cover the material in a cursory fashion, summarizing various architectures and research results but not analyzing them in depth or providing a modern engineering framework. Others focus on specific projects but do not carry the principles over to alternative approaches. The research reports in the area provide a sizable body of ideas and empirical data, but it is not distilled into a coherent picture. By focusing on the salient issues in the context of technological and architectural convergence rather than on the rich and varied history that has brought us to this point, we hope to provide a deeper and more coherent understanding of this exciting and rapidly changing field. This was a deeply collaborative effort, reflected in the alternation of the order of our names on the book covers.

Intended Audience

The subject matter of this book is core material that is important for researchers, students, and practicing engineers in the fields of computer architecture, systems software, and applications. The relevance for computer architects is obvious, given the growing importance of multiprocessors. Chip designers must understand what constitutes a viable building block for multiprocessor systems. Bus and memory system design are dominated by issues related to parallelism. I/O system design must address fast scalable networks, clustering, and devices that are shared by multiple processors.

Systems software—including operating systems, compilers, programming languages, run-time systems, and performance debugging tools—needs to address new issues and will provide new opportunities in parallel computers. Thus, an understanding of architectural evolution and the forces guiding that evolution is critical. Research and development in compilers and programming languages have addressed aspects of parallel computing for some time. However, the new convergence with commodity technology suggests that these aspects may need to be reexamined and addressed in a more general context. The traditional boundaries between hardware, operating system, and user program are also shifting in the context of parallel

computing, where programs often want more direct control over resources for better performance.

Applications areas, such as computer graphics and multimedia, scientific computing, computer-aided design, databases, decision support, and transaction processing, are all likely to see a tremendous transformation as a result of the vast computing power available at low cost through parallel computing. However, developing parallel applications that are robust and that provide good parallel speedup across current and future multiprocessors is a challenging task and requires a deep understanding of system interactions and architectural directions. The book seeks to provide this understanding but also to stimulate the exchange between the applications fields and computer architecture so that better architectures can be designed—those that make the programming task easier and performance both higher and more robust.

Organization of the Book

The book is organized into 12 chapters. Chapter 1 provides an overview of parallel architecture. It opens with a discussion of why the expanding role of multiprocessors is inevitable, given current trends in technology, architecture, and applications. It briefly introduces the diverse multiprocessor architectures that have shaped the field (shared memory, message passing, data parallel, dataflow, and systolic) and shows how the technology and architectural trends are driving a convergence in the field to a set of commodity processing nodes connected by a communication architecture. This convergence does not mean the end to innovation but, on the contrary, that we will now see a time of rapid progress, as designers start talking with each other rather than past each other. The chapter develops a layered framework (including the programming model, communication abstraction, user/system interface, and hardware/software interface) for understanding wide variety of communication architectures and implementations. Viewing the convergence of the field in this framework, the last portion of the chapter lays out the fundamental design issues that must be addressed at each of the interfaces between layers: naming, ordering, replication, and communication performance (overhead, latency, and bandwidth). These issues form an underlying theme throughout the rest of this book. The chapter ends with a set of historical references.

Chapter 2 provides an introduction to the process of parallel programming. It describes a set of motivating applications for multiprocessors that are used throughout the rest of the book. It shows what parallel programs look like in the major programming models and hence what primitives a system must support. It uses the application case studies to illustrate the steps of decomposition, assignment, orchestration, and mapping in creating a parallel program and identifies the key performance goals of these steps.

Chapter 3 describes the basic techniques that good parallel programmers use to get performance out of the underlying architecture. It provides an understanding of hardware/software trade-offs and illustrates what aspects of performance can be addressed through architectural means and what aspects must be addressed either

by the compiler or the programmer. The analogy in sequential computing is that architecture cannot transform an $O(n^2)$ algorithm into an $O(n \log n)$ algorithm, but it can improve the average access time for common memory reference patterns. The chapter shows clearly the core algorithmic and programming challenges that cut across programming models as well as the model-specific orchestration issues. This material shows how architectural advance can ease the burden of effective parallel programming in addition to increasing the achievable performance. The programming techniques are a key factor in any quantitative evaluation of design trade-offs, and the chapter concludes by applying them to the motivating applications to produce high-performance versions.

Chapter 4 takes up the challenge of performing solid workload-driven evaluation of design trade-offs. Architectural evaluation is difficult even for modern uniprocessors, where we typically look at moderate design variations—such as pipeline or memory system organizations—against a fixed set of programs. In parallel architecture, we have many more degrees of freedom to explore. The interactions between aspects of the design are more profound, and the interactions between hardware and software are more significant as well as of wider scope. We are often interested in performance as the machine and the program scale, and it is impossible to scale one without affecting the other. It is easy to arrive at incomplete or even misleading conclusions if the evaluation is not methodologically sound, so the characteristics of parallel programs must be adequately understood. Chapter 4 discusses how application and architectural parameters interact and how they should be scaled together and presents benchmarks that are used throughout later chapters. It provides methodological guidelines for the evaluation of real machines and of architectural ideas through simulation. The Appendix provides additional reference material on parallel benchmarking efforts.

Chapters 5 and 6 provide a complete understanding of the bus-based, symmetric shared memory multiprocessors (SMPs) that form the bread and butter of modern commercial machines beyond the desktop. Chapter 5 presents the high-level, logical design of “snooping” bus protocols, which ensure that automatically replicated data is coherent across multiple caches. This chapter provides an important discussion of memory consistency, which brings us to terms with what shared memory really means to algorithm designers. It discusses the spectrum of design options and how machines are optimized against typical reference patterns occurring in user programs and in the operating system. Given this conceptual understanding of SMPs, the chapter reflects on the implications for parallel software, including applications and support for synchronization.

Chapter 6 examines the protocol issues in more depth as well as physical design of bus-based multiprocessors. It digs into the engineering issues that arise in supporting modern microprocessors with multilevel caches on modern buses, which are highly pipelined, as well as how the high-level protocols of the previous chapter are realized and extended on these systems. The presentation here provides a very complete understanding of the design issues in this regime. It is all the more important because these small-scale designs form a building block for large-scale designs and because many of the concepts appear later in the book on a larger scale with a

broader set of concerns. The chapter also provides self-contained case studies on the SGI Challenge and Sun Enterprise servers.

Chapters 7, 8, 9, and 10 provide a complete understanding of the scalable multiprocessor architectures that represent the high end of computing and the future of the midrange as technology continues to advance.

Chapter 7 presents the hardware organization and architecture of a range of machines that are scalable to large or very large configurations. The key organizational concept is that of a network transaction, analogous to the bus transaction that is the fundamental primitive for the smaller designs in Chapters 5 and 6. However, in scalable machines the global arbitration and globally visible information is lost and a large number of transactions can be outstanding. The chapter shows how programming models are realized in terms of network transactions and studies a spectrum of important design points organized according to the level of direct hardware interpretation of the network transaction, including case studies of the nCUBE/2, Thinking Machines CM-5, Intel Paragon, Meiko CS-2, CRAY T3D, and CRAY T3E. It examines modern clusters in this framework with case studies of the Myrinet NOW and the DEC Memory Channel. A performance comparison is conducted across these designs.

Chapter 8 puts the results of the previous chapters together to demonstrate how to realize a shared physical address space with automatic hardware replication and cache coherence on scalable systems. This style of machine is increasingly popular in the industry. The chapter provides a complete treatment of directory-based cache coherence protocols and hardware design alternatives, including case studies of the SGI Origin2000 and Sequent NUMA-Q. It examines workload behavior on these machines and extends the discussions of programming implications and synchronization.

Chapter 9 examines a spectrum of alternatives for shared address space systems that push the boundaries of hardware/software trade-offs to obtain higher performance, reduce hardware cost and complexity, or both. It covers relaxed memory consistency models, cache-only memory architectures that replicate data coherently in hardware in main memory, and software-based coherent replication. Much of this material is in the transitional phase from academic research to commercial product at the time of this writing, and its role will be further shaped as cluster technology emerges. It exposes very important design concepts not treated elsewhere in the book.

Chapter 10 addresses the design of scalable high-performance communication networks, which underlies all the scalable machines discussed in previous chapters but was deferred to complete our understanding of the processor, memory system, and network interface design that drive these networks. The chapter builds a general framework for understanding where hardware costs, transfer delays, and bandwidth restrictions arise in networks. It looks at a variety of trade-offs in routing techniques, switch design, and interconnection topology with respect to these cost-performance metrics. The trade-offs are made concrete through case studies of recent designs.

Given the foundation established by the first 10 chapters, Chapter 11 examines a set of crosscutting issues involved in tolerating the significant latencies that arise in

multiprocessor systems without impeding performance. The techniques exploit two basic capabilities: overlapping latency with useful work and pipelining the transfer of data. The simplest of these techniques are essentially bulk transfers, which pipeline the movement of a large regular sequence of data items and often can be off-loaded from the processor. The other techniques attempt to hide the latency incurred in collections of individual loads and stores. Write latencies are hidden by exploiting weak consistency models, which recognize that ordering is conveyed by only a small set of the accesses to shared memory in a program. Read latencies are hidden by implicit or explicit prefetching of data or by lookahead techniques in modern dynamically scheduled processors. Some of the techniques extend to hiding synchronization latencies as well. The chapter provides a thorough examination of these alternatives, the impact on compilation techniques, and a quantitative evaluation of effectiveness.

Finally, Chapter 12 examines the trends in technology, architecture, software systems, and applications that are likely to shape the future evolution of the field. It looks at evolutionary scenarios, walls we may hit, and potential breakthroughs from a hardware/software perspective.

Using the Book

The book is organized to meet the needs of several potential audiences. It can serve as a graduate text, a professional reference for engineers, and as a general reference for members of the technical community who find themselves dealing ever more frequently with parallel computing. There is sufficient material, if covered in full depth, for a full-year study of parallel computing, covering the entire range of machine design and practical parallel programming experience. However, it can also be used in smaller segments.

Chapter 1 is intended to provide a stand-alone, general understanding of parallel architectures as would be appropriate for a segment of a general computer architecture course at the graduate or upper-division undergraduate level. It would also be appropriate for the engineering manager or corporate executive needing to understand the vocabulary and basic concepts of parallel computing and how the technology will impact their business. It lays out clearly where to go to learn more as your interest or need to understand parallel computing increases. The chapter can also be used as a basic background in parallel architecture for compiler, database, operating system, or programming courses. Chapters 1 and 12 together provide a well-rounded “outer skin” of parallel computer architecture.

A parallel architecture course oriented toward machine organization and design is comprised of the core material of Chapters 5, 6, 7, 8, and 10, in addition to the overview of Chapter 1. However, the chapters go into greater depth of design than has been common in traditional courses because the material was not available in any published form or put together in a design-oriented framework, and they provide detailed quantitative illustrations of trade-offs. Chapters 5 and 6 develop the key requirements of correctness in cache-coherent systems and show how to satisfy them with high performance in increasingly complex designs. Chapter 7 takes apart

scalable machines in a manner not available from commercial sources or research publications and addresses emerging high-performance clusters in this framework. Chapter 8 describes the cache coherence protocols of prominent commercial distributed-memory machines in a framework and level of detail not available elsewhere. Chapter 10 provides a compact, rounded treatment of network design. The treatment is deep enough in these chapters to provide even the seasoned system designer with a new understanding and a clean design framework. A serious yet pragmatic treatment of memory consistency models is carried throughout these chapters (as well as in the first part of Chapter 9), as is a discussion of implementing synchronization operations. These chapters on machine organization and design can be supplemented with Chapter 11, which covers the increasingly important topic of latency tolerance.

The exciting opportunity presented by this text is that, with the core material packaged in a cohesive form, it becomes possible to strengthen the basic parallel architecture course along several dimensions. First, thorough coverage of Chapters 2 and 3 allows the treatment to reach across the hardware/software boundary. This gives the architecture student a much more solid grasp of the impact of architectural decisions and what parallel programming is all about. It also broadens the appeal of the course to a wider audience of operating systems, languages, and applications students who are viewing the architectural issues from a software perspective. A second dimension along which the basic course can be strengthened is quantitative performance analysis of hardware and software design decisions. Building upon a basic understanding from Chapters 2 and 3, Chapter 4, the Appendix, and the “Implications for Parallel Software” sections of the later chapters carry this thread throughout the core machine design material. They provide an informed, critical perspective with which to view published results, as well as methodological guidelines for performing evaluations. A third dimension is a sharp focus on hardware/software trade-offs. This is the underlying issue that is framed by the quantitative analysis and explored in the synchronization and programming sections of each chapter. It comes to the fore in Chapter 9, where the division of responsibilities in providing a coherent shared address space is examined in detail, and in Chapter 11 in the discussion of latency tolerance. Each of these dimensions represents a group of professionals who have an increasing need to understand more deeply how to deal with parallel architectures.

The book also serves well as the primary text for a hands-on parallel programming course. With Chapter 1 providing a general introduction, Chapters 2 and 3 offer a strong framework for how to reason about the behavior of parallel programs. This is further solidified by the workload analysis in Chapter 4 and the “Implications for Parallel Software” sections in Chapters 5, 7, 8, and 9. This material should be supplemented with a reference on the parallel programming environment used in the course, such as MPI, parallel threads, or HPF. The case studies in Chapters 6, 7, and 8 provide thorough coverage of machines similar to what students are likely to use. Chapter 11 provides a convenient framework for an examination of how best to solve the challenges of communication in parallel programming.

We believe parallel computer architecture is an exciting core field of study and practice whose importance will continue to grow. It has reached a point of maturity at which a serious textbook based on design and engineering principles makes sense. From a rich diversity of ideas and approaches, a dramatic convergence is now occurring in the field. It is time to go beyond surveying the machine landscape to an understanding of the fundamental design principles. We have intimately participated in the convergence of the field; this text arises from our experience, and we hope it conveys some of the excitement that we feel for this dynamic and growing area. Since parallel architecture does change so rapidly, case studies, performance analyses, and workloads need to be refreshed periodically. The Web page for this book will provide a repository for such timely material, as well as for additional teaching materials, and we hope that you will help contribute to that repository through the high-quality products of your courses and commercial developments. The URL for the book is www.mkp.com/pca.

We also encourage readers to report any errors or bugs so that we may correct them in subsequent printings. Please email them to pcabugs@mkp.com. Please also check the errata page at www.mkp.com/pca to see if the bug has already been reported and fixed.

Acknowledgments

This book has been in gestation in various forms for quite some time, and it has benefited from the efforts of many individuals. It had its roots in notes and slides for our parallel processing courses and in our research projects. Our students and staff have been invaluable throughout. Although this is the first edition, drafts have been available on the Web as the material was being developed. In the way of the Web, we have no idea of all the institutions around the world that have used it in courses and research, but we receive suggestions from the most exotic places. Many people have made contributions to it directly, indirectly, or even anonymously, so we would like to thank all of you.

Numerous students have improved this book by their questions, ideas, solutions, and projects. We want to thank the students in CS 258 (Parallel Processors) and CS 267 (Applications of Parallel Computers) at Berkeley, CS 598 (Parallel Computer Architecture and Programming) at Princeton, and CS 315A (Parallel Computer Architecture and Programming) and CS 315B (Parallel Programming Project) at Stanford. Special thanks go to Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Brent Chun, Seth Goldstein, Alan Mainwaring, Rich Martin, Lok Tin Liu, Steve Lummett, Chad Yoshikawa, and Frederick Chun Bong Wong at Berkeley; Angelos Bilas, Liviu Iftode, Dongming Jiang, Steven Kleinstein, Sanjeev Kumar, Hongzhang Shan, and Yuanyuan Zhou at Princeton; and Cheng Chen, John Heinlein, Moriyoshi Ohara, Evan Torrie, and Steven Cameron Woo at Stanford, all of whom have contributed valuable insight, data, and analysis to the book through their tireless efforts. Jiang, Kumar, Ohara, Torrie, Wong, and Woo deserve an especially hearty thanks for their contributions.

Many people in academia and industry provided invaluable assistance in reviewing drafts, explaining to us how things really worked, trying out the book, and guiding us along the path. We would especially like to thank Sarita Adve, Arvind, Russell Clapp, Michel Dubois, Mike Galles, Kourosh Gharachorloo, Jim Gray, John Hennessy, Mark Hill, Phil Krueger, James Laudon, Edward Lazowska, Dan Lenoski, W. R. Michalson, Todd Mowry, Greg Papadopoulos, Dave Patterson, Randy Rettberg, Shuichi Sakai, Klaus Schauser, Ashok Singhal, Burton Smith, Jim Smith, Mark Smotherman, Per Stenstrom, Thorsten von Eicken, Maurice Wilkes, David Wood, and Chengzhong Xu. Thanks, John and Dave, for guidance throughout. Many people assisted us by teaching from portions of the book, including our earliest adopters, Sarita Adve, Andrew Chien, Jim Demmel, Wallid Najjar, Constantine Polychronopoulos, Radhika Thekkath, and Kathy Yelick.

We also want to thank the National Science Foundation, the Defense Advanced Research Projects Agency, the Department of Energy, and numerous corporate sponsors for supporting the research that underlies the material in this book and the dramatic advance of parallel computing.

We wish to thank the impressive team at Morgan Kaufmann Publishers who managed to get this book all the way to the end. Denise Penrose picked up the reins and led the team with unbelievable energy, dedication, and enthusiasm. It was an absolute pleasure to work with her. Elisabeth Beller managed the entire production process very smoothly. Meghan Keeffe and Jane Elliott coordinated reviews and photo searches and tied up many a loose end. A crew of talented proofreaders kept all the right words in all the right places. Thanks also to Jennifer Mann, who managed the project before Denise joined up, and to Bruce Spatz, who has moved on from MKP since starting this book on its way.

We must also thank our university staff, Gabriela Aranda, Ginny Hogan, Chris Kranz, Terry Lessard-Smith, Bob Miller, Thoi Nguyen, Matt Norcross, Charlie Orgish, Jim Roberts, and Chris Tengi, for countless bits of help along the way.

Above all, our deepest thanks, appreciation, and love go to our families for their immeasurable support, patience, kindness, and wisdom throughout the entire process.



Introduction

For over a decade, we have enjoyed explosive growth in the performance and capability of computer systems. The theme of this dramatic success story is the advance of the underlying VLSI technology, which allows clock rates to increase and larger numbers of components to fit on a chip. The plot of this story centers on computer architecture, which translates the raw potential of the technology into greater performance and expanded capability of the computer system. The story's leading character is parallelism. A larger volume of resources means that more operations can be performed at once, in parallel. Parallel computer architecture is about organizing these resources so that they work well together. Computers of all types have harnessed parallelism more and more effectively to gain performance from the raw technology, and the level at which parallelism is exploited continues to rise. Another key character is storage. The data that is operated on at an ever faster rate must be held somewhere in the machine. Thus, the story of parallel processing is deeply intertwined with data locality and communication. The computer architect must sort out these changing relationships to design the various levels of a computer system so as to maximize performance and programmability within the limits imposed by technology and cost at any particular time.

Parallelism is a fascinating perspective from which to understand computer architecture because it applies at all levels of design, it interacts with essentially all other architectural concepts, and it presents a unique dependence on the underlying technology. In particular, the basic issues of locality, bandwidth, latency, and synchronization arise at many levels of the design of parallel computer systems. The trade-offs must be resolved in the context of real application workloads.

Parallel computer architecture, like any other aspect of design, involves elements of form and function. These elements are captured nicely in the following definition (Almasi and Gottlieb 1989):

A parallel computer is a “collection of processing elements that communicate and cooperate to solve large problems fast.”

However, this simple definition raises many questions. How large a collection are we talking about? How powerful are the individual processing elements, and can the number be increased in a straightforward manner? How do these elements communicate and cooperate? How is data transmitted between processors, what sort of interconnection is provided, and what operations are available to sequence the actions carried out on different processors? What are the primitive abstractions that

the hardware and software provide to the programmer? And finally, how does it all translate into performance? In answering these questions, we will see that small, moderate, and very large collections of processing elements each have important roles to fill in modern computing. Thus, it is important to understand parallel machine design across the scale, from the small to the very large. Some design issues apply throughout the scale of parallelism; others are most germane to a particular regime, such as within a chip, within a box, or on a very large machine. It is safe to say that parallel machines occupy a rich and diverse design space. This diversity makes the area exciting, but it also means that it is important that we develop a clear framework in which to understand the many design alternatives.

Parallel architecture is itself changing rapidly. Historically, parallel machines have demonstrated innovative organizational structures, often tied to specific programming models, as architects sought to obtain the ultimate in performance out of a given technology. In many cases, radical organizations were justified on the grounds that advances in the base technology would eventually run out of steam. These dire predictions appear to have been overstated, as logic densities and switching speeds have continued to improve and more modest parallelism has been employed at lower levels to sustain continued improvement in processor performance. Nonetheless, application demand for computational performance continues to outpace what individual processors can deliver, and multiprocessor systems occupy an increasingly important place in mainstream computing. What has changed is the novelty of these parallel architectures. Even large-scale parallel machines today are built out of the same basic components as workstations and personal computers. They are subject to the same engineering principles and cost-performance trade-offs. Moreover, to yield the utmost in performance, a parallel machine must extract the full performance potential of its individual components. Thus, an understanding of modern parallel architectures must include an in-depth treatment of engineering trade-offs, not just a descriptive taxonomy of possible machine structures.

Parallel architectures will play an increasingly central role in information processing. This view is based not so much on the assumption that individual processor performance will soon reach a plateau but rather on the estimation that the next level of system design, the multiprocessor level, will become increasingly attractive with increases in chip density. *The goal of this book is to articulate the principles of computer design at the multiprocessor level.* It examines the design issues present for each of the system components—processors, memory systems, and networks—and the relationships between these components. A key aspect is understanding the division of responsibilities between hardware and software in evolving parallel machines. Understanding this division requires familiarity with the requirements that parallel programs place on the machine and the interaction of machine design and the practice of parallel programming.

The process of learning computer architecture is frequently likened to peeling an onion, and this analogy is even more appropriate for parallel computer architecture. At each level of understanding we find a complete whole with many interacting facets, including the structure of the machine, the abstractions it presents, the tech-

nology it rests upon, the software that exercises it, and the models that describe its performance. However, if we dig deeper into any of these facets, we discover another layer of design and a new set of interactions. The holistic, multilevel nature of parallel computer architecture makes the field challenging to learn and challenging to present. Some sense of the layer-by-layer structure is unavoidable.

This introductory chapter presents the “outer skin” of parallel computer architecture. It first outlines the reasons why parallel machine design may become pervasive, from desktop machines to supercomputers. It also examines the technological, architectural, and economic trends that have led to the current state of computer architecture and that provide the basis for anticipating future parallel architectures. Section 1.1 focuses on the forces that have brought about the dramatic advance of processor performance and the restructuring of the entire computing industry around commodity microprocessors. These forces include the insatiable application demand for computing power, the continued improvements in the density and level of integration in VLSI chips, and the utilization of parallelism at higher and higher levels of the architecture.

Next is a quick look at the spectrum of important architectural styles, which give the field such a rich history and contribute to the modern understanding of parallel machines. Within this diversity of design, a common set of design principles and trade-offs arise, driven by the same advances in the underlying technology. These forces are rapidly leading to a convergence in the field, which forms the emphasis of this book. Section 1.2 surveys traditional parallel machines, including shared memory, message passing, data parallel, systolic arrays, and dataflow, and illustrates the different ways that they address common architectural issues. The discussion shows the dependence of parallel architecture on the underlying technology and, more importantly, demonstrates the convergence that has come about with the dominance of microprocessors.

Building on this convergence, Section 1.3 examines the fundamental design issues that cut across parallel machines: what can be named at the machine level as a basis for communication and coordination, what is the latency or time required to perform these operations, and what is the bandwidth or overall rate at which they can be performed? This shift from conceptual structure to performance components provides a framework for quantitative, rather than merely qualitative, study of parallel computer architecture.

With this initial broad understanding of parallel computer architecture in place, the following chapters dig deeper into its technical substance. Chapters 2 and 3 delve into the structure and requirements of parallel programs to provide a basis for understanding the interaction between parallel architecture and applications. Chapter 4 builds a framework for evaluating design decisions in terms of application requirements and performance measurements. Chapters 5 and 6 are a complete study of parallel computer architecture at the limited scale employed widely in commercial multiprocessors—from a few processors to a few tens of processors. The concepts and structures introduced here form the building blocks for more aggressive large-scale designs presented over the final five chapters.

1.1 WHY PARALLEL ARCHITECTURE

Computer architecture, technology, and applications evolve together and have very strong interactions. Parallel computer architecture is no exception. A new dimension is added to the design space—the number of processors—and the design is even more strongly driven by the demand for performance at acceptable cost. Whatever the performance of a single processor at a given time, higher performance can, in principle, be achieved by utilizing many such processors. How much additional performance is gained and at what additional cost depends on a number of factors, which we will explore throughout the book.

To better understand this interaction, let us consider the performance characteristics of the processor building blocks. Figure 1.1¹ illustrates the growth in processor performance over time for several classes of computers (Hennessy and Jouppi 1991). The dashed extensions of the trend lines represent a naive extrapolation of the trends. Although we should be careful in drawing sharp quantitative conclusions from such limited data, the figure suggests several valuable observations.

First, the performance of the highly integrated, single-chip CMOS microprocessor is steadily increasing and is surpassing the larger, more expensive alternatives. Microprocessor performance has been improving at a rate of about 50% per year. The advantages of using small, inexpensive, low-power, mass-produced processors as the building blocks for computer systems with many processors are intuitively clear. However, until recently the performance of the processor best suited to parallel architecture was far behind that of the fastest single-processor system. This is no longer true. Although parallel machines have been built at various scales since the earliest days of computing, the approach is more viable today than ever before because the basic processor building block is better suited to the job.

The second and perhaps more fundamental observation is that change, even dramatic change, is the norm in computer architecture. The continuing process of change has profound implications for the study of computer architecture because we need to understand not only how things are but how they might evolve and why. Change is one of the key challenges in writing this book—and one of the key motivations. Parallel computer architecture has matured to the point where it needs to be studied from a basis of engineering principles and quantitative evaluation of performance and cost. These are rooted in a body of facts, measurements, and designs of real machines. Unfortunately, existing data and designs are necessarily frozen in time

1. The figure is drawn from an influential paper that sought to explain the dramatic changes taking place in the computing industry (Hennessy and Jouppi 1991). The metric of performance is a bit tricky because it reaches across such a range of time and market segment. The study draws data from general-purpose benchmarks, such as the SPEC benchmark, which is widely used to assess performance on technical computing applications (Hennessy and Patterson 1996). After publication, microprocessors continued to track the prediction while mainframes and supercomputers went through tremendous crises and emerged using multiple CMOS microprocessors in their market niche.

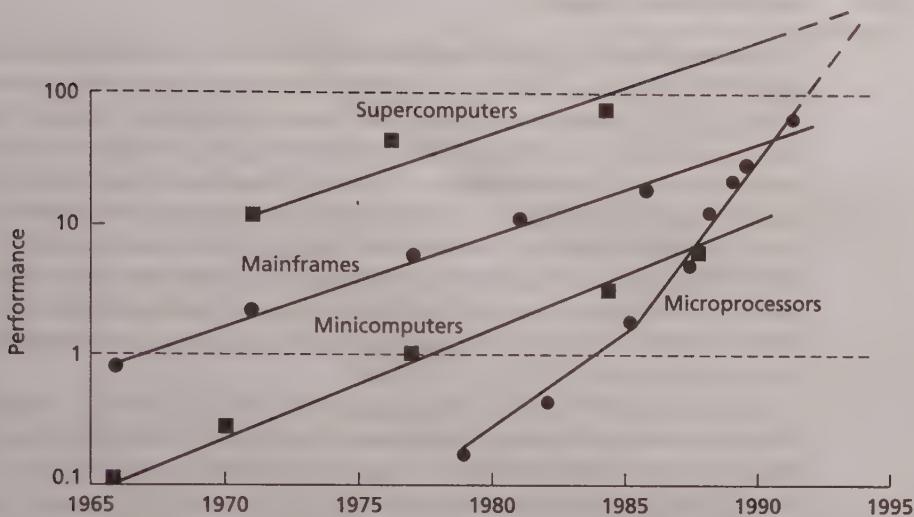


FIGURE 1.1 Performance trends over time of micros, minicomputers, mainframes, and supercomputers. Performance of microprocessors has been increasing at a rate of about 50% per year since the mid-1980s. More traditional mainframe and supercomputer performance has been increasing at a rate of roughly 25% per year. As a result, we are seeing the processor that is best suited to parallel architecture become the performance leader as well. Source: Hennessy and Jouppi (1991).

and will become dated as the field progresses. This book presents hard data and examines real machines in the form of a late 1990s technological snapshot in order to retain a clear grounding. However, the methods of evaluation underlying the analysis of concrete design trade-offs transcend the chronological and technological reference point of the book.

The late 1990s happens to be a particularly interesting snapshot because we are in the midst of a dramatic technological realignment as the single-chip microprocessor is poised to dominate every sector of computing and as parallel computing takes hold in many areas of mainstream computing. Of course, the prevalence of change suggests being cautious about extrapolating into the future. The remainder of this section examines more deeply the forces and trends that are giving parallel architectures an increasingly important role throughout the computing field and pushing parallel computing into the mainstream. It looks first at the application demand for increased performance and then at the underlying technological and architectural trends that strive to meet these demands. We see that parallelism is inherently attractive as computers become more highly integrated and that it is being exploited at increasingly high levels of the design. Finally, this section closes with a look at the role of parallelism in the machines at the very high end of the performance spectrum.

1.1.1 Application Trends

The demand for ever greater application performance is a familiar feature of every aspect of computing. Advances in hardware capability enable new application functionality, which grows in significance and places even greater demands on the architecture. This cycle drives the tremendous ongoing design, engineering, and manufacturing effort underlying the sustained exponential performance increase in microprocessor performance. It drives parallel architecture even harder since parallel architecture focuses on the most demanding of these applications. With a 50% annual improvement in processor performance, a parallel machine of a hundred processors can be viewed as providing to applications the computing power that will be widely available 10 years in the future, whereas a thousand processors reflects nearly a 20-year horizon.

Application demand also leads computer vendors to provide a range of models with increasing performance and capacity at progressively increasing cost. The largest volume of machines and the greatest number of users are at the low end, whereas the most demanding applications are served by the high end. One effect of this “platform pyramid” is that the pressure for increased performance is greatest at the high end and is exerted by an important minority of the applications. Prior to the microprocessor era, greater performance was obtained through exotic circuit technologies and machine organizations. Today, to obtain performance significantly greater than the state-of-the-art microprocessor, the primary option is multiple processors, and the most demanding applications are written as parallel programs. Thus, parallel architectures and parallel applications are subject to the most acute demands for greater performance.

A key reference point for both the architect and the application developer is how the use of parallelism improves the performance of the application. We may define the *speedup* on p processors as

$$\text{Speedup}(p \text{ processors}) \equiv \frac{\text{Performance}(p \text{ processors})}{\text{Performance}(1 \text{ processor})} \quad (1.1)$$

For a single, fixed problem, the performance of the machine on the problem is simply the reciprocal of the time to complete the problem, so we have the following important special case:

$$\text{Speedup}_{\text{fixed problem}}(p \text{ processors}) = \frac{\text{Time}(1 \text{ processor})}{\text{Time}(p \text{ processors})} \quad (1.2)$$

Scientific and Engineering Computing

The direct reliance on increasing levels of performance is well established in a number of endeavors but is perhaps most apparent in the fields of computational science and engineering. Basically, in these fields computers are used to simulate physical phenomena that are impossible or very costly to observe through empirical means.

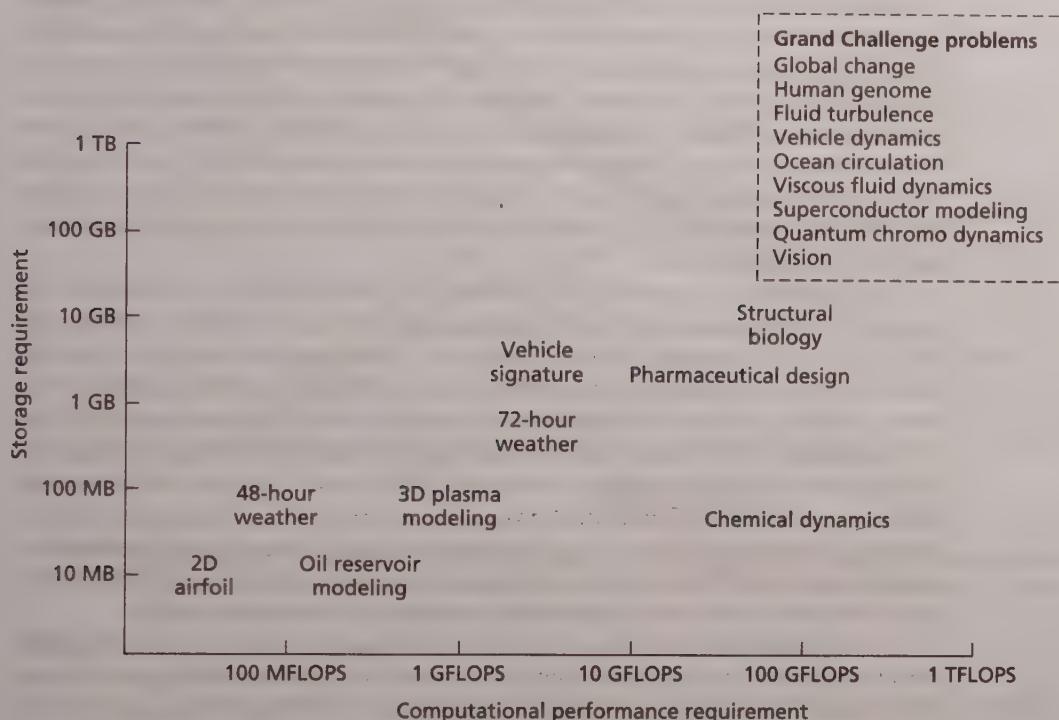


FIGURE 1.2 Grand Challenge application requirements. A collection of important scientific and engineering problems is positioned in a space defined by computational performance and storage capacity. Given the exponential growth rate of performance and capacity, both of these axes map directly to time. In the upper right corner appear some of the Grand Challenge applications identified by the U.S. High Performance Computing and Communications program.

Typical examples include modeling global climate change over long periods, the evolution of galaxies, the atomic structure of materials, the efficiency of combustion with an engine, the flow of air over surfaces of vehicles, the damage due to impacts, and the behavior of microscopic electronic devices. Computational modeling allows in-depth analyses to be performed cheaply on hypothetical designs through computer simulation. A direct correspondence can be drawn between levels of computational performance and the problems that can be studied through simulation. Figure 1.2 summarizes the 1993 findings of the Committee on Physical, Mathematical, and Engineering Sciences of the federal Office of Science and Technology Policy (1993). It indicates the computational rate and storage capacity required to tackle a number of important science and engineering problems. Even with dramatic increases in processor performance, very large parallel architectures are needed to address these problems in the near future. Some years further down the road, new grand challenges will be in view.

Parallel architectures have become the mainstay of scientific computing, including physics, chemistry, material science, biology, astronomy, earth sciences, and others. The engineering application of these tools for modeling physical phenomena is now essential to many industries, including petroleum (reservoir modeling), automotive (crash simulation, drag analysis, combustion efficiency), aeronautics (airflow analysis, engine efficiency, structural mechanics, electromagnetism), pharmaceutical (molecular modeling), and others. In almost all of these applications, there is a large demand for visualization of the results, which is itself a demanding application amenable to parallel computing.

The visualization component has brought the traditional areas of scientific and engineering computing closer to the entertainment industry. In 1995, the first full-length, computer-animated motion picture, *Toy Story*, was produced on a parallel computer system composed of hundreds of Sun workstations. This application was finally possible because the underlying technology and architecture crossed three key thresholds: the decreased cost of computing allowed the rendering to be accomplished within the budget typically associated with a feature film, and the increase in both the performance of individual processors and the scale of parallelism made it possible to complete the task in a reasonable amount of time (several months on several hundred processors). Each science and engineering application has an analogous threshold of computing capacity and cost at which it becomes viable.

Let us take an example from the Grand Challenge program to help understand the strong interaction between applications, architecture, and technology in the context of parallel machines. A 1995 study (Pfeiffer et al. 1995) examined the effectiveness of a wide range of parallel machines on a variety of applications, including a molecular dynamics package, known as AMBER (Assisted Model Building through Energy Refinement). AMBER is widely used to simulate the motion of large biological models such as proteins and DNA, which consist of sequences of residues (amino acids and nucleic acids, respectively) each composed of individual atoms. The code was developed on CRAY vector supercomputers, which employ custom processors, large and expensive SRAM memories (instead of caches), and machine instructions that perform arithmetic or data movement on a sequence, or *vector*, of data values. Figure 1.3 shows the speedup obtained on three versions of this code on a 128-processor microprocessor-based machine—the Intel Paragon, described later. The particular test problem involved the simulation of a protein solvated by water. This test consisted of 99 amino acids and 3,375 water molecules for approximately 11,000 atoms.

The initial parallelization of the code (version 8/94) resulted in good speedup for small configurations but poor speedup on larger configurations. A modest effort to improve the balance of work done by each processor, using techniques discussed in Chapter 2, improved the scaling of the application significantly (version 9/94). An additional effort to optimize communication produced a highly scalable code (version 12/94). This 128-processor version achieved a performance of 406 MFLOPS; the best previously achieved was 145 MFLOPS on a CRAY C90 vector processor. The same application on a more efficient parallel architecture, the CRAY T3D, achieved 891 MFLOPS on 128 processors. This sort of learning curve is quite typical in the

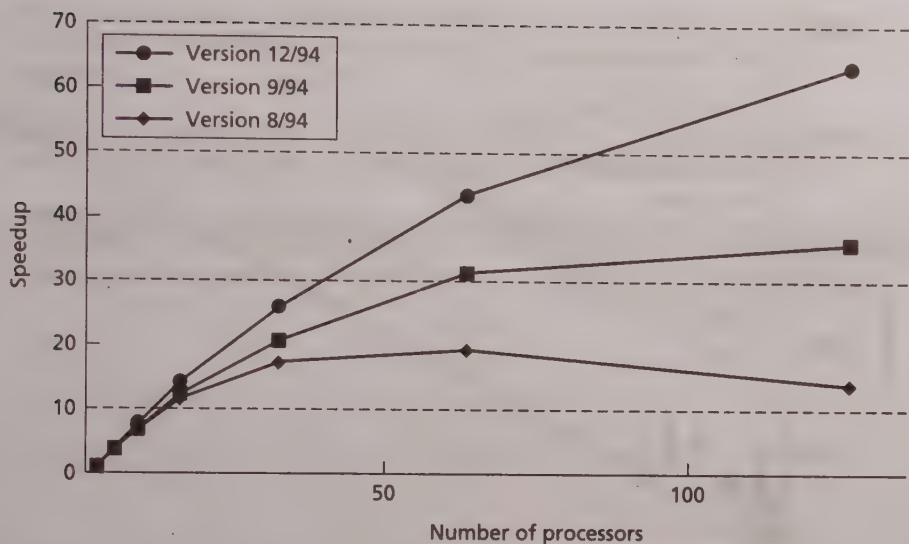


FIGURE 1.3 Speedup on three versions of a parallel program. The parallelization learning curve is illustrated by the speedup obtained on three successive versions of this molecular dynamics code on the Intel Paragon.

parallelization of important applications, as is the interaction between application and architecture. The application writer typically studies the application to understand the demands it places on the available architectures and how to improve its performance on a given set of machines. The architect may study these demands as well in order to understand how to make the machine more effective on a given set of applications. Ideally, the end user of the application enjoys the benefits of both efforts.

The demand for ever increasing performance is a natural consequence of the modeling activity. For example, in electronic CAD there is obviously more to simulate as the number of devices on the chip increases. In addition, the increasing complexity of the design requires that more test vectors be used and, because higher-level functionality is incorporated into the chip, each of these tests must run for a larger number of clock cycles. Furthermore, an increasing level of confidence is required because the cost of fabrication is so great. The cumulative effect is that the computational demand for the design verification of each new generation is increasing at an even faster rate than the performance of the microprocessors themselves.

Commercial Computing

Commercial computing has also come to rely on parallel architectures for its high end. Although the scale of parallelism is typically not as large as in scientific computing, the use of parallelism is even more widespread. Multiprocessors have provided the high end of the commercial computing market since the mid-1960s. In

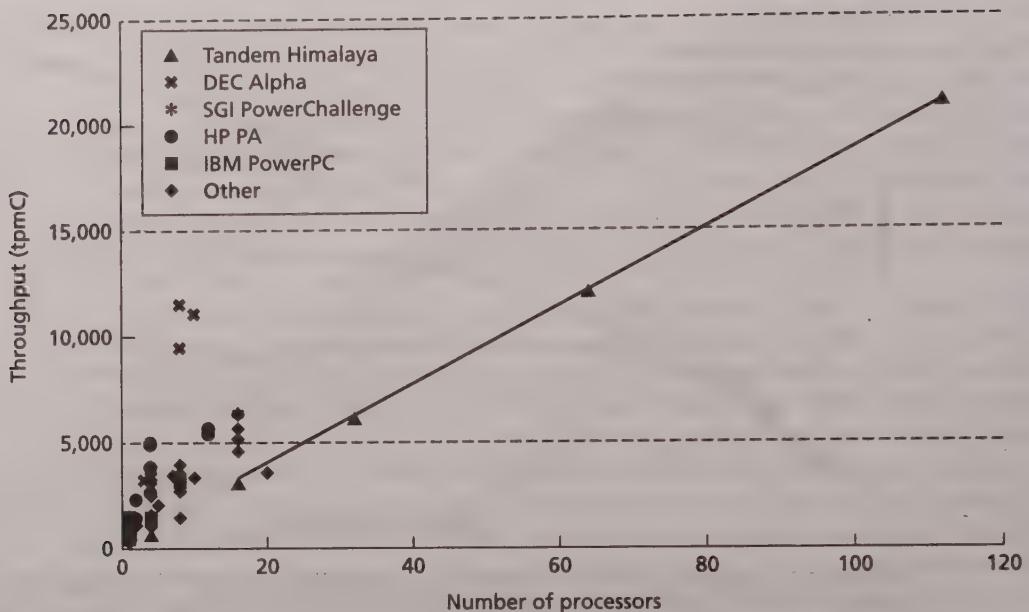


FIGURE 1.4 TPC-C throughput versus number of processors on TPC. The March 1996 TPC report documents the transaction processing performance for a wide range of systems. The figure shows the number of processors employed for all of the high-end systems, highlighting five leading vendor product lines. All of the major database vendors utilize multiple processors for their high-performance options, although the scale of parallelism varies considerably.

this arena, computer system speed and capacity translate directly into the scale of business that can be supported by the system. The relationship between performance and scale of business enterprise is clearly articulated in the on-line transaction processing (OLTP) benchmarks sponsored by the Transaction Processing Performance Council (TPC). These benchmarks rate the performance of a system in terms of its throughput in *transactions per minute* (tpm) on a typical workload. The TPC-C benchmark is an order entry application with a mix of interactive and batch transactions, including realistic features like queued transactions, aborting transactions, and elaborate presentation features (Gray 1991). The benchmark includes explicit scaling criteria to make the problem more realistic: the size of the database and the number of terminals in the system increase as the tpmC (the tpm on TPC-C) rating rises. Thus, a faster system must operate on a larger database and service a larger number of users.

Figure 1.4 shows the tpmC ratings for the collection of systems appearing in one edition of the TPC results (March 1996), with the achieved throughput on the vertical axis and the number of processors employed in the server along the horizontal axis. This data includes a wide range of systems from a variety of hardware and software vendors, a few of which are highlighted here. Since the problem solved in the benchmark run scales with system performance, we cannot simply compare times to

see the effectiveness of parallelism. Instead, we use the throughput of the system as the metric of performance in Equation 1.1. The resulting speedup is illustrated in Example 1.1.

EXAMPLE 1.1 The tpmC for the Tandem Himalaya and IBM PowerPC systems are given in the following table. What is the speedup obtained on each?

tpmC		
Number of Processors	IBM RS6000 PowerPC	Himalaya K10000
1	735	
4	1,438	
8	3,119	
16		3,043
32		6,067
64		12,021
112		20,918

Answer For the IBM system, we may calculate speedup relative to the uniprocessor system; in the Tandem case, we can only calculate speedup relative to a 16-processor system. The IBM machine appears to carry a significant penalty in the parallel database implementation of moving from one to four processors; however, the scaling is very good (superlinear) from four to eight processors. The Tandem system achieves good scaling, although the speedup appears to flatten toward the 100-processor regime. ■

Speedup _{tpmC}		
Number of Processors	IBM RS6000 PowerPC	Himalaya K10000
1	1	
4	1.96	
8	4.24	
16		1
32		1.99
64		3.95
112		6.87

Several important observations can be drawn from the TPC data. First, the use of parallel architectures is prevalent. Essentially all of the vendors supplying database hardware or software offer multiprocessor systems that provide performance substantially beyond their uniprocessor product. Second, it is not only large-scale parallelism that is important but modest-scale multiprocessor servers with tens of processors and even small-scale multiprocessors with two or four processors.

Finally, even a set of well-documented measurements of a particular class of system at a specific point in time cannot provide a true technological snapshot. Technology evolves rapidly, systems take time to develop and deploy, and real systems have a useful lifetime. Thus, the best systems available from a collection of vendors will be at different points in their life cycle at any time. For example, the DEC Alpha and IBM PowerPC systems in the March 1996 TPC report were much newer than the Tandem Himalaya system. Furthermore, we cannot conclude, for example, that the Tandem system is inherently less efficient as a result of its scalable design. We can, however, conclude that even very large-scale systems must track the technology to retain their advantage.

The transition to parallel programming, including new algorithms or attention to communication and synchronization requirements in existing algorithms, has largely taken place in the high-performance end of computing. The transition is in progress among the much broader base of commercial engineering software. Typically, engineering and commercial applications target more modest-scale multiprocessors, which dominate the server market. In the commercial world, all of the major database vendors support parallel machines for their high-end products. Several major database vendors also offer “shared-nothing” versions for large parallel machines and collections of workstations on a fast network, often called *clusters*. In addition, multiprocessor machines are heavily used to improve throughput on multiprogramming workloads. Even the desktop demonstrates a significant number of concurrent processes, with a host of active windows and daemons. Quite often a single user will have tasks running on many machines within the local area network or will farm tasks out across the network. All of these trends provide a solid application demand for parallel architectures of a variety of scales.

1.1.2 Technology Trends

The importance of parallelism in meeting the application demand for ever greater performance can be brought into sharper focus by looking more closely at the advancements in the underlying technology and architecture. These trends suggest that it may be increasingly difficult to “wait for the single processor to get fast enough” while parallel architectures become more attractive. Moreover, the examination shows that the critical issues in parallel computer architecture are fundamentally similar to those that we wrestle with in “sequential” computers, such as how the resource budget should be divided among functional units that do the work, caches that exploit locality, and wires that provide communication bandwidth.

The primary technological advance is a steady reduction in the basic VLSI feature size. This makes transistors, gates, and circuits faster and smaller, so more fit in the same area. In addition, the useful die size is growing, so there is more area to use. Intuitively, clock rate improves in proportion to the improvement in feature size while the number of transistors grows as the square, or even faster, due to increasing overall die area. Thus, in the long run, the use of many transistors at once (i.e., parallelism) can be expected to contribute more than clock rate to the observed performance improvement of the single-chip building block.

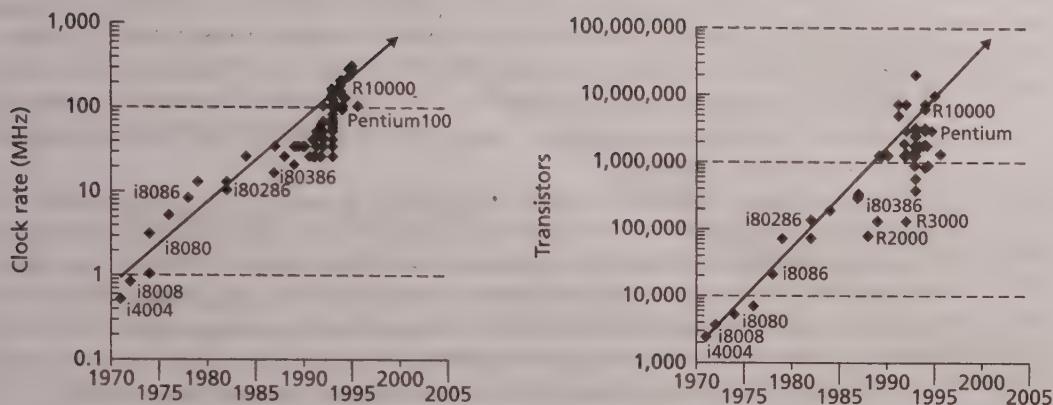


FIGURE 1.5 Improvement in logic density and clock frequency of microprocessors. Improvements in lithographic technique, process technology, circuit design, and datapath design have yielded a sustained improvement in logic density and clock rate.

This intuition is borne out by examination of commercial microprocessors. Figure 1.5 shows the increase in clock frequency and transistor count for several important microprocessor families. Clock rates for the leading microprocessors increase by about 30% per year while the number of transistors increases by about 40% per year. Thus, if we look at the raw computing power of a chip (total transistors switching per second), transistor capacity has contributed an order of magnitude more than clock rate over the past two decades.² The performance of microprocessors on standard benchmarks has been increasing at a much greater rate than clock frequency. The most widely used benchmark for measuring workstation performance is the SPEC suite, which includes several realistic integer programs and floating-point programs (SPEC 1995). Integer performance on SPEC has been increasing at about 55% per year and floating-point performance at 75% per year. The LINPACK benchmark (Dongarra 1994) is the most widely used metric of performance on numerical applications. LINPACK floating-point performance has been increasing at more than 80% per year. Thus, processors are getting faster in large part by making more effective use of an ever larger volume of computing resources.

The simplest analysis of these technology trends suggests that the basic single-chip building block will provide increasingly large capacity—in the vicinity of 100 million transistors by the year 2000. This raises the possibility of placing more of the computer system on the chip, including memory and I/O support, or of placing multiple processors on the chip (Gwennap 1994a). The former yields a small and

2. There are many reasons why the transistor count does not increase as the square of the clock rate. One is that much of the area of a processor is consumed by wires, serving to distribute control, data, or clock (i.e., on-chip communication). We will see that the communication issue reappears at every level of parallel computer architecture.

conveniently packaged building block for parallel architectures. The latter brings parallel architecture into the single-chip regime (Gwennap 1994b). Both possibilities are in evidence commercially, with the system-on-a-chip becoming first established in embedded systems, portables, and low-end personal computer products. The use of multiple processors on a chip is becoming established in digital signal processing (Feigel 1994).

The divergence between capacity and speed is much more pronounced in memory technology. From 1980 to 1995, the capacity of a DRAM chip increased a thousand-fold, quadrupling every three years, while the memory cycle time improved by only a factor of two. In the time frame of the 100-million-transistor microprocessor, we anticipate gigabit DRAM chips, but the gap between processor cycle time and memory cycle time will have grown substantially wider. Thus, the memory bandwidth demanded by the processor (bytes per memory cycle) is growing rapidly.

The latency of a memory operation is determined by the access time, which is smaller than the memory cycle time, but still the number of processor cycles per memory access time is large and increasing. To reduce the average latency experienced by the processor and to increase the bandwidth that can be delivered to the processor, we must make more effective use of the levels of the memory hierarchy that lie between the processor and the DRAM memory. Essentially all modern microprocessors provide one or two levels of caches on chip, and most system designs provide an additional level of external cache. A fundamental question as we move into multiprocessor designs is how to organize the collection of caches that lies between the many processors and the many memory modules. For example, one of the immediate benefits of parallel architectures is that the total size of each level of the memory hierarchy can increase with the number of processors without increasing the access time.

Extending these observations to disks, we see a similar divergence. Parallel disk storage systems, such as RAID, are becoming the norm. Large, multilevel caches for files or disk blocks are predominant.

1.1.3 Architectural Trends

Advances in technology determine what is possible; architecture translates the potential of the technology into performance and capability. Fundamentally, the two ways in which a larger volume of resources (e.g., more transistors) improves performance are parallelism and locality. Moreover, these two approaches compete for the same resources. Whenever multiple operations are performed in parallel, the number of cycles required to execute the program is reduced. However, resources are required to support each of the simultaneous activities. Whenever data references are performed close to the processor, the latency of accessing deeper levels of the storage hierarchy is avoided and the number of cycles to execute the program is reduced. However, resources are required to provide this local storage. In general, the best performance is obtained by an intermediate strategy that devotes resources to exploiting a degree of parallelism and a degree of locality. Indeed, we will see throughout the book that parallelism and locality interact in interesting ways in sys-

tems of all scales, from within a chip to across a large parallel machine. In current microprocessors, the die area is divided roughly equally between cache storage, processing, and off-chip interconnect. Larger-scale systems may exhibit a somewhat different split because of differences in cost and performance trade-offs, but the basic issues are the same.

Microprocessor Design Trends¹

Examining the trends in microprocessor architecture helps build intuition toward the issues we will be dealing with in parallel machines. It also illustrates how fundamental parallelism is to conventional computer architecture and how current architectural trends are leading toward multiprocessor designs. (The discussion of processor design techniques in this book is cursory since many readers are expected to be familiar with those techniques from traditional architecture texts [Hennessy and Patterson 1996] or the many discussions in the trade literature. It does provide a unique perspective on those techniques, however, and will serve to refresh your memory.)

The history of computer architecture has traditionally been divided into four generations identified by the basic logic technology: tubes, transistors, integrated circuits, and VLSI. The entire period covered by the figures in this chapter is lumped into the fourth, or VLSI, generation. Clearly, there has been tremendous architectural advance over this period, but what delineates one era from the next within this generation? The strongest delineation is the kind of parallelism that is exploited as indicated in Figure 1.6.

The period up to about 1986 is dominated by advancements in *bit-level parallelism*, with 4-bit microprocessors replaced by 8-bit, 16-bit, and so on. Doubling the width of the datapath reduces the number of cycles required to perform a full 32-bit operation. Once a 32-bit word size is reached in the mid-1980s, this trend slows, with only partial adoption of 64-bit operation obtained a decade later. Further increases in word width will be driven by demands for improved floating-point representation and a larger address space rather than performance. With address space requirements growing by less than a bit per year, the demand for 128-bit operation appears to be well in the future. The early microprocessor period was able to reap the benefits of the easiest form of parallelism: bit-level parallelism in every operation. The dramatic inflection point in the microprocessor growth curve shown in Figure 1.1 marks the arrival in 1986 of full 32-bit word operation combined with the prevalent use of caches.

The period from the mid-1980s to the mid-1990s is dominated by advancements in *instruction-level parallelism*, performing portions of several machine instructions concurrently. Full-word operation meant that the basic steps in instruction processing (instruction decode, integer arithmetic, and address calculation) could each be performed in a single cycle; with caches, the instruction fetch and data access could also be performed in a single cycle most of the time. The RISC approach demonstrated that, with care in the instruction set design, it was straightforward to pipeline the stages of instruction processing so that an instruction is executed almost every

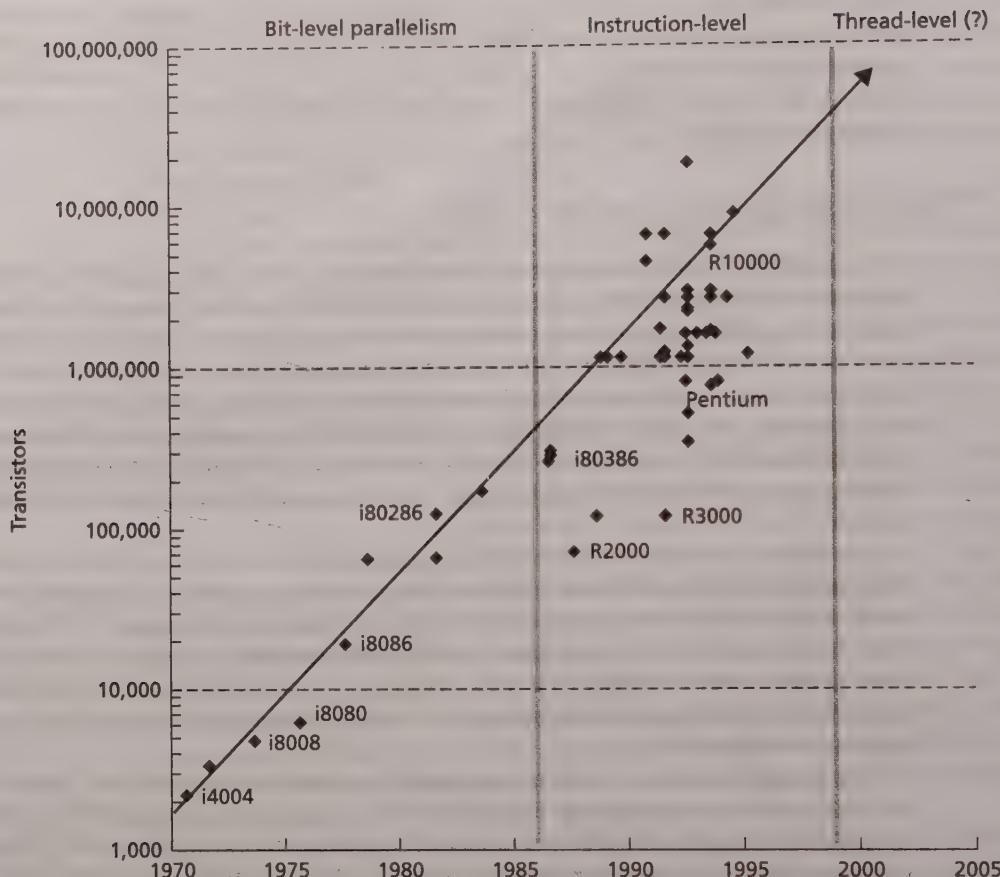


FIGURE 1.6 Number of transistors per processor chip over the last 25 years. The growth essentially follows Moore's Law, which says that the number of transistors doubles every two years. Forecasting from past trends, we can reasonably expect to be designing for a 50- to 100-million-transistor budget at the end of the decade. Also indicated are the epochs of design within the fourth, or VLSI, generation of computer architecture, reflecting the increasing level of parallelism.

cycle, on average. Thus, the parallelism inherent in the steps of instruction processing could be exploited across a small number of instructions. While pipelined instruction processing was not new, it had never before been so well suited to the underlying technology. In addition, advances in compiler technology made instruction pipelines more effective.

The mid-1980s microprocessor-based computers consisted of a small constellation of chips: an integer processing unit, a floating-point unit, a cache controller, and SRAMs for the cache data and tag storage. As chip capacity increased, these components were coalesced into a single chip, which reduced the cost of communicating among them. Thus, a single chip contained separate hardware for integer

arithmetic, memory operations, branch operations, and floating-point operations. In addition to pipelining individual instructions, it became very attractive to fetch multiple instructions at a time and issue them in parallel to distinct function units whenever possible. This form of instruction-level parallelism came to be called *superscalar* execution. It provided a natural way to exploit the ever increasing number of available chip resources. More function units were added, more instructions were fetched at a time, and more instructions could be issued in each clock cycle to the function units.

However, increasing the amount of instruction-level parallelism that the processor can exploit is only worthwhile if the processor can be supplied with instructions and data fast enough to keep it busy. In order to satisfy the increasing instruction and data bandwidth requirement, larger and larger caches were placed on chip with the processor, further consuming the ever increasing number of transistors. With the processor and cache on the same chip, the path between the two could be made very wide to satisfy the bandwidth requirement of multiple instruction and data accesses per cycle. However, as more instructions are issued each cycle, the performance impact of each control transfer and each cache miss becomes more significant. A control transfer may have to wait for the depth, or *latency*, of the processor pipeline until a particular instruction reaches the end of the pipeline and determines which instruction to execute next. Similarly, instructions that use a value loaded from memory may cause the processor to wait for the latency of a cache miss.

Processor designs in the 1990s deploy a variety of complex instruction processing mechanisms in an effort to reduce the performance degradation resulting from latency in “wide-issue” superscalar processors. Sophisticated branch prediction techniques are used to avoid pipeline latency by guessing the direction of control flow before branches are actually resolved. Larger, more sophisticated caches are used to avoid the latency of cache misses. Instructions are scheduled dynamically and allowed to complete out of order so if one instruction encounters a miss, other instructions can proceed ahead of it as long as they do not depend on the result of the instruction. A larger window of instructions that are waiting to issue is maintained within the processor and whenever an instruction produces a new result, several waiting instructions may be issued to the function units. These complex mechanisms allow the processor to tolerate the latency of a cache miss or pipeline dependence when it does occur. However, each of these mechanisms places a heavy demand on chip resources and carries a very heavy design cost.

Given the expected increases in chip density, the natural question to ask is how far will instruction-level parallelism go within a single thread of control? At what point will the emphasis shift to supporting the higher levels of parallelism available as multiple processes or multiple threads of control within a process, that is, *thread-level parallelism*? Several research studies have sought to answer the first part of the question, either through simulation of aggressive machine designs (Chang et al. 1991; Horst, Harris, and Jardine 1990; Lee, Kwok, and Briggs 1991; Melvin and Patt 1991) or through analysis of the inherent properties of programs (Butler et al. 1991; Jouppi and Wall 1989; Johnson 1991; Smith, Johnson, and Horowitz 1989; Wall 1991). The most complete treatment appears in Johnson’s book devoted to the topic

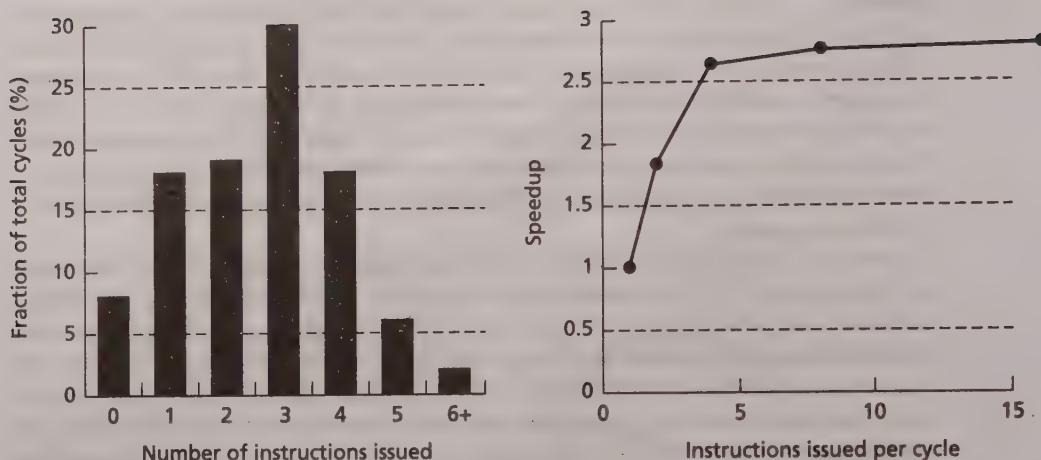


FIGURE 1.7 Distribution of potential instruction-level parallelism and estimated speedup under idealized superscalar execution. The figure shows the distribution of available instruction-level parallelism and maximum potential speedup under idealized superscalar execution, including unbounded processing resources and perfect branch prediction. Data is an average of that presented for several benchmarks by Johnson (1991).

(1991). Simulation of aggressive machine designs generally shows that two-way superscalar, that is, issuing two instructions per cycle, is very profitable and four-way offers substantial additional benefit, but wider issue widths (e.g., eight-way superscalar) provide little additional gain. The design complexity increases dramatically because control transfers occur roughly once in five instructions, on average.

To estimate the maximum potential speedup that can be obtained by issuing multiple instructions per cycle, the execution trace of a program is simulated on an ideal machine with unlimited instruction fetch bandwidth, as many function units as the program can use, and perfect branch prediction. (The latter is easy, since the trace correctly follows each branch.) These generous machine assumptions ensure that no instruction is held up because a function unit is busy or because the instruction is beyond the lookahead capability of the processor. Furthermore, to ensure that no instruction is delayed because it updates a location that is used by logically previous instructions, storage resource dependences are removed by a technique called *renaming*. Each update to a register or memory location is treated as introducing a new “name,” and subsequent uses of the value in the execution trace refer to the new name. In this way, the execution order of the program is constrained only by essential data dependences; each instruction is executed as soon as its operands are available. Figure 1.7 summarizes the result of this ideal machine analysis based on data presented by Johnson (1991). The histogram on the left shows the fraction of cycles in which no instruction could issue, only one instruction could issue, and so on. Johnson’s ideal machine retains realistic function unit latencies, including cache

misses, which accounts for the zero-issue cycles. (Other studies ignore cache effects or ignore pipeline latencies and thereby obtain more optimistic estimates.) We see that, even with infinite machine resources, perfect branch prediction, and ideal renaming, no more than four instructions issue in a cycle 90% of the time. Based on this distribution, we can estimate the speedup obtained at various issue widths, as shown in the right portion of the figure. Recent work (Lam and Wilson 1992; Sohi, Breach, and Vijaykumar 1995) provides empirical evidence that to obtain significantly larger amounts of parallelism, multiple threads of control must be pursued simultaneously. Barring some unforeseen breakthrough in instruction-level parallelism, the leap to the next level of useful parallelism—multiple concurrent threads—is increasingly compelling as chips increase in capacity.

System Design Trends

The trend toward thread- or process-level parallelism has been strong at the computer system level for some time. Computers containing multiple state-of-the-art microprocessors sharing a common memory became prevalent in the mid-1980s, when the 32-bit microprocessor was first introduced (Bell 1985). As indicated by Figure 1.8, which shows the number of processors available in commercial multiprocessors over time, this bus-based shared memory multiprocessor approach has maintained a substantial multiplier to the increasing performance of the individual processors. Almost every commercial microprocessor introduced since the mid-1980s provides hardware support for multiprocessor configurations, as discussed in Chapter 5. Multiprocessors dominate the server and enterprise (or mainframe) markets and have migrated to the desktop.

The early multi-microprocessor systems were introduced by small companies competing for a share of the minicomputer market, including Synapse (Nestle and Inselberg 1985), Encore (Schanin 1986), Flex (Matelan 1985), Sequent (Rodgers 1985), and Myrias (Savage 1985). They combined 10 to 20 microprocessors to deliver competitive throughput on time-sharing loads. With the introduction of the 32-bit Intel i80386 as the base processor, these systems obtained substantial commercial success, especially in transaction processing. However, the rapid performance advance of RISC microprocessors, exploiting instruction-level parallelism, sapped the CISC multiprocessor momentum in the late 1980s and all but eliminated the minicomputer. Shortly thereafter, several large companies began producing RISC multiprocessor systems, especially as servers and mainframe replacements. These designs highlight the critical role of bandwidth. In most of these multiprocessor designs, all the processors plug into a common bus. Since a bus has a fixed bandwidth, as the processors become faster, a smaller number can be supported by the bus. The early 1990s brought a dramatic advance in the shared memory bus technology, including faster electrical signaling, wider datapaths, pipelined protocols, and multiple paths. Each of these provided greater bandwidth, growing with time and design experience, as indicated in Figure 1.9. This increase in bandwidth allowed the multiprocessor designs to ramp back up to the 10-to-20 range and beyond while tracking the microprocessor advances (Alexander et al. 1994; Cekleov et al. 1993;

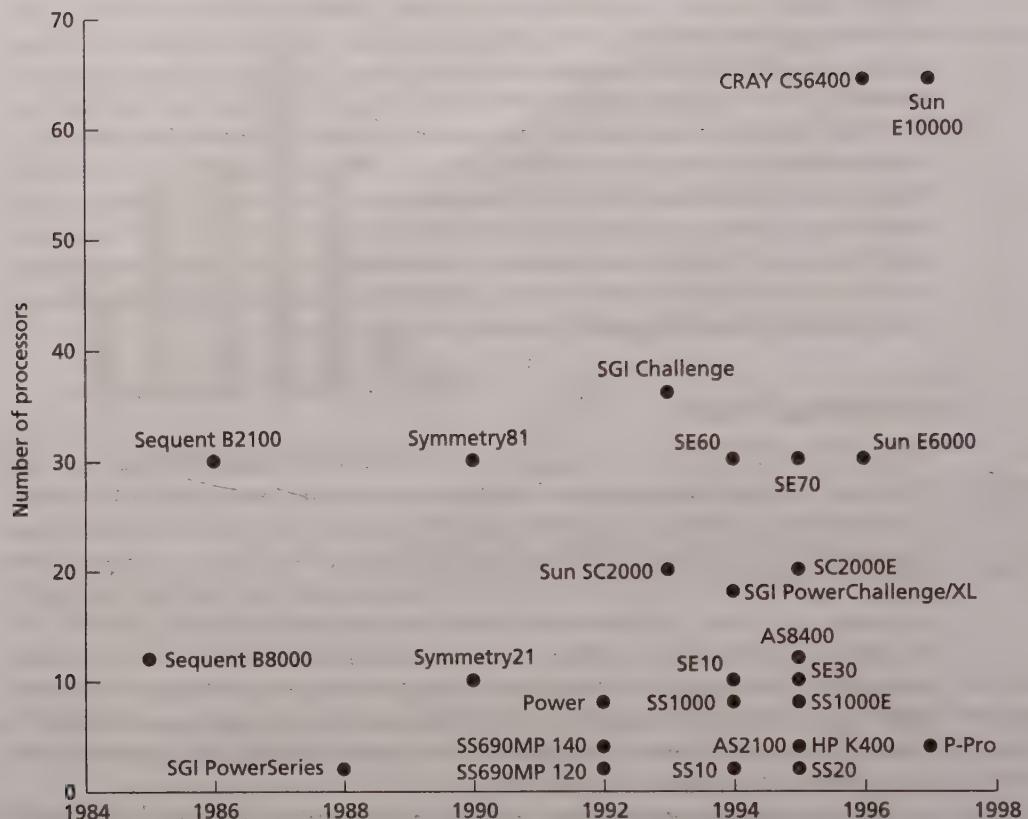


FIGURE 1.8 Number of processors in fully configured commercial bus-based shared memory multiprocessors. After an initial era of 10- to 20-way shared memory processors based on slow CISC microprocessors, companies such as Sun, HP, DEC, SGI, IBM, and CRI began producing sizable RISC-based SMPs, as did commercial vendors not shown here, including NCR/ATT, Tandem, and Pyramid.

Fenwick et al. 1995; Frank, Burkhardt, and Rothnie 1993; Galles and Williams 1993; Godiwala and Maskas 1995).

The picture in the mid-1990s is very interesting. Not only has the bus-based shared memory multiprocessor approach become ubiquitous in the industry, it is present at a wide range of scale. Desktop systems and small servers commonly support two to four processors, larger servers support tens, and large commercial systems are moving toward one hundred. Indications are that this trend will continue. As an illustration of the shift in emphasis, in 1994 Intel defined a standard approach to the design of multiprocessor PC systems around its Pentium microprocessor (Slater 1994). The follow-on Pentium Pro microprocessor allowed four-processor configurations to be constructed by wiring the chips together without even any glue logic; bus drivers, arbitration, and so on are in the microprocessor. This development is expected to make small-scale multiprocessors a true commodity. Addition-

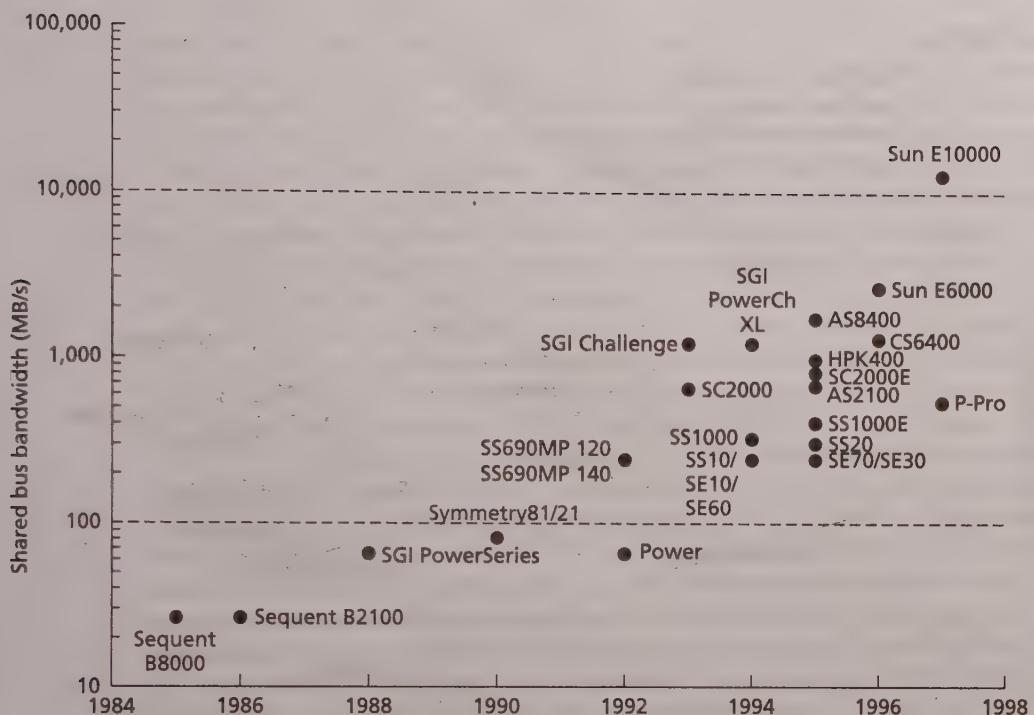


FIGURE 1.9 Bandwidth of the shared memory bus in commercial multiprocessors. After slow growth for several years, a new era of memory bus design began in 1991, which supported the use of substantial numbers of very fast microprocessors.

ally, a shift in the industry business model has been noted, where multiprocessors are being pushed by software vendors, especially database companies, rather than just by the hardware vendors. Combining these trends with the technology trends, it appears that the question is when, not if, multiple processors per chip will become prevalent.

1.1.4 Supercomputers

We have looked at the forces driving the development of parallel architecture in the general market. A second, confluent set of forces comes from the quest to achieve absolute maximum performance, known as *supercomputing*. Although commercial and information processing applications are increasingly becoming important drivers of the high end, scientific computing has historically been a kind of proving ground for innovative architecture. In the mid-1960s, this included pipelined instruction processing and dynamic instruction scheduling, which are commonplace in microprocessors today. Starting in the mid-1970s, supercomputing was dominated by *vector processors*, which perform operations on sequences of data

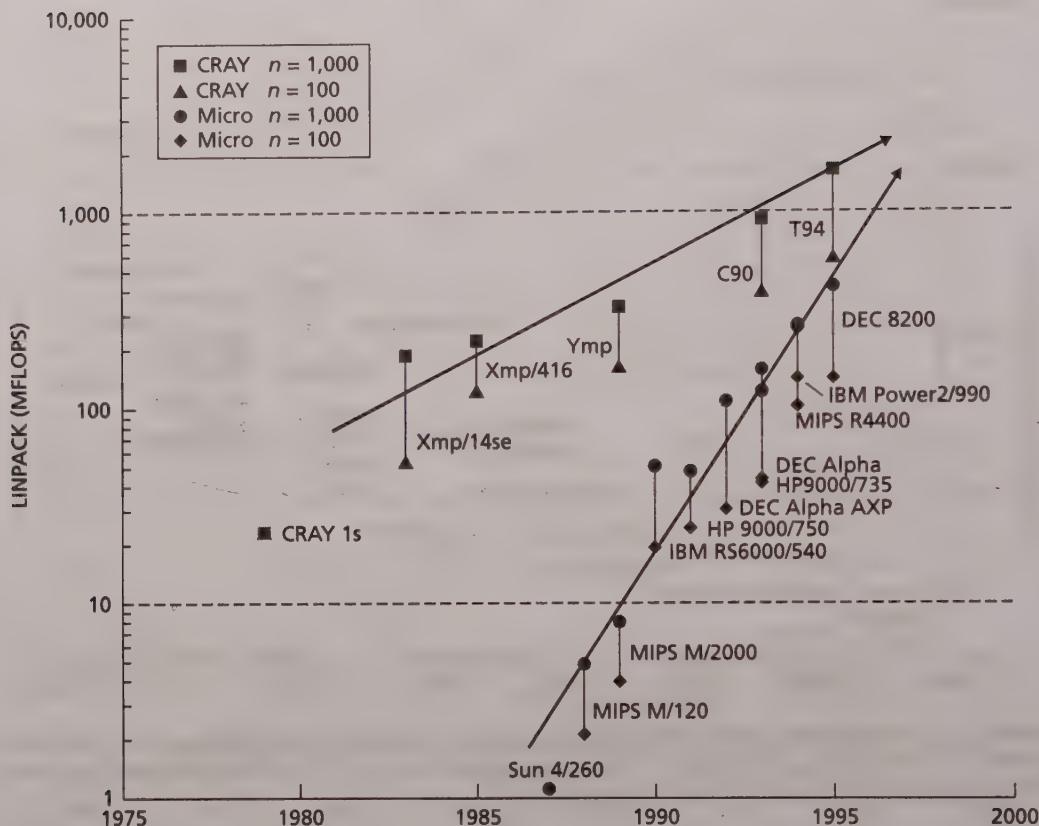


FIGURE 1.10 Uniprocessor performance of supercomputers and microprocessor-based systems on the LINPACK benchmark. Performance in MFLOPS for a single processor on solving dense linear equations is shown for the leading CRAY vector supercomputer and the fastest workstations on a 100×100 and $1,000 \times 1,000$ matrix.

elements; that is, a vector rather than individual scalar data. Vector operations permit more parallelism to be obtained within a single thread of control. In addition, these vector supercomputers were implemented in very fast, expensive, high-power circuit technologies.

Dense linear algebra is an important component of scientific computing and the specific emphasis of the LINPACK benchmark. Although this benchmark evaluates a narrow aspect of system performance, it is one of the few measurements available for a very wide class of machines over a long period of time. Figure 1.10 shows the LINPACK performance trend for one processor of the leading CRAY vector supercomputers (August et al. 1989; Russel 1978) compared with that of the fastest contemporary microprocessor-based workstations and servers. For each system two

data points are provided. The lower one is the performance obtained on a 100×100 matrix and the higher one on a $1,000 \times 1,000$ matrix. Within the vector processing approach, the single-processor performance improvement is dominated by modest improvements in cycle time and more substantial increases in the vector memory bandwidth. In the microprocessor systems, we see the combined effect of increasing clock rate, using on-chip pipelined floating-point units, increasing on-chip cache size, increasing off-chip second-level cache size, and increasing use of instruction-level parallelism. The gap in uniprocessor performance is rapidly closing.

Multiprocessor architectures are adopted by both the vector processor and microprocessor designs, but the scale is quite different. The CRAY Xmp first provided two and then four processors, the Ymp eight, the C90 sixteen, and the T94 thirty-two. The microprocessor-based supercomputers initially provided about 100 processors, increasing to roughly 1,000 from 1990 on. These aggregations of processors, known as *massively parallel processors* (MPPs), have tracked the microprocessor advance, with typically a lag of one to two years behind the leading microprocessor-based workstation or personal computer. As shown in Figure 1.11, the large number of slightly slower microprocessors has proved dominant for the LINPACK benchmark. (Note the change of scale from MFLOPS in Figure 1.10 to GFLOPS here.) The performance advantage of the MPP systems over traditional vector supercomputers is less substantial on more complete applications (Bailey et al. 1994) owing to the relative immaturity of the programming languages, compilers, and algorithms; however, the trend toward MPPs is still very pronounced. The importance of this trend was apparent enough in 1993 that CRAY Research announced its T3D, based on the DEC Alpha microprocessor.

Recently, the LINPACK benchmark has been used to rank the fastest computer systems in the world. Figure 1.12 shows the number of multiprocessor parallel vector processors (PVPs), MPPs, and bus-based shared memory multiprocessors (SMPs) appearing in the list of the top 500 systems. The latter two are both microprocessor based, and the trend is clear.

1.1.5 Summary

In examining current trends from a variety of perspectives—economics, technology, architecture, and application demand—we see that parallel architecture is increasingly attractive and increasingly central. The quest for performance is so keen that parallelism is being exploited at many different levels and at various points in the computer design space. Instruction-level parallelism is exploited in all modern high-performance processors. Essentially, all machines beyond the desktop are multiprocessors, including servers, mainframes, and supercomputers. The very high end of the performance curve is dominated by massively parallel processors. The use of large-scale parallelism in applications is broadening. The focus of this book is the multiprocessor level of parallelism. We study the design principles embodied in parallel machines from the modest scale to the very large, so that we may understand the spectrum of viable parallel architectures that can be built from well-proven components.

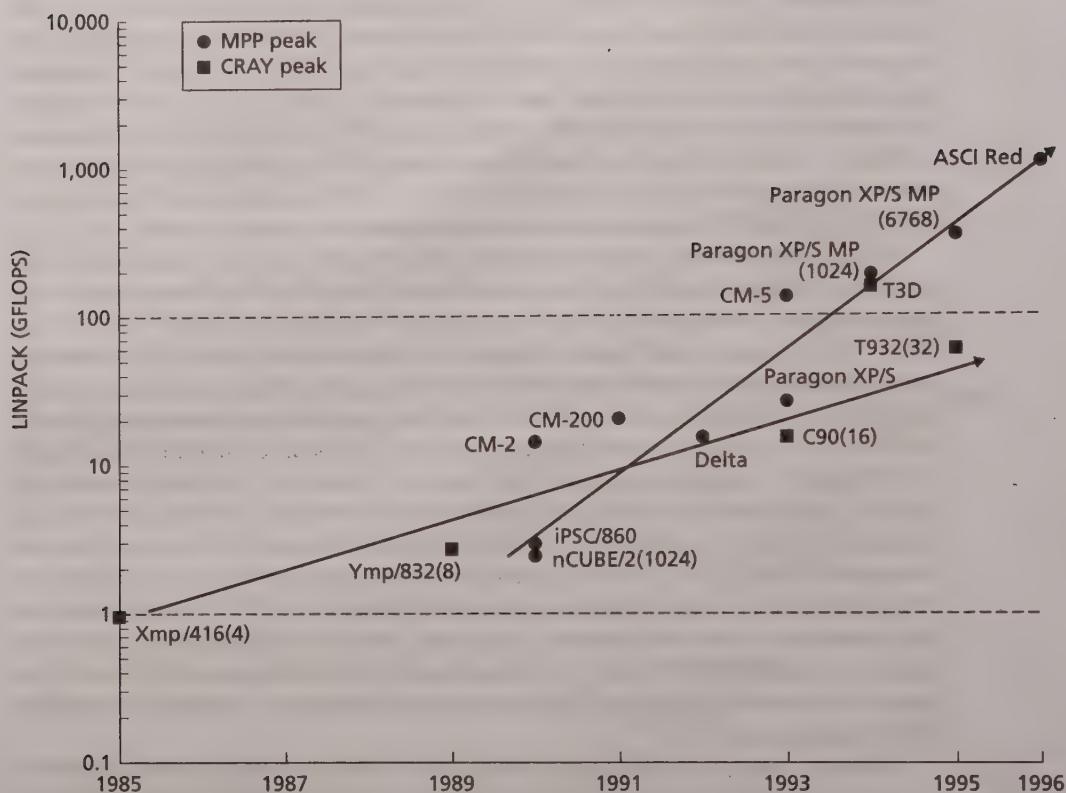


FIGURE 1.11 Performance of supercomputers and MPPs on the LINPACK peak performance benchmark. Peak performance in GFLOPS for solving dense linear equations is shown for the leading CRAY multiprocessor vector supercomputer and the fastest MPP systems. Note the change in scale from Figure 1.10 (MFLOPS to GFLOPS).

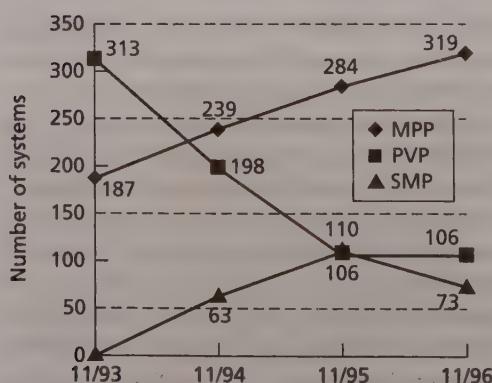


FIGURE 1.12 Types of systems used in the 500 fastest computer systems in the world. Parallel vector processors (PVPs) have given way to microprocessor-based massively parallel processors (MPPs) and bus-based symmetric shared memory multiprocessors (SMPs) at the high end of computing.

This discussion of the trends toward parallel computers has been primarily from the processor perspective, but you may arrive at the same conclusion from the memory system perspective. Consider briefly the design of a memory system to support a very large amount of data, that is, the data set of *large* problems. One of the few physical laws of computer architecture is that fast memories are small, large memories are slow. This occurrence is due to many factors, including the increased address decode time, the delays on increasingly long bit lines, the small drive of increasingly dense storage cells, and the selector delays. The result is that memory systems are constructed as a hierarchy of increasingly larger and slower memories: on average, a large hierarchical memory is fast, as long as the references exhibit good locality. The other trick we can play to cheat the laws of physics and obtain fast access on a very large data set is to use multiple processors and have the different processors access independent smaller memories. Of course, physics is not easily fooled. We pay the cost when a processor accesses nonlocal data, which we call *communication*, and when we need to orchestrate the actions of the many processors (i.e., in synchronization operations).

1.2 CONVERGENCE OF PARALLEL ARCHITECTURES

Historically, parallel machines have developed within several distinct architectural camps, and most texts on the subject are organized around a taxonomy of these designs. However, in looking at the evolution of parallel architecture, it is clear that the designs are strongly influenced by the same technological forces and similar application requirements. It is not surprising therefore that a great deal of convergence has occurred in the field. The goal of this section is to construct a framework for understanding the entire spectrum of parallel computer architectures and to build intuition as to the nature of the convergence. Along the way comes a quick overview of the evolution of parallel machines, starting from the traditional camps and moving toward the point of convergence.

1.2.1 Communication Architecture

Given that a parallel computer is “a collection of processing elements that communicate and cooperate to solve large problems fast” (Almasi and Gottlieb 1989), we may reasonably view parallel architecture as the extension of conventional computer architecture to address issues of communication and cooperation among processing elements. In essence, parallel architecture extends the usual concepts of a computer architecture with a communication architecture. Computer architecture has two distinct facets. One is the definition of critical abstractions, especially the hardware/software boundary and the user/system boundary. The architecture specifies the set of operations at the boundary and the data types that these operate on. The other facet is the organizational structure that realizes these abstractions to deliver high performance in a cost-effective manner. A *communication architecture* has these two

facets as well. It defines the basic communication and synchronization operations, and it addresses the organizational structures that realize these operations.

The framework for understanding communication in a parallel machine is illustrated in Figure 1.13. The top layer is the programming model, which is the conceptualization of the machine that the programmer uses in coding applications. Each programming model specifies how parts of the program running in parallel communicate information to one another and what synchronization operations are available to coordinate their activities. Applications are written in a programming model. In the simplest case, the model consists of multiprogramming a large number of independent sequential programs; no communication or cooperation takes place at the programming level. The more interesting cases include true parallel programming models, such as shared address space, message passing, and data parallel programming. We can describe these models intuitively as follows:

- *Shared address* programming is like using a bulletin board, where you can communicate with one or many colleagues by posting information at known, shared locations. Individual activities can be orchestrated by taking note of who is doing what task.
- *Message passing* is akin to telephone calls or letters, which convey information from a specific sender to a specific receiver. There is a well-defined event when the information is sent or received, and these events are the basis for orchestrating individual activities. However, no shared location is accessible to all.
- *Data parallel* processing is a more regimented form of cooperation, where several agents perform an action on separate elements of a data set simultaneously and then exchange information globally before continuing en masse. The global reorganization of data may be accomplished through accesses to shared addresses or messages since the programming model only defines the overall effect of the parallel steps.

A more precise definition of these programming models will be developed later in the text; at this stage, it is most important to understand the layers of abstraction.

A programming model is realized in terms of the user-level communication primitives of the system, referred to here as the *communication abstraction*. Typically, the programming model is embodied in a parallel language or programming environment, so a mapping exists from the generic language constructs to the specific primitives of the system. These user-level primitives may be provided directly by the hardware, by the operating system, or by machine-specific user software that maps the communication abstractions to the actual hardware primitives. The distance between the lines in Figure 1.13 is intended to indicate that the mapping from one layer to the next may be very simple or very involved. For example, access to a shared location is realized directly by load and store instructions on a machine in which all processors use the same physical memory; however, passing a message on such a machine may involve a library or system call to write the message into a buffer area or to read it out.

The communication architecture defines the set of communication operations available to the user software, the format of these operations, and the data types they

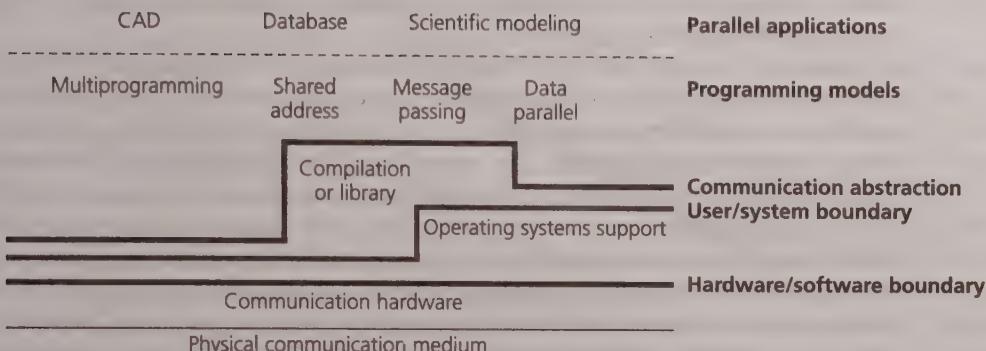


FIGURE 1.13 Layers of abstraction in parallel computer architecture. Critical layers of abstractions lie between the application program and the actual hardware. The application is written for a programming model, which dictates how pieces of the program share information and coordinate their activities. The specific operations providing communication and synchronization form the communication abstraction, which is the boundary between the user program and the system implementation. This abstraction is realized through compiler or library support using the primitives available from the hardware or from the operating system, which uses privileged hardware primitives. The communication hardware is organized to provide these operations efficiently on the physical wires connecting the machine together.

operate on, much as an instruction set architecture does for a processor. Note that even in conventional instruction sets, some operations may be realized by a combination of hardware and software, such as a load instruction that relies on operating system intervention in the case of a page fault. The communication architecture also extends the computer organization with the hardware structures that support communication.

As with conventional computer architecture, a great deal of debate has gone on over the years about what should be incorporated into each layer of abstraction in parallel architecture and how large the gap between the layers should be. This debate has been fueled by differing assumptions about the underlying technology and more qualitative assessments of “ease of programming.” The hardware/software boundary in Figure 1.13 is depicted as flat to indicate that the available hardware primitives in different designs is more or less of uniform complexity. Indeed, this is becoming more the case as the field matures. In most early designs, the physical hardware organization was strongly oriented toward a particular programming model; that is, the communication abstraction supported by the hardware was essentially identical to the programming model. This “high-level” parallel architecture approach resulted in tremendous diversity in the hardware organizations. However, as the programming models have become better understood and implementation techniques have matured, compilers and run-time libraries have grown to provide an important bridge between the programming model and the underlying hardware. Simultaneously, the technological trends discussed in Section 1.1.2 have exerted a strong influence, regardless of the programming model. The result has

been a convergence in the organizational structure with relatively simple, general-purpose communication primitives.

Sections 1.2.2–1.2.6 survey the most widely used programming models and the corresponding styles of machine design in past and current parallel machines. With the historical orientation to a particular programming model, it was common to lump the programming model, the communication abstraction, and the machine organization together as “the architecture,” for example, shared memory architecture, message-passing architecture, and so on. This approach is less appropriate today since a large commonality exists across parallel machines and since many machines support several programming models. It is important to see how this convergence has come about, so these sections begin from the traditional perspective, looking at machine designs associated with particular programming models and explaining their intended roles and the technological opportunities that influenced their design. The goal of the survey is not to develop a taxonomy of parallel machines per se but to identify a set of core concepts that form the basis for assessing design trade-offs across the entire spectrum of potential designs today and in the future. It also demonstrates the influence that the dominant technological direction established by microprocessor and DRAM technologies has had on parallel machine design, which makes a common treatment of the fundamental design issues natural or even imperative. Specifically, shared address, message-passing, data parallel, data-flow, and systolic approaches are presented. In each case, the abstraction embodied in the programming model is explained, and the reasons for the particular style of design, as well as the intended scale and application, are presented. The technological motivations for the approach are also examined, as well as how they have changed over time. These changes are reflected in the machine organization, which determines what is fast and what is slow. The performance characteristics ripple up to influence aspects of the programming model. The outcome of this brief survey is a clear organizational convergence, which is captured in a generic parallel machine in Section 1.2.7.

1.2.2 Shared Address Space

One of the most important classes of parallel machines is *shared memory multiprocessors*. The key property of this class is that communication occurs implicitly as a result of conventional memory access instructions (i.e., loads and stores). This class has a long history, dating at least to precursors of mainframes in the early 1960s,³ and today it has a role in almost every segment of the computer industry. Shared memory multiprocessors serve to provide better throughput on multiprogramming workloads, as well as to support parallel programs. Thus, they are naturally found across a wide range of scale, from a few processors to perhaps hundreds. This sec-

3. Some say that BINAC was the first multiprocessor, but it was intended to improve reliability. The two processors checked each other at every instruction. They seldom agreed, so people eventually turned one of them off.

tion examines the communication architecture of shared memory machines and the key organizational issues for small-scale designs and large configurations.

The primary programming model for these machines is essentially that of time-sharing on a single processor, except that real parallelism replaces interleaving in time. Formally, a *process* is a virtual address space and one or more threads of control. Processes can be configured so that portions of their address space are shared, that is, are mapped to a common physical location, as suggested by Figure 1.14. (Multiple threads within a process, by definition, share portions of the address space.) Cooperation and coordination among threads is accomplished by reading and writing shared variables and pointers referring to shared addresses. Writes to a logically shared address by one thread are visible to reads of the other threads. The communication architecture employs the conventional memory operations to provide communication through shared addresses as well as special atomic operations for synchronization. Even completely independent processes typically share the kernel portion of the address space, although this is only accessed by operating system code. Nonetheless, the shared address space model is utilized within the operating system to coordinate the execution of the processes.

Although shared memory can be used for communication among arbitrary collections of processes, most parallel programs are quite structured in their use of the virtual address space. They typically have a common code image, private segments for the stack and other private data, and shared segments that are in the same region of the virtual address space of each process or thread of the program. This simple structure implies that the private variables in the program are present in each process and that shared variables have the same address and meaning in each thread. Often, straightforward parallelization strategies are employed. For example, each process may perform a subset of the iterations of a common parallel loop or, more generally, processes may operate as a pool of workers obtaining work from a shared queue. Chapter 2 discusses the structure of parallel programs more deeply. Here we look at the basic evolution and development of this important architectural approach.

The communication hardware for shared memory multiprocessors is a natural extension of the memory system found in most computers. Essentially all computer systems allow a processor and a set of I/O controllers to access a collection of memory modules through some kind of hardware interconnect, as illustrated in Figure 1.15. The memory capacity is increased simply by adding memory modules. Additional capacity may or may not increase the available memory bandwidth, depending on the specific system organization. I/O capacity is increased by adding devices to I/O controllers or by inserting additional I/O controllers. There are two possible ways to increase the processing capacity: wait for a faster processor to become available or add more processors. On a time-sharing workload, increasing processing capacity should increase the throughput of the system. With more processors, more processes can run at once and throughput is increased. If a single application is programmed to make use of multiple threads, more processors should speed up the application. The hardware primitives are essentially one to one with the communication abstraction, and these operations are available in the programming model.

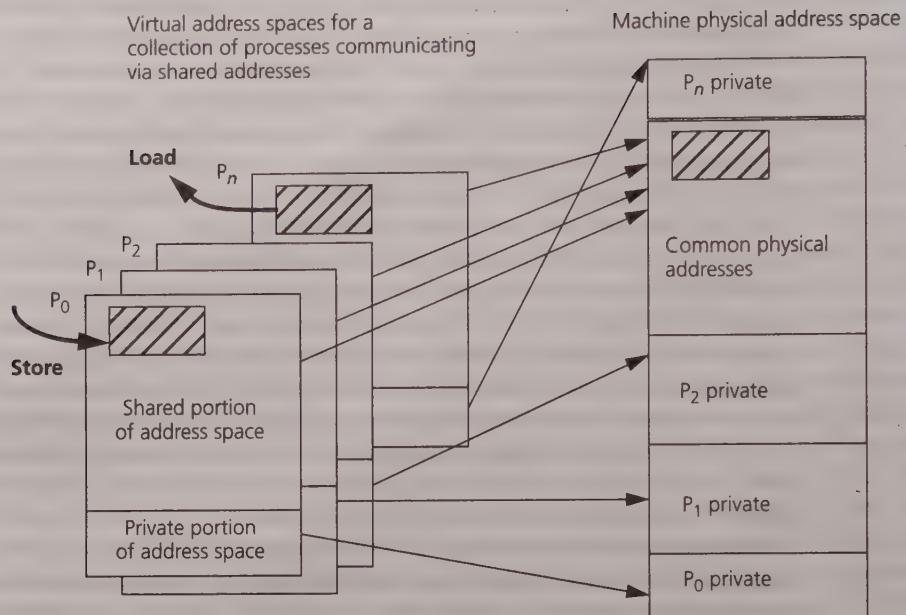


FIGURE 1.14 Typical memory model for shared memory parallel programs. Collections of processes have a common region of physical addresses mapped into their virtual address space, in addition to the private region, which typically contains the stack and private data.

Within the general framework of Figure 1.15, a great deal of evolution of shared memory machines has taken place as the underlying technology has advanced. The early machines were “high-end” mainframe configurations (Lonergan and King 1961; Padegs 1981). On the technology side, memory in early mainframes was slow compared to the processor, so it was necessary to interleave data across several memory banks to obtain adequate bandwidth for even a single processor; this required an interconnect between the processor and each of the banks. On the application side, these systems were primarily designed for throughput on a large number of jobs. Thus, to meet the I/O demands of a workload, several I/O channels and devices were attached. The I/O channels also required direct access to each of the memory banks. Therefore, these systems were typically organized with a *crossbar switch* connecting the CPU and several I/O channels to several memory banks, as indicated by Figure 1.16a. Adding processors was primarily a matter of expanding the switch; the hardware structure to access a memory location from a port on the processor and I/O side of the switch was unchanged. The size and cost of the processor limited these early systems to a small number of processors, but as the hardware density and cost improved, larger systems could be contemplated. The cost of scaling the crossbar became the limiting factor, and in many cases it was replaced by a *multistage interconnect*, suggested by Figure 1.16b, for which the cost increases more slowly with

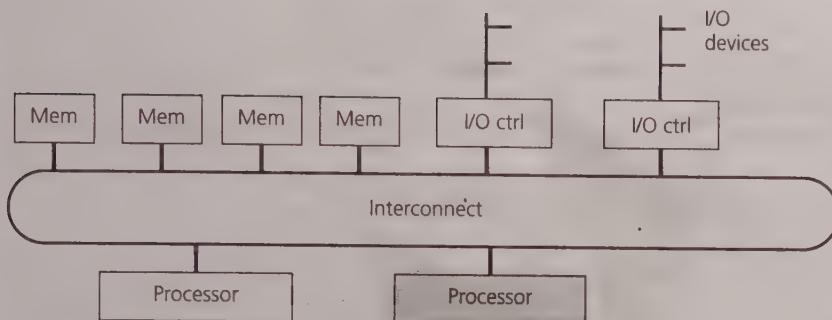


FIGURE 1.15 Extending a system into a shared memory multiprocessor by adding processor modules. Most systems consist of one or more memory modules accessible by a processor and I/O controllers through a hardware interconnect, typically a bus, crossbar, or multistage interconnect. Memory and I/O capacity are increased by attaching memory and I/O modules. Shared memory machines allow processing capacity to be increased by adding processor modules (shown as shaded).

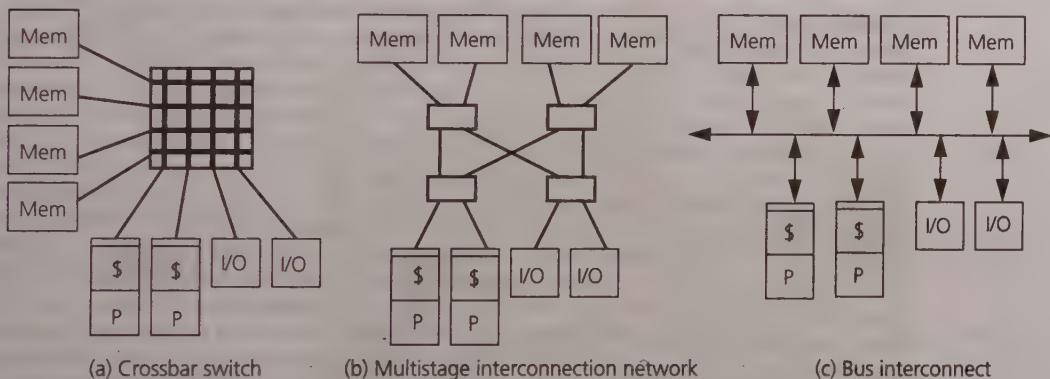


FIGURE 1.16 Typical shared memory multiprocessor interconnection schemes. The interconnection of multiple processors, with their local caches (indicated by \$), and I/O controllers to multiple memory modules may be via crossbar, multistage interconnection network, or bus.

the number of ports. These savings come at the expense of increased latency and decreased bandwidth per port if all are used at once. The ability to access all memory directly from each processor has several advantages: any processor can run any process or handle any I/O event, and data structures can be shared within the operating system.

The widespread use of shared memory multiprocessor designs came about with the 32-bit microprocessor revolution in the mid-1980s because the processor, cache, floating-point, and memory management unit fit on a single board (Bell 1985) or even two to a board. Most mid-range machines, including minicomputers, servers, workstations, and personal computers, are organized around a central memory bus, as illustrated in Figure 1.16c, and the bus could be adapted to support multiple

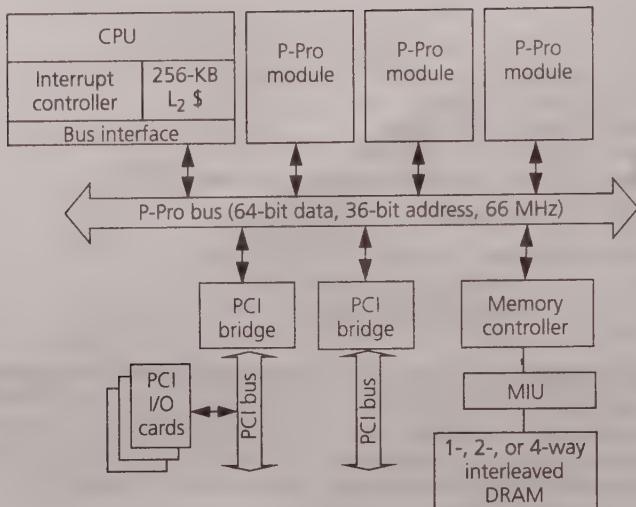


FIGURE 1.17(a) Physical and logical organization of the Intel Pentium Pro four-processor "quad pack." The Intel quad-processor Pentium Pro motherboard employed in many multiprocessor servers illustrates the major design elements of most small-scale shared memory multiprocessors. Its logical block diagram (a) shows that it can accommodate up to four processor modules, each containing a Pentium Pro processor, first-level caches, translation lookaside buffer, a 256-KB second-level cache, an interrupt controller, and a bus interface in a single chip connecting directly to a 64-bit memory bus. The bus operates at 66 MHz, and memory transactions are pipelined to give a peak bandwidth of 528 MB/s. A two-chip memory controller and four-chip memory interleave unit (MIU) connect the bus to multiple banks of DRAM. Bridges connect the memory bus to two independent PCI buses, which host display, network, SCSI, and lower-speed I/O connections. The Pentium Pro module contains all the logic necessary to support the multiprocessor communication architecture, including that required for memory and cache consistency. The structure of the Pentium Pro "quad pack" is similar to a large number of earlier SMP designs but has a much higher degree of integration and is targeted at a much larger volume. (b) shows an expanded view of a typical Pentium Pro SMP, an HP NetServer in the LX series. Source: Reproduced with permission of Hewlett-Packard Company.

processors. The standard bus access mechanism allows any processor to access any physical address in the system. Like the switch-based designs, all memory locations are equidistant to all processors, so all processors experience the same access time, or latency, on a memory reference. This configuration is usually called a *symmetric multiprocessor* (SMP).⁴ SMPs are heavily used for execution of parallel programs as well as multiprogramming. The typical organization of the bus-based symmetric multiprocessor is illustrated in more detail by Figure 1.17, which describes the first

4. The term SMP is widely used but causes a bit of confusion. What exactly needs to be symmetric? Many designs are symmetric in some respect. The more precise description of what is intended by SMP is a shared memory multiprocessor where the cost of accessing a memory location is the same for all processors; that is, it has uniform access costs when the access actually is to memory. If the location is cached, the access will be faster, but cache access times and memory access times are the same on all processors.

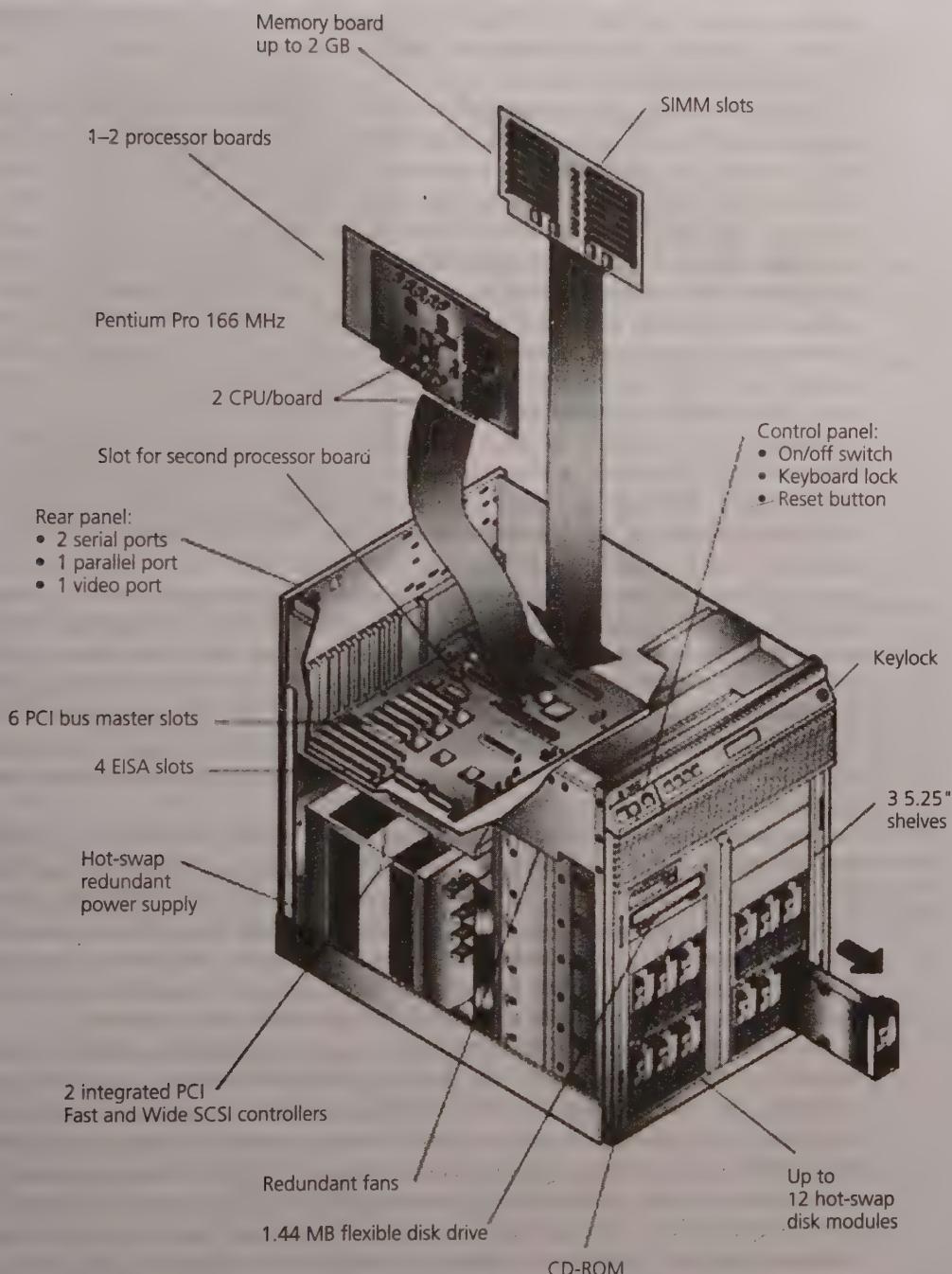


FIGURE 1.17(b) Physical organization of the Intel Pentium Pro four-processor "quad pack"

highly integrated SMP for the commodity market. Figure 1.18 illustrates a high-end server organization that distributes the physical memory over the processor modules, but retains symmetric access.

The factors limiting the number of processors that can be supported with a bus-based organization are quite different from those in the switch-based approach. Adding processors to the switch is expensive; however, the aggregate bandwidth increases with the number of ports. The cost of adding a processor to the bus is small, but the aggregate bandwidth is fixed. Dividing this fixed bandwidth among the larger number of processors limits the practical scalability of the approach. (It is this critical bus bandwidth that is depicted in Figure 1.9.) Fortunately, caches reduce the bandwidth demand of each processor since many references are satisfied by the cache rather than by the memory. However, with data replicated in local caches, there is the potentially challenging problem of keeping the caches “consistent,” which will be examined in detail in Chapters 5, 6, and 8.

Starting from a baseline of small-scale shared memory machines, illustrated in Figures 1.16–1.18, we may ask what is required to scale the design to a large number of processors. The basic processor component is well suited to the task since it is small and economical, but a problem clearly exists with the interconnect. The bus does not scale because it has a fixed aggregate bandwidth. The crossbar does not scale well because the cost increases as the square of the number of ports. Many alternative scalable interconnection networks exist, such that the aggregate bandwidth increases as more processors are added, but the cost does not become excessive. We need to be careful about the resulting increase in latency because the processor may stall while a memory operation moves from the processor to the memory module and back. If the latency of access becomes too large, the processors will spend much of their time waiting, and the advantages of more processors may be offset by poor utilization.

One natural approach to building scalable shared memory machines is to maintain the uniform memory access (or “dancehall”) approach of Figure 1.15 and provide a scalable interconnect between the processors and the memories. Every memory access is translated into a message transaction over the network, much as it might be translated to a bus transaction in the SMP designs. The primary disadvantage of this approach is that the round-trip network latency is experienced on every memory access and a large bandwidth must be supplied to every processor.

An alternative approach is to interconnect complete processors, each with a local memory, as illustrated in Figure 1.19. In this nonuniform memory access (NUMA) approach, the local memory controller determines whether to perform a local memory access or a message transaction with a remote memory controller. Accessing local memory is faster than accessing remote memory. (The I/O system may either be a part of every node or consolidated into special I/O nodes, not shown.) Accesses to private data, such as code and stack, can often be performed locally, as can accesses to shared data that, by accident or intent, are stored on the local node. The ability to access the local memory quickly does not increase the time to access remote data

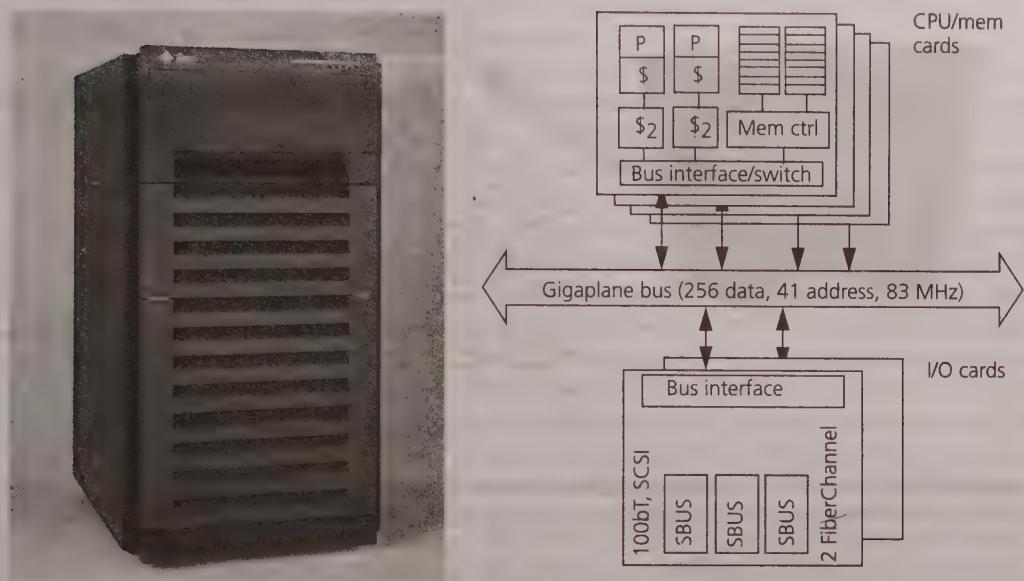


FIGURE 1.18 Physical and logical organization of the Sun Enterprise Server. A larger-scale design is illustrated by the Sun UltraSparc-based Enterprise multiprocessor server. The diagram shows its physical structure and logical organization. A wide (256-bit), highly pipelined memory bus delivers 2.5 GB/s of memory bandwidth. This design uses a hierarchical structure, where each card is either a complete dual processor with memory or a complete I/O system. The full configuration supports 16 cards of either type, with at least one of each. The CPU/mem card contains two UltraSparc processor modules, each with 16-KB level 1 and 512-KB level 2 caches, plus two 512-bit-wide memory banks and an internal switch. Thus, adding processors adds memory capacity and memory interleaving. The I/O card provides three SBUS slots for I/O extensions, a SCSI connector, a 100bT Ethernet port, and two FiberChannel interfaces. A typical complete configuration would be 24 processors and 6 I/O cards. Although memory banks are physically packaged with pairs of processors, all memory is equidistant from all processors and accessed over the common bus, preserving the SMP characteristics. Data may be placed anywhere in the machine with no performance impact. Source: The copyright for this photograph is owned by Sun Microsystems, Inc. and is used herein by permission.

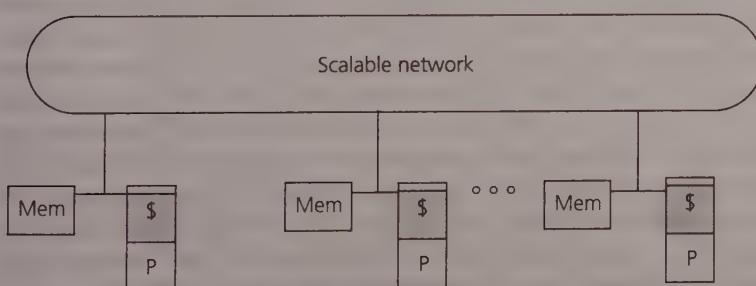


FIGURE 1.19 Nonuniform memory access (NUMA) scalable shared memory multiprocessor organization. Processor and memory modules are closely integrated such that access to local memory is faster than access to remote memories.

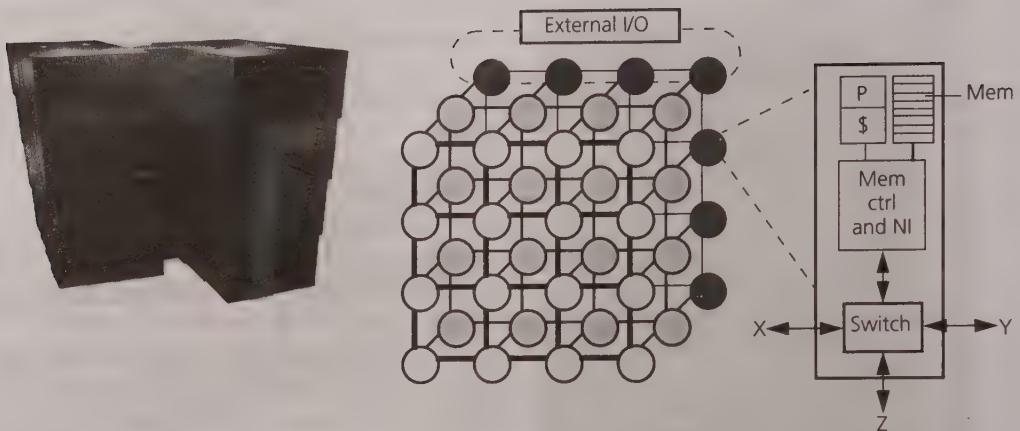


FIGURE 1.20 CRAY T3E scalable shared address space machine. The CRAY T3E is designed to scale up to a thousand processors supporting a global shared address space. Each node contains a DEC Alpha processor, local memory, a network interface integrated with the memory controller, and a network switch. The machine is organized as a three-dimensional cube, with each node connected to its six neighbors through 650-MB/s point-to-point links. Any processor can read or write any memory location; however, the NUMA characteristic of the machine is exposed in the communication architecture as well as in its performance characteristics. A short sequence of instructions is required to establish addressability to remote memory, which can then be accessed by conventional loads and stores. The memory controller captures the access to a remote memory and conducts a message transaction with the memory controller of the remote node on the local processor's behalf. The message transaction is automatically routed through intermediate nodes to the desired destination, with a small delay per "hop." The remote data is not cached since there is no hardware mechanism to keep it consistent. (We will look at other design points that allow shared data to be replicated throughout the processor caches.) The CRAY T3E I/O system is distributed over a collection of nodes on the surface of the cube, which are connected to the external world through an additional I/O network. Source: Photo courtesy of CRAY Research.

appreciably, so it reduces the average access time, especially when a large fraction of the accesses are to local data. The bandwidth demand placed on the network is also reduced. Although some conceptual simplicity arises from having all shared data equidistant from any processor, the NUMA approach has become far more prevalent at a large scale because of its inherent performance advantages and because it harnesses more of the mainstream processor memory system technology. One example of this style of design is the CRAY T3E, illustrated in Figure 1.20. This machine reflects the viewpoint that, although all memory is accessible to every processor, the distribution of memory across processors is exposed to the programmer. Caches are used only to hold data (and instructions) from local memory. It is the programmer's job to avoid frequent remote references. The SGI Origin is an example of a machine with a similar organizational structure, but it allows data from any memory to be replicated into any of the caches and provides hardware support to keep the caches consistent without relying on a bus connecting all the modules with a common set

of wires. While this book was being written, these two designs literally converged following the merger of the two companies.

To summarize, communication and cooperation in the shared address space programming model consists of reads and writes to shared variables; these operations are mapped directly to a communication abstraction consisting of load and store instructions accessing a global, shared address space, which is supported directly in hardware through access to shared physical memory locations. The programming model and communication abstraction are very close to the actual hardware. Each processor can name every physical location in the machine; a process can name all data it shares with others within its virtual address space. Data is transferred either as primitive types in the instruction set (bytes, words, etc.) or as cache blocks. Each process performs memory operations on addresses in its virtual address space; the address translation process identifies a physical location, which may be local or remote to the processor and may be shared with other processes. In either case, the hardware accesses it directly, without user or operating system software intervention. The address translation realizes protection within the shared address space, just as it does for uniprocessors, since a process can only access the data in its virtual address space.

The effectiveness of the shared memory approach depends on the latency incurred on memory accesses as well as the bandwidth of data transfer that can be supported. Just as a memory storage hierarchy allows data that is bound to an address to be migrated toward the processor, expressing communication in terms of the storage address space allows shared data to be migrated toward the processor that accesses it. However, migrating and replicating data across a general-purpose interconnect presents a unique set of challenges. We will see that to achieve scalability in such a design, the entire solution, including the hardware interconnect mechanisms used for maintaining the consistent shared memory abstractions, must scale well.

1.2.3 Message Passing

A second important class of parallel machines, called *message-passing architectures*, employs complete computers as building blocks—including the microprocessor, memory, and I/O system—and provides communication between processors as explicit I/O operations. The high-level block diagram for a message-passing machine is essentially the same as the NUMA shared memory approach shown in Figure 1.19. The primary difference is that communication is integrated at the I/O level rather than into the memory system. This style of design also has much in common with networks of workstations, or *clusters*, except that the packaging of the nodes is typically much tighter, there is no monitor or keyboard per node, and the network is of much higher capability than a standard local area network. The integration between the processor and the network tends to be much tighter than in traditional I/O structures, which support connection to devices that are much slower than the processor, since message passing is fundamentally processor-to-processor communication.

In message passing, a substantial distance exists between the programming model and the actual hardware primitives, with user communication performed through operating system or library calls that perform many lower-level actions, including the actual communication operation. Thus, our discussion of message passing begins with a look at the communication abstraction and then briefly surveys the evolution of hardware organizations supporting this abstraction.

The most common user-level communication operations on message-passing systems are variants of send and receive. In its simplest form, *send* specifies a local data buffer that is to be transmitted and a receiving process (typically on a remote processor). *Receive* specifies a sending process and a local data buffer into which the transmitted data is to be placed. Together, the matching send and receive cause a data transfer from one process to another, as indicated in Figure 1.21. In most message-passing systems, the send operation also allows an identifier or *tag* to be attached to the message, and the receiving operation specifies a matching rule (such as a specific tag from a specific processor, or any tag from any processor). Thus, the user program names local addresses and entries in an abstract process-tag space. *The combination of a send and a matching receive accomplishes a pairwise synchronization event and a memory-to-memory copy, where each end specifies its local data address.* There are several possible variants of this synchronization event, depending upon whether the send completes when the receive has been executed, when the send buffer is available for reuse, or when the request has been accepted. Similarly, the receive can potentially wait until a matching send occurs or simply post the receive. Each of these variants has somewhat different semantics and different implementation requirements.

Message passing has long been used as a means of communication and synchronization among arbitrary collections of cooperating sequential processes, even on a single processor. Important examples include programming languages, such as CSP and Occam, and common operating systems functions, such as sockets. Parallel programs using message passing are typically quite structured. Most often, all nodes execute identical copies of a program, with the same code and private variables. Usually, processes can name each other using a simple linear ordering of the processes comprising a program.

Early message-passing machines provided hardware primitives that were very close to the simple send/receive user-level communication abstraction, with some additional restrictions. A node was connected to a fixed set of neighbors in a regular pattern by point-to-point links that behaved as simple FIFOs (Seitz 1985). This sort of design is illustrated in Figure 1.22 for a small 3D cube. Many early machines were *hypercubes*, where each node is connected to n other nodes differing by one bit in the binary address, for a total of 2^n nodes. Others were *meshes*, where the nodes are connected to neighbors on two or three dimensions. The network topology was especially important in the early message-passing machines because only the neighboring processors could be named in a send or receive operation. The data transfer involved the sender writing to a link and the receiver reading from the link. The FIFOs were small and so the sender would not be able to finish writing the mes-

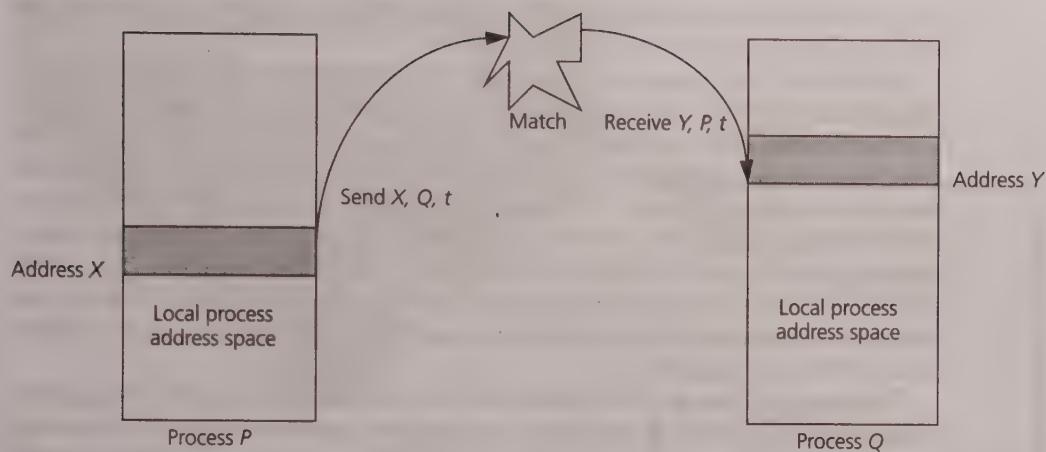


FIGURE 1.21 User-level send/receive message-passing abstraction. A data transfer from one local address space to another occurs when a send to a particular process is matched with a receive posted by that process.

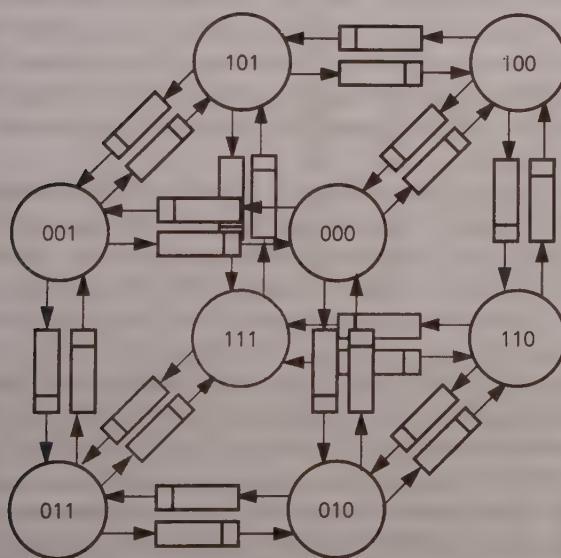


FIGURE 1.22 Typical structure of an early message-passing machine. Each node is connected to neighbors in three dimensions via FIFOs.

sage until the receiver started reading it, so the send would block until the receive occurred. (In modern terms, this is called *synchronous message passing* because the two events coincide in time.) The details of moving data were hidden from the

programmer in a message-passing library, forming a layer of software between send and receive calls and the actual hardware.⁵

The direct FIFO design was soon replaced by more versatile and more robust designs that provided *direct memory access* (DMA) transfers on either end of the communication event. A DMA device is a special-purpose controller that transfers data between memory and an I/O device without engaging the processor until the transfer is complete. The use of DMA allowed *nonblocking sends*, where the sender is able to initiate a send and continue with useful computation (or even perform a receive) while the send completes. On the receiving end, the transfer is accepted via a DMA transfer by the message layer into a buffer and queued until the target process performs a matching receive, at which point the data is copied into the address space of the receiving process.

The physical topology of the communication network so dominated the programming model of these early machines that parallel algorithms were often stated in terms of a specific interconnection topology, for example, a ring, a grid, or a hypercube (Fox et al. 1988). However, to make the machines more generally useful, the designers of the message layers provided support for communication between arbitrary processors rather than only between physical neighbors. This was originally supported by forwarding the data within the message layer along links in the network. Soon this routing function was moved into the hardware (as discussed in Chapter 10), so each node consisted of a processor with memory and a switch that could forward messages. However, in this approach, known as *store-and-forward*, the time to transfer a message is proportional to the number of hops it takes through the network, so an emphasis remained on interconnection topology. (See Exercise 1.7 for a brief store-and-forward example.)

The emphasis on network topology was significantly reduced with the introduction of more general-purpose networks, which pipelined the message transfer through each of the routers forming the interconnection network (Barton, Crownie, and McLaren 1994; Bomans and Roose 1989; Dunigan 1988; Homewood and McLaren 1993; Leiserson et al. 1996; Pierce and Regnier 1994; von Eicken et al. 1992). In most modern message-passing machines, the incremental delay introduced by each hop is small enough that the transfer time is dominated by the time to simply move that data between the processor and the network, not how far it travels (Groscup 1992; Homewood and McLaren 1993; Horiw et al. 1993; Pierce and Regnier 1994). This greatly simplifies the programming model; typically, the processors are viewed as simply forming a linear sequence with uniform communication costs. In other words, the communication abstraction reflects an organizational structure much as in Figure 1.19. One important example of such a machine is the IBM SP-2, illustrated in Figure 1.23, which is constructed from RS6000 workstation nodes, a scalable network, and a network interface containing a dedicated processor. Another

5. The motivation for synchronous message passing was not just from the machine structure; it was also present in important programming languages, especially CSP (Hoare 1978), because of its clean theoretical properties. Early in the microprocessor era, the approach was captured in a single-chip building block, the Transputer, which was widely touted during its development by INMOS as a revolution in computing.

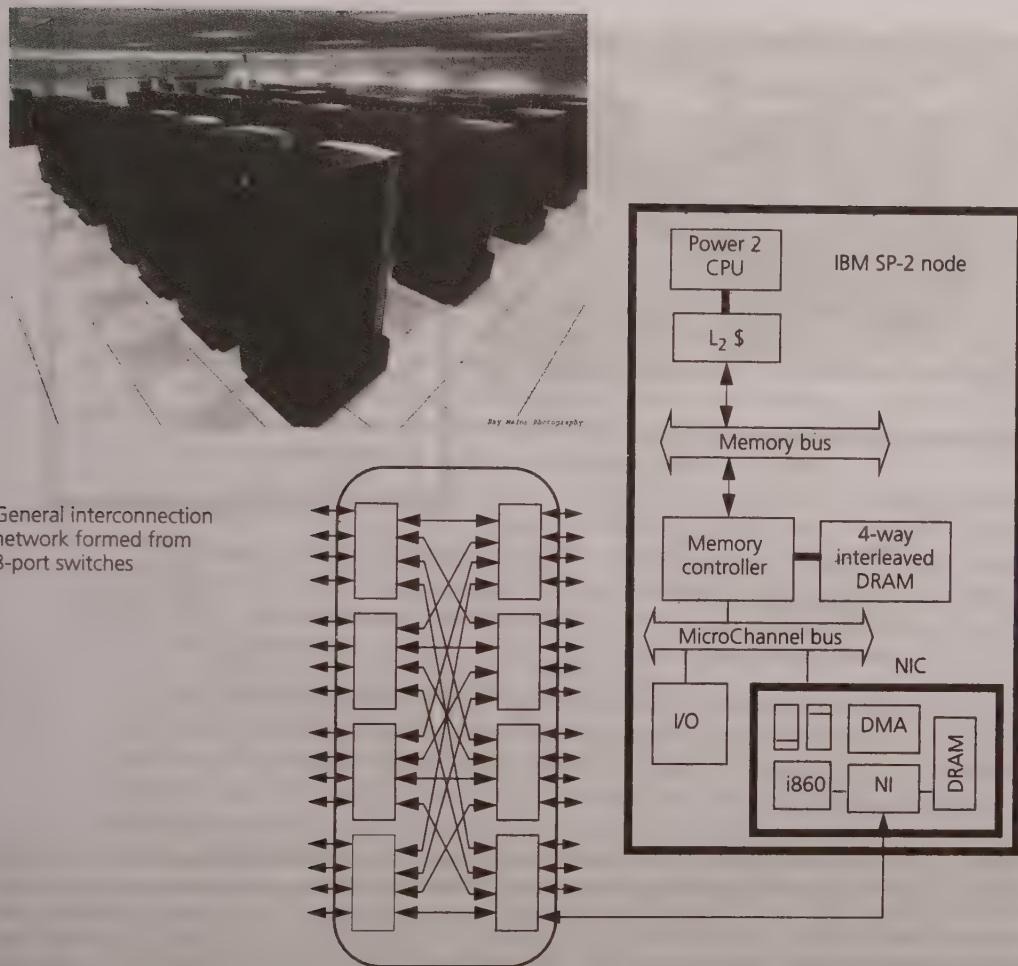


FIGURE 1.23 IBM SP-2 message-passing machine. The IBM SP-2 is a scalable parallel machine constructed essentially out of complete RS6000 workstations. Modest modifications are made to package the workstations into standing racks. A network interface card (NIC) is inserted at the MicroChannel I/O bus. The NIC contains the drivers for the actual link into the network, a substantial amount of memory to buffer message data, a direct memory access (DMA) engine, and a complete i860 microprocessor to move data between host memory and the network. The network itself is a butterfly-like structure, constructed by cascading 8×8 crossbar switches. The links operate at 40 MB/s in each direction, which is the full capability of the I/O bus. Several other machines employ a similar network interface design but connect directly to the memory bus rather than at the I/O bus. Source: Ray Mains Photography.

is the Intel Paragon, illustrated in Figure 1.24, which integrates the network interface more tightly to the processors in SMP nodes, where one of the processors is dedicated to supporting message passing.

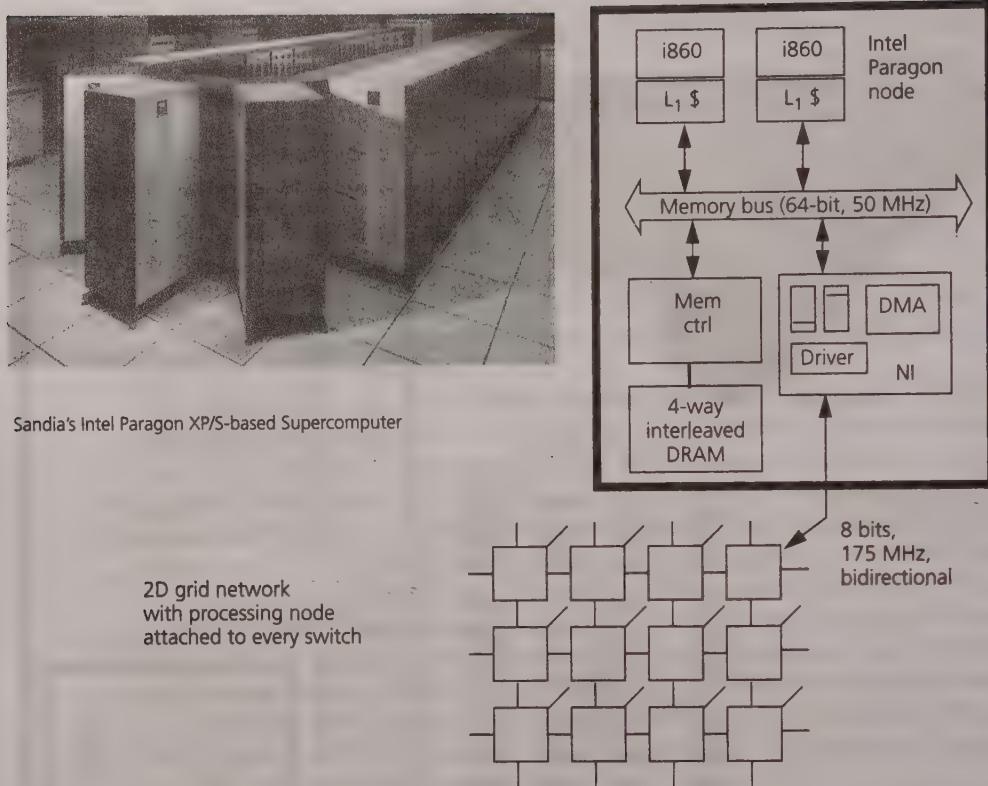


FIGURE 1.24 Intel Paragon. The Intel Paragon illustrates a much tighter packaging of nodes. Each card is an SMP with two or more i860 processors and a network interface chip connected to the cache-coherent memory bus. One of the processors is dedicated to servicing the network. In addition, the node has a DMA engine to transfer contiguous chunks of data to and from the network at a high rate. The network is a 3D grid, much like the CRAY T3E, with links operating at 175 MB/s in each direction. Source: Photo courtesy of Intel Corporation.

A processor in a message-passing machine can name only the locations in its local memory and each of the processors, perhaps by number or by route. A user process can only name private addresses and other processes; it can transfer data using the send/receive calls.

1.2.4 Convergence

Evolution of the hardware and software has blurred the once clear boundary between the shared memory and message-passing camps. First, consider the communication operations available to the user process.

- Traditional message-passing operations (send/receive) are supported on most shared memory machines through shared buffer storage. Send involves writing data, or a pointer to data, into the buffer; receive involves reading the data from shared storage. Flags or locks are used to control access to the buffer and to indicate events such as message arrival.
- On a message-passing machine, a user process may construct a global address space of sorts by carrying along pointers specifying the process and local virtual address in that process. Access to such a global address can be performed in software through an explicit message transaction. Most message-passing libraries allow a process to accept a message for any process, so each process can serve data requests from the others. A logical read is realized by sending a request to the process containing the object and receiving a response. The actual message transaction may be hidden from the user; it may be carried out by compiler-generated code for access to a shared variable.
- A shared virtual address space can be established on a message-passing machine at the page level. A collection of processes has a region of shared addresses but, for each process, only the pages that are local to it are accessible. Upon access to a missing (i.e., remote) page, a page fault occurs and the operating system engages the remote node in a message transaction to transfer the page and map it into the user address space.

At the level of machine organization, substantial convergence has occurred as well. Modern message-passing architectures appear essentially identical at the block diagram level to the scalable NUMA design illustrated in Figure 1.19. In the shared memory case, the network interface was integrated with the cache controller or memory controller in order for that device to observe cache misses and to conduct a message transaction to access memory in a remote node. In the message-passing approach, the network interface is essentially an I/O device. However, the trend has been to integrate this device more deeply into the memory system as well and to transfer data directly from and to the user address space. Some designs provide DMA transfers across the network, from memory on one machine to memory on the other machine, so the network interface is integrated fairly deeply with the memory system. Message passing is implemented on top of these remote memory copies (Bartron, Crownie, and McLaren 1994). In some designs, a complete processor assists in communication, sharing a cache-coherent memory bus with the main processor (Groscup 1992; Pierce and Regnier 1994). Viewing the convergence from the other side, clearly all large-scale shared memory operations are ultimately implemented as message transactions at some level.

In addition to the convergence of scalable message-passing and shared memory machines, switch-based local area networks, including fast Ethernet, ATM, Fiber-Channel, and several proprietary designs (Boden et al. 1995; Gillett 1996) have emerged, providing scalable interconnects that are approaching what traditional parallel machines offer. These new networks are being used to connect collections of machines (which may be shared memory multiprocessors in their own right) into

clusters, which may operate as a parallel machine on individual large problems or as many individual machines on a multiprogramming load. Essentially all SMP vendors provide some form of network clustering to obtain better reliability.

In summary, message passing and a shared address space represent two clearly distinct programming models, each providing a well-defined paradigm for sharing, communication, and synchronization. However, the underlying machine structures have converged toward a common organization, represented by a collection of complete computers, augmented by a “communication assist” connecting each node to a scalable communication network. Thus, it is natural to consider supporting aspects of both in a common framework. Integrating the communication assist more tightly into the memory system tends to reduce the latency of network transactions and improve the bandwidth that can be supplied to or accepted from the network. We will want to look much more carefully at the precise nature of this integration and understand how it interacts with cache design, address translation, protection, and other traditional aspects of computer architecture.

1.2.5 Data Parallel Processing

A third important class of parallel machines has been variously called processor arrays, single-instruction-multiple-data machines, and data parallel architectures. The changing names reflect a gradual separation of the user-level abstraction from the machine operation. *The key characteristic of the data parallel programming model is that operations can be performed in parallel on each element of a large regular data structure, such as an array or matrix.* The program is logically a single thread of control, carrying out a sequence of either sequential or parallel steps. Within this general paradigm have been many novel designs, exploiting various technological opportunities, and considerable evolution as microprocessor technology has become such a dominant force.

An influential paper in the early 1970s (Flynn 1972) developed a taxonomy of computers, known as *Flynn's taxonomy*, which characterizes designs in terms of the number of distinct instructions issued at a time and the number of data elements they operate on: conventional sequential computers being *single-instruction-single-data* (SISD) and parallel machines built from multiple conventional processors being *multiple-instruction-multiple-data* (MIMD). The revolutionary alternative was *single-instruction-multiple-data* (SIMD). Its history is rooted in the mid-1960s when an individual processor was a cabinet full of equipment and an instruction fetch cost as much in time and hardware as performing the actual instruction. The idea was that all the instruction sequencing could be consolidated in the control processor. The data processors included only the ALU, memory, and a simple connection to nearest neighbors.

In the SIMD machines, the data parallel programming model was rendered directly in the physical hardware (Ball et al. 1962; Bouknight et al. 1972; Cornell 1972; Reddaway 1973; Slotnick, Borck, and McReynolds 1962; Slotnick 1967; Vick and Cornell 1978). Typically, a control processor broadcasts each instruction to an array of data processing elements (PEs), which are connected to form a regular grid,

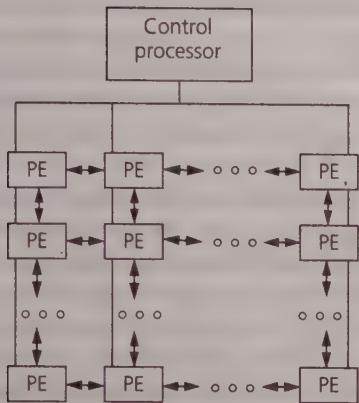


FIGURE 1.25 Typical organization of a data parallel (SIMD) machine. Individual processing elements (PEs) operate in lockstep under the direction of a single control processor. Traditionally, SIMD machines have provided a limited, regular interconnect among the PEs, although this was generalized in later machines, such as the Thinking Machines Corporation Connection Machine and the MasPar.

as suggested by Figure 1.25. It was observed that many important scientific computations involved uniform calculation on every element of an array or matrix, often involving neighboring elements in the row or column. Thus, the parallel problem data was distributed over the memories of the data processors, and scalar data was retained in the control processor's memory. The control processor instructed the data processors to each perform an operation on local data elements or to all perform a communication operation. For example, to average each element of a matrix with its four neighbors, a copy of the matrix would be shifted across the PEs in each of the four directions and a local accumulation performed in each PE. Data PEs typically included a condition flag, allowing some to abstain from an operation. In some designs, the local address could be specified with an indirect addressing mode, allowing all processors to do the same operation but with different local data addresses.

The development of arrays of processors was almost completely eclipsed in the mid-1970s with the development of vector processors. In these machines, a scalar processor is integrated with a collection of function units that operate on vectors of data out of one memory in a pipelined fashion. The ability to operate on vectors anywhere in memory eliminated the need to map application data structures onto a rigid interconnection structure and greatly simplified the problem of getting data aligned so that local operations could be performed. The first vector processor, the CDC Star-100, provided vector operations in its instruction set that combined two source vectors from memory and produced a result vector in memory. The machine only operated at full speed if the vectors were contiguous, and hence a large fraction of the execution time was spent simply transposing matrices. A dramatic change occurred in 1976 with the introduction of the CRAY-1, which extended the concept of a load-store architecture employed in the CDC 6600 and CDC 7600 (and rediscovered in modern RISC machines) to apply to vectors. Vectors in memory, of any fixed stride, were transferred to or from contiguous vector registers by vector load and store instructions. Arithmetic was performed on the vector registers. The use of a very fast scalar processor (operating at the unprecedented rate of 80 MHz), tightly

integrated with the vector operations and utilizing a large semiconductor memory rather than core, took over the world of supercomputing. Over the next twenty years, CRAY Research led the supercomputing market by increasing the bandwidth for vector memory transfers, the number of processors, the number of vector pipelines, and the length of the vector registers, resulting in the performance growth indicated in Figures 1.10 and 1.11.

The SIMD data parallel machine experienced a renaissance in the mid-1980s, as VLSI advances made simple 32-bit processors just barely practical (Batcher 1974, 1980; Hillis 1985; Nickolls 1990; Tucker and Robertson 1988). The unique twist in the data parallel regime was to place 32 very simple 1-bit processing elements on each chip, along with serial connections to neighboring processors, while consolidating the instruction sequencing capability in the control processor. In this way, systems with several thousand bit-serial processing elements could be constructed at reasonable cost. In addition, it was recognized that the utility of such a system could be increased dramatically with the provision of a general interconnect allowing an arbitrary communication pattern to take place in a single, rather long step, in addition to the regular grid neighbor connections (Hillis 1985; Hillis and Steele 1986; Nickolls 1990). The sequencing mechanism that expanded conventional integer and floating-point operations into a sequence of bit-serial operations also provided a means of “virtualizing” the processing elements, so that a few thousand processing elements could give the illusion of operating in parallel on millions of data elements with one virtual PE per data element.

The technological factors that made this bit-serial design attractive also provided fast, inexpensive, single-chip floating-point units and rapidly gave way to very fast microprocessors with integrated floating point and caches. This eliminated the cost advantage of consolidating the sequencing logic and provided equal peak performance on a much smaller number of complete processors. The simple, regular calculations on large matrices that motivated the data parallel approach also have tremendous spatial and temporal locality (if the computation is properly mapped onto a smaller number of complete processors), with each processor responsible for a large number of logically contiguous data points. Caches and local memory can be brought to bear on the set of data points local to each node while communication occurs across the boundaries or as a global rearrangement of data.

Thus, while the user-level abstraction of parallel operations on large regular data structures continued to offer an attractive solution to an important class of problems, the machine organization employed with data parallel programming models evolved toward a more generic parallel architecture of multiple cooperating microprocessors, much like scalable shared memory and message-passing machines, although several designs maintain specialized network support for global synchronization. One such example of network support is for a barrier, which causes each process to wait at a particular point in the program until all other processes have reached that point (Horiw et al. 1993; Leiserson et al. 1996; Kumar 1992; Kessler and Schwarzmeier 1993; Koeninger, Furtney, and Walker 1994). Indeed, the SIMD approach evolved into the SPMD (single-program-multiple-data) approach, in

which all processors execute copies of the same program, and has thus largely converged with the more structured forms of shared memory and message-passing programming.

Data parallel programming languages are usually implemented by viewing the local address spaces of a collection of processes, one per processor, as forming an explicit global address space. Data structures are laid out across this global address space with a simple mapping from indexes to processor and local offset. The computation is organized as a sequence of “bulk synchronous” phases of either local computation or global communication, separated by a global barrier (Valiant 1990). Because all processors perform communication together and share a global view of what is going on, either a shared address space or message passing can be employed. For example, if a phase involved every processor doing a write to an address in the processor “to the left,” it could be realized by each doing a send to the left and a receive “from the right” into the destination address. Similarly, every processor doing a read can be realized by every processor sending the address and then every processor sending back the data. In fact, the code that is produced by compilers for modern data parallel languages is essentially the same as for the structured control-parallel programs that are most common in shared memory and message-passing programming models. The convergence in machine structure has been accompanied by a convergence in how the machines are actually used.

1.2.6 Other Parallel Architectures

The mid-1980s renaissance gave rise to several other architectural directions that received considerable investigation by academia and industry, but enjoyed less commercial success than the three classes just discussed and therefore experienced less use as a vehicle for parallel programming. Two approaches that were developed into complete programming systems were dataflow architectures and systolic architectures. Both represent important conceptual developments of continuing value as the field evolves.

Dataflow Architecture

Dataflow models of computation sought to make the essential aspects of a parallel computation explicit at the machine level, without imposing artificial constraints that would limit the available parallelism in the program. The idea is that the program is represented by a graph of essential data dependences, as illustrated in Figure 1.26, rather than as a fixed collection of explicitly sequenced threads of control. An instruction may execute whenever its data operands are available. The graph may be spread arbitrarily over a collection of processors. Each node specifies an operation to perform and the address of each of the nodes that need the result. In the original form, a processor in a dataflow machine operates as a simple circular pipeline. A message, or *token*, from the network consists of data and an address, or *tag*, of its destination node. The tag is compared against those in a matching store. If

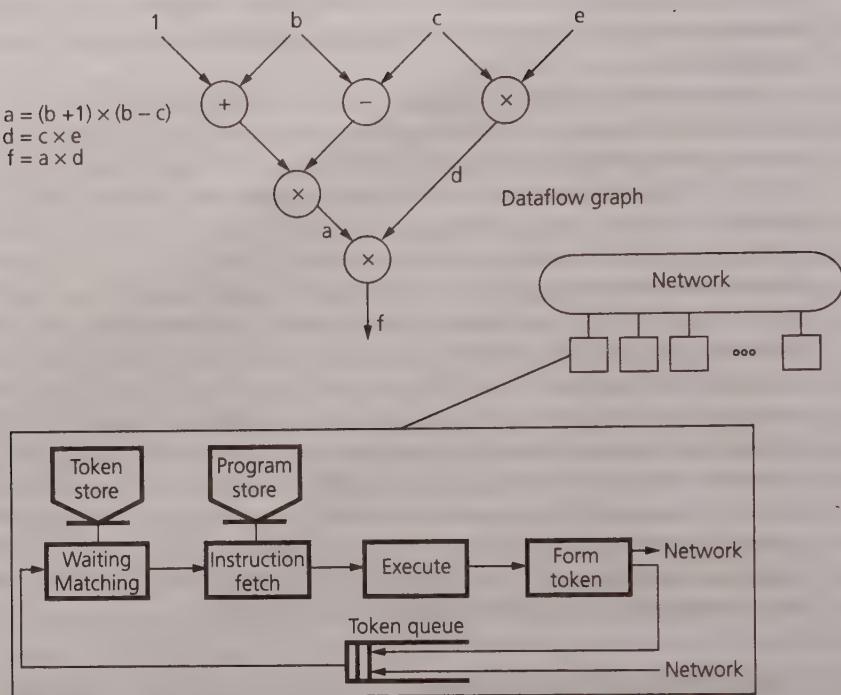


FIGURE 1.26 Dataflow graph and basic execution pipeline. A node in the graph fires when operands are present on its input. It produces results on its outputs that are delivered to adjacent nodes in the graph. The execution pipeline implements this firing rule by detecting when matching data tokens are present, fetching the corresponding instruction, performing the operation, and forming result tokens.

present, the matching token is extracted and the instruction is issued for execution. If not, the token is placed in the store to await its partner. When a result is computed, a new message or token containing the result data is sent to each of the destinations specified in the instruction. The same mechanism can be used whether the successor instructions are local or on a remote processor.

The primary division within dataflow architectures is whether the graph is *static*, with each node representing a primitive operation, or *dynamic*, in which case a node can represent the invocation of an arbitrary function, itself represented by a graph. In *dynamic*, or *tagged-token*, architectures, the effect of dynamically expanding the graph on function invocation is usually achieved by carrying additional context information in the tag, rather than actually modifying the program graph.

The key characteristics of dataflow architectures are the ability to name operations performed anywhere in the machine, the support for synchronization of independent operations, and dynamic scheduling at the machine level. As the dataflow machine designs matured into real systems programmed in high-level parallel

languages, a more conventional structure emerged. Typically, parallelism was generated in the program as a result of parallel function calls and parallel loops, so it was attractive to allocate these larger chunks of work to processors. This led to a family of designs organized essentially like the NUMA design of Figure 1.19, the key differentiating features being direct support for a large, dynamic set of threads of control and the integration of communication with thread generation. The network is closely integrated with the processor; in many designs, the “current message” is available in special registers, and hardware support is available for dispatching to a thread identified in the message. In addition, many designs provide extra state bits on memory locations in order to provide fine-grained synchronization (i.e., synchronization on an element-by-element basis) rather than using locks to synchronize accesses to an entire data structure. In particular, each message could schedule a chunk of computation that could make use of local registers and memory.

By contrast, in shared memory machines, the generally adopted view is that a static or slowly varying set of processes operates within a shared address space, so the compiler or program maps the logical parallelism in the program to a set of processes by assigning loop iterations, maintaining a shared work queue, or the like. Similarly, message-passing programs involve a static, or nearly static, collection of processes that can name one another in order to communicate. In data parallel architectures, the compiler or sequencer maps a large set of “virtual processor” operations onto processors by assigning iterations of a regular loop nest. In the dataflow case, the machine provides the ability to name a very large and dynamic set of threads that can be mapped arbitrarily to processors. Typically, these machines provide a global address space as well. As was the case with message-passing and data parallel machines, dataflow architectures experienced a gradual separation of programming model and hardware structure as the approach matured.

Systolic Architectures

Another novel approach was *systolic architectures*, which sought to replace a single sequential processor by a regular array of simple processing elements and, by carefully orchestrating the flow of data between PEs, obtain very high throughput with modest memory bandwidth requirements. These designs differ from conventional pipelined function units in that the array structure can be nonlinear (e.g., hexagonal), the pathways between PEs may be multidirectional, and each PE may have a small amount of local instruction and data memory. They differ from SIMD in that each PE might do a different operation.

The early proposals were driven by the opportunity offered by VLSI to provide inexpensive special-purpose chips. A given algorithm could be represented directly as a collection of specialized computational units connected in a regular, space-efficient pattern. Data would move through the system at regular “heartbeats” as determined by local state. Figure 1.27 illustrates a design for computing convolutions using a simple linear array. At each beat the input data advances to the right, is multiplied by a local weight, and is accumulated into the output sequence as it also

$$y(i) = w1 \times x(i) + w2 \times x(i+1) + w3 \times x(i+2) + w4 \times x(i+3)$$

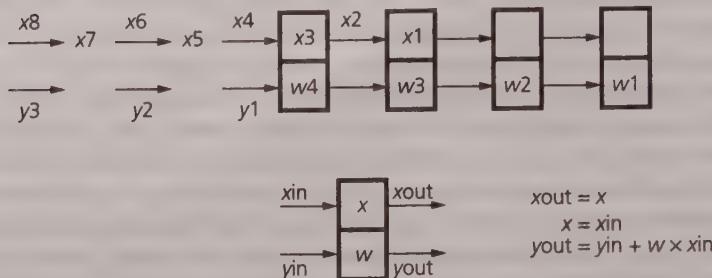


FIGURE 1.27 Systolic array computation of an inner product. Each box represents a computational unit performing a specific function. Every time the clock beats, all units accept inputs, compute results, and generate outputs. Data moves through the systolic array with each beat.

advances to the right. The systolic approach has aspects in common with message-passing, data parallel, and dataflow models but takes on a unique character for a specialized class of problems.

Practical realizations of these ideas, such as iWarp (Borkar et al. 1990), provided quite general programmability in the nodes, so that a variety of algorithms could be realized on the same hardware. The key differentiation is that the network can be configured as a collection of dedicated channels, representing the systolic communication pattern, and data can be transferred directly from processor registers to processor registers across a channel. The global knowledge of the communication pattern is exploited to reduce contention and even to avoid deadlock. The key characteristic of systolic architectures is the ability to integrate highly specialized computation under simple, regular, and highly localized communication patterns.

Systolic algorithms have also been generally amenable to solutions on generic machines, using the fast barrier to delineate coarser-grained phases. The regular, local communication pattern of these algorithms yields good locality when large portions of the logical systolic array are executed on each process, the communication bandwidth needed is low, and the synchronization requirements are simple. Thus, these algorithms have proved effective on the entire spectrum of parallel machines.

1.2.7 A Generic Parallel Architecture

In examining the evolution of the major approaches to parallel architecture, we see a clear convergence for scalable machines toward a generic parallel machine organization, illustrated in Figure 1.28. The machine comprises a collection of essentially complete computers, each with one or more processors and memory, connected through a scalable communication network via *communication assist*—a controller

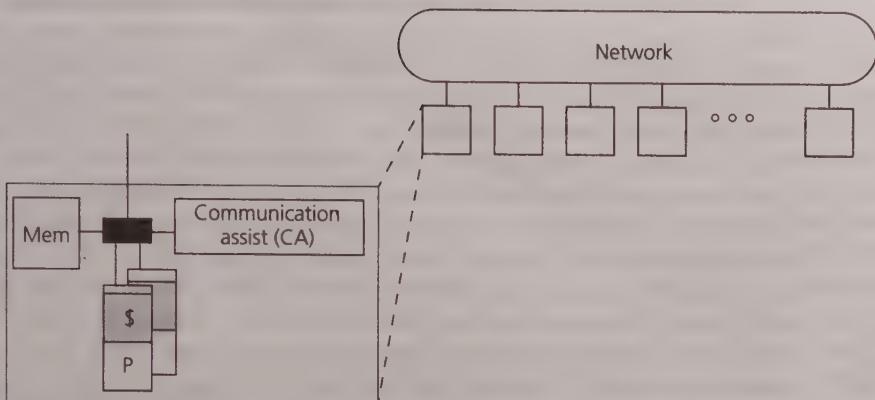


FIGURE 1.28 Generic scalable multiprocessor organization. A collection of essentially complete computers, including one or more processors and memory, communicating through a general-purpose, high-performance, scalable interconnect. Typically, each node contains a controller that assists in communication operations across the network.

or auxiliary processing unit that assists in generating outgoing messages or handling incoming messages. While the consolidation within the field may seem to narrow the design space, in fact, great diversity and debate remains, centered on what functionality should be provided within the assist and how it interfaces to the processor, memory system, and network. Recognizing that these are specific differences within a largely similar organization helps us to understand and evaluate the important organizational trade-offs.

Not surprisingly, different programming models place different requirements on the design of the communication assist and influence which operations are common and should be optimized. In the shared memory case, the assist is tightly integrated with the memory system in order to capture the memory events that may require interaction with other nodes. The assist must also accept messages and perform memory operations and state transitions on behalf of other nodes. In the message-passing case, communication is initiated by explicit actions, either at the system or user level, so it is not required that memory system events be observed. Instead, a need exists to initiate the messages quickly and to respond to incoming messages. The response may require that a tag match be performed, that buffers be allocated, that data transfer commence, or that an event be posted. The data parallel and systolic approaches place an emphasis on fast global synchronization, which may be supported directly in the network or in the assist. Dataflow places an emphasis on fast dynamic scheduling of computation based on an incoming message. Systolic algorithms present the opportunity to exploit global patterns in local scheduling. Even with these differences, it is important to observe that all of these approaches share common aspects; they need to initiate network transactions as a result of specific processor events, and they need to perform simple operations on the remote node to carry out the desired event.

We also see that a separation has emerged between programming model and machine organization as parallel programming environments have matured. For example, Fortran 90 and High Performance Fortran provide a shared address, data parallel programming model that is implemented on a wide range of machines—some supporting a shared physical address space, others with only message passing. The compilation techniques for these machines differ radically, even though the machines appear organizationally similar, because of differences in communication and synchronization operations provided in the communication abstraction and vast differences in the performance characteristics of these operations. As a second example, popular message-passing libraries, such as PVM (parallel virtual machine) and MPI (message-passing interface), are implemented on this same range of machines, but the implementation of the libraries differs dramatically from one kind of machine to another. The same observations hold for parallel operating systems.

1.3 FUNDAMENTAL DESIGN ISSUES

Given how the state of the art in parallel architecture has advanced, we need to take a fresh look at how to organize the body of material in the field. Traditional machine taxonomies, such as SIMD/MIMD, are of little help since multiple general-purpose processors are so dominant. We cannot focus entirely on programming models since in many cases widely differing machine organizations support a common programming model. We cannot just look at hardware structures either, since common elements are employed in many different ways. Instead, we should focus our attention on the architectural distinctions that make a difference to the software that is to run on the machine. In particular, we need to highlight those aspects that influence how a compiler would generate code from a high-level parallel language, how a library writer would code a well-optimized library, or how an application would be written in a low-level parallel language. We can then approach the design problem as one that is constrained from above by how programs use the machine and from below by what the basic technology can provide.

The guiding principles presented in this book for understanding modern parallel architecture are indicated by the layers of abstraction shown in Figure 1.13. Fundamentally, we must understand the operations that are provided at the user-level communication abstraction, how various programming models are mapped to these primitives, and how these primitives are mapped to the actual hardware. Excessive emphasis on the high-level programming model without attention to how it can be mapped to the machine would detract from understanding the fundamental architectural issues, as would excessive emphasis on the specific hardware mechanisms in each particular machine.

This section looks more closely at the communication abstraction and the basic requirements of a programming model. It then defines more formally the key concepts that tie the layers together: naming, ordering, and communication and replication of data. Finally, it introduces the basic performance models required to resolve design trade-offs.

1.3.1 Communication Abstraction

The communication abstraction forms the key interface between the programming model and the system implementation. It plays a role very much like the instruction set in conventional sequential computer architecture. Viewed from the software side, it must have a precise, well-defined meaning so that the same program will run correctly on many implementations. In addition, the operations provided at this layer must be simple, composable entities with clear costs, so that the software can be optimized for performance. Viewed from the hardware side, it must also have a well-defined meaning so that the machine designer can determine where performance optimizations can be performed without violating the software assumptions. While the abstraction needs to be precise, the machine designer would like it not to be overly specific, so it does not prohibit useful techniques for performance enhancement or frustrate efforts to exploit properties of newer technologies.

The communication abstraction is, in effect, a contract between the hardware and the software allowing each the flexibility to improve what it does while working correctly together. To understand the “terms” of this contract, we need to look more carefully at the basic requirements of a programming model.

1.3.2 Programming Model Requirements

A parallel program consists of one or more threads of control operating on data. A *parallel programming model* specifies what data can be *named* by the threads, what *operations* can be performed on the named data, and what *ordering* exists among these operations.

To make these issues concrete, consider the programming model for a uniprocessor. A thread can name the locations in its virtual address space and can name machine registers. In some systems, the address space is broken up into distinct code, stack, and heap segments whereas in others it is flat. Similarly, different programming languages provide access to the address space in different ways; for example, some allow pointers and dynamic storage allocation, others do not. Regardless of these variations, the instruction set provides the operations that can be performed on the named locations. For example, in RISC machines the thread can load data from or store data to memory but can perform arithmetic and comparisons only on data in registers. Older instruction sets support arithmetic on either. Compilers typically mask these differences at the hardware/software boundary, so the user's programming model is one of performing operations on variables that hold data. The hardware translates each virtual address to a physical address on every operation.

The ordering among memory operations is *sequential program order*. The programmer's view is that variables are read and modified in the top-to-bottom, left-to-right order specified in the program. More precisely, the value returned by a read to an address is the last value written to the address in the sequential execution order of the program. This ordering assumption is essential to the logic of the program. However, the reads and writes may not actually be performed in program order

because the compiler performs optimizations when translating the program to the instruction set and the hardware performs optimizations when executing the instructions. Both make sure the program cannot tell that the order has been changed. The compiler and hardware preserve the *dependence order*, that is, if a variable is written and then read later in the program order, they make sure that the later operation uses the proper value, but they may avoid actually writing and reading the value to and from memory or may defer the write until later. Collections of reads with no intervening writes may be completely reordered and, generally, writes to different addresses can be reordered as long as dependences from intervening reads are preserved. This reordering occurs at the compilation level, for example, when the compiler allocates variables to registers, manipulates expressions to improve pipelining, or transforms loops to reduce overhead and improve the data access pattern. It occurs at the machine level when instruction execution is pipelined, multiple instructions are issued per cycle, or write buffers are used to hide memory latency. We depend on these optimizations for performance. They work because for the program to observe the effect of a write, it must read the variable; this creates a dependence, which is preserved. Thus, the illusion of program order is preserved while actually executing the program in the weaker dependence order.⁶ We operate in a world where essentially all programming languages embody a programming model of sequential order of operations on variables in a virtual address space, and the system enforces a weaker order wherever it can do so without changing the results of the program.

Now let's return to parallel programming models. The informal discussion earlier in this chapter indicated the distinct positions adopted on naming, operation set, and ordering. Naming and operation set are what typically characterize the models; however, ordering is of key importance. A parallel program must coordinate the activity of its threads to ensure that the dependences within the program are enforced; this requires explicit synchronization operations when the ordering implicit in the basic operations is not sufficient. As architects (and compiler writers), we need to understand the ordering properties to see what optimization “tricks” we can play for performance. We can focus on shared address and message-passing programming models since they are the most widely used; other models, such as data parallel, are usually implemented in terms of one of them.

The shared address space programming model assumes one or more threads of control, each operating in an address space that contains a region shared between threads, and may contain a region that is private to each thread. Typically, the shared region is shared by all threads. All the operations defined on private addresses are defined on shared addresses; in particular, the program accesses and updates shared variables simply by using them in expressions and assignment statements.

Message-passing models assume a collection of processes each operating in a private address space and each able to name the other processes. The normal unipro-

6. The illusion breaks down a little bit for system programmers, say, if the variable is actually a control register on a device. Then the actual program order must be preserved. This is usually accomplished by flagging the variable as special; for example, using the volatile type modifier in C.

cessor operations are provided on the private address space, in program order. The additional operations, send and receive, operate on the local address space and the global process space. Send transfers data from the local address space to a process. Receive accepts data into the local address space from a process. Each send/receive pair is a specific point-to-point synchronization operation. Many message-passing languages offer global, or collective, communication operations as well, such as broadcast.

Naming

The position adopted on naming in the programming model is presented to the programmer through the programming language or programming environment. It is what the logic of the program is based upon. However, the issue of naming is critical at each level of the communication architecture. Certainly one possible strategy is to have the operations in the programming model be one to one with the operations in the communication abstraction at the user/system boundary and to have these be one to one with the hardware primitives. However, it is also possible for the compiler and libraries to provide a level of translation between the programming model and the communication abstraction, or for the operating system to intervene to handle some of the operations at the user/system boundary. These alternatives allow the architect to consider implementing the common, simple operations directly in hardware and supporting the more complex operations partly or wholly in software.

Let us consider the ramifications of naming at the layers using the two primary programming models: shared address and message passing. First, in a shared address model, accesses to shared variables in the program are usually mapped by the compiler to load and store instructions on shared virtual addresses, just like access to any other variable. This is not the only option, however. The compiler could generate special code sequences for accesses to shared variables. A machine supports a *global physical address space* if any processor is able to generate a physical address for any location in the machine and access the location in a single memory operation. It is straightforward to realize a shared virtual address space on a machine providing a global physical address space: establish the virtual-to-physical mapping so that shared virtual addresses map to the same physical location (i.e., the processes have the same entries in their page tables). However, the existence of the level of translation allows for other approaches. A machine supports *independent local physical address spaces* if each processor can only access a distinct set of locations. Even on such a machine, a shared virtual address space can be provided by mapping virtual addresses that are local to a process to the corresponding physical address. The non-local addresses are left unmapped so upon access to a nonlocal shared address a page fault will occur, allowing the operating system to intervene and access the remote shared data. Although this approach can provide the same naming, operations, and ordering to the program, it clearly has different hardware requirements at the hardware/software boundary. The architect's job is to resolve these design trade-offs across layers of the system implementation so that the result is efficient and cost-effective for the target application workload on available technology.

Second, message-passing operations could be realized directly in hardware, but the matching and buffering aspects of the send/receive operations are better suited to software implementation. More basic data transport primitives are well supported in hardware. Thus, in essentially all parallel machines, the message-passing programming model is realized via a software layer that is built upon a simpler communication abstraction. At the user/system boundary, one approach is to have all message operations go through the operating system as if they were I/O operations. However, the frequency of message operations is much greater than I/O operations, so it makes sense to use the operating system support to set up resources, privileges, and so on and allow the frequent, simple data transfer operations to be supported directly in hardware. On the other hand, we might consider adopting a shared virtual address space as the lower-level communication abstraction, in which case send and receive operations involve writing and reading shared buffers and posting the appropriate synchronization events.

The issue of naming arises at each level of abstraction in a parallel architecture, not just in the programming model. As architects, we need to design against the frequency and type of operations that occur at the communication abstraction, understanding that the trade-offs at this boundary involve what is supported directly in hardware and in software.

Operations

Each programming model defines a specific set of operations that can be performed on the data or objects that can be named within the model. For the case of a shared address model, these include reading and writing shared variables as well as various atomic read-modify-write operations on shared variables, which are used to synchronize the threads. For message passing, the operations are send and receive on private (local) addresses and process identifiers, as described previously. Each element of data in the program is named by a process number and a local address within the process. A message-passing model does define a global address space of sorts. However, no operations are defined on these global addresses. They can be passed around and interpreted by the program, for example, to emulate a shared address style of programming on top of message passing, but they cannot be operated on directly at the communication abstraction. As architects, we need to be aware of the operations defined at each level of abstraction. In particular, we need to be very clear on what ordering among operations is assumed to be present at each level of abstraction, where communication takes place, and how data is replicated.

Ordering

The properties of the specified order among operations have a profound effect throughout the layers of parallel architecture. Notice, for example, that the message-passing model places no assumption on the ordering of operations by distinct processes except the explicit program order associated with the send/receive operations,

whereas a shared address model must specify aspects of how processes see the order of operations performed by other processes. Ordering issues are important and rather subtle. Many of the tricks that we play for performance in the uniprocessor context involve relaxing the order assumed by the programmer to gain performance, either through parallelism or improved locality or both. Exploiting parallelism and locality is even more important in the multiprocessor case. Thus, we need to understand what new tricks can be played. We also need to examine which of the old tricks are still valid. Can we perform the traditional sequential optimizations at the compiler and architecture level on each process of a parallel program? Where can the explicit synchronization operations be used to allow ordering to be relaxed on the conventional operations? To answer these questions, we need to develop a much more complete understanding of how programs use the communication abstraction, what properties they rely upon, and what machine structures we would like to exploit for performance.

A natural position to adopt on ordering is that operations in a thread are in program order. That is what the programmer would assume for the special case of one thread. However, there remains the question of what ordering can be assumed among operations performed on shared variables by different threads. The threads operate independently and, potentially, at different speeds so no clear notion of “latest” is defined. If we have in mind that the machines behave as a collection of simple processors operating on a common, centralized memory, then it is reasonable to expect the global order of memory accesses to be some arbitrary interleaving of the individual program orders. In reality we won’t build the machines this way, but it establishes what operations are implicitly ordered by the basic operations in the model. This interleaving is also what we expect of a collection of threads that are time-shared, perhaps at a very fine level, on a uniprocessor.

Where the implicit ordering is not enough, explicit synchronization operations are required. Parallel programs require two types of synchronization:

- *Mutual exclusion* ensures that certain operations on certain data are performed by only one thread or process at a time. We can imagine a room that must be entered to perform such an operation, and only one process can be in the room at a time. This is accomplished by locking the door upon entry and unlocking it on exit. If several processes arrive at the door together, only one will get in and the others will wait until it leaves. The order in which the processes are allowed to enter does not matter and may vary from one execution of the program to the next; what matters is that they do so one at a time. Mutual exclusion operations tend to serialize the execution of processes.
- *Events* are used to inform other processes that some point of execution has been reached so that they can proceed knowing that certain dependences have been satisfied. These operations are like passing a baton from one runner to the next in a relay race or the starter firing a gun to indicate the start of a race. If one process writes a value that another is supposed to read, an event synchronization operation must take place to indicate that the value is ready to be read. Events may be *point-to-point*, involving a pair of processes, or they may be *global*, involving all processes or a group of processes.

1.3.3 Communication and Replication

The final issues that are closely tied to the layers of parallel architecture are communication and data replication. Communication and replication are inherently related. Consider first a message-passing operation. The effect of the send/receive pair is to copy data that is in the sender's address space into a region of the receiver's address space. This transfer is essential for the receiver to access the data. If the data is produced by the sender, it reflects a *true communication* of information from one process to the other. If the data just happens to be stored at the sender, perhaps because that was the initial configuration of the data or because the data set was simply too large to fit on any one node, then this transfer merely makes a replica of the data where it is used. In this case, the processes are not actually communicating information from one to another via the data transfer. If the data were replicated or positioned properly over the processes to begin with, there would be no need to communicate it in a message. More importantly, if the receiver uses the data over and over again, it can reuse its replica without additional data transfers. The sender can modify the region of addresses that was previously communicated with no effect on the previous receiver. If the effect of these later updates is to be communicated, an additional transfer must occur.

Consider now a conventional data access on a uniprocessor through a cache. If the cache does not contain the desired address, a miss occurs and the block is transferred from the memory that serves as a backing store. The data is implicitly replicated into the cache near the processor that accesses it. If the processor reuses the data while it resides in the cache, further transfers with the memory are avoided. In the uniprocessor case, the processor produces the data and the processor consumes it, so the "communication" with the memory occurs only because the data does not fit in the cache or is being accessed for the first time.

Interprocess communication and data transfer within the storage hierarchy become melded together in a shared physical address space. Cache misses cause a data transfer across the machine interconnect whenever the physical backing storage for an address is remote to the node accessing the address, whether the address is private or shared and whether the transfer is a result of true communication or just a data access. The natural tendency of the machine is to replicate data into the caches of the processors that access the data. If the data is reused while it is in the cache, no data transfers occur; this is a major advantage. However, when a write to shared data occurs, something must be done to ensure that later reads by other processors get the new data rather than the old data that was replicated into their caches. This will involve more than a simple data transfer.

To be clear on the relationship of communication and replication, it is important to distinguish several concepts that are frequently bundled together. When a program performs a write, it binds a data value to an address; a read obtains the data value bound to an address. The data resides in some physical storage element in the machine. A *data transfer* occurs whenever data in one storage element is transferred into another. This does not necessarily change the bindings of addresses and values. The same data may reside in multiple physical locations as it does in the uniprocessor storage hierarchy, but the one nearest to the processor is the only one that the

processor can observe. If it is updated, the other hidden replicas, including the actual memory location, must eventually be updated. Copying data binds a new set of addresses to the same set of values. Generally, this will cause data transfers. Once the copy has been made, the two sets of bindings are completely independent (unlike the implicit replication that occurs within the storage hierarchy), so updates to one set of addresses do not affect the other. *Communication* between processes occurs when data written by one process is read by another. This may cause a data transfer within the machine, either on the write or the read, or the data transfer may occur for other reasons. Communication may involve establishing a new binding or not doing so, depending on the particular communication abstraction.

In general, replication avoids unnecessary communication; that is, transferring data to a consumer that was not produced since the data was previously accessed. The ability to perform replication automatically at a given level of the communication architecture depends very strongly on the naming and ordering properties of the layer. Moreover, replication is not a panacea—it too requires data transfers. It is disadvantageous to replicate data that is not going to be used. We will see that replication plays an important role throughout parallel computer architecture.

1.3.4 Performance

In defining the set of operations for communication and cooperation, the data types, and the addressing modes, the communication abstraction specifies how shared objects are named, what ordering properties are preserved, and how synchronization is performed. However, the performance characteristics of the available primitives determine how they are actually used. Programmers and compiler writers will avoid costly operations where possible. In evaluating architectural trade-offs, the decision between feasible alternatives ultimately rests upon the performance they deliver. Thus, to complete an introduction to the fundamental issues of parallel computer architecture, we need a framework for understanding performance at many levels of design.

Fundamentally, there are three important metrics: *latency*, the time taken for an operation; *bandwidth*, the rate at which operations are performed; and *cost*, the impact these operations have on the execution time of the program. In a simple world where processors do only one thing at a time, these metrics are directly related—the bandwidth (operations per second) is the reciprocal of the latency (seconds per operation), and the cost is simply the latency times the number of operations performed. However, modern computer systems do many different operations at once, and the relationship between these performance metrics is much more complex. Consider the following basic example.

EXAMPLE 1.2 Suppose a component can perform a specific operation in 100 ns. Clearly, it can support a bandwidth of 10 million operations per second. However, if the component is pipelined internally as 10 equal stages, it is able to provide a peak bandwidth of 100 million operations per second. The rate at which operations can be initiated is determined by how long the slowest stage is occupied, 10 ns, rather than by the latency of an individual operation. The bandwidth delivered on

an application depends on how frequently it initiates the operations. If the application starts an operation every 200 ns, the delivered bandwidth is 5 million operations per second, regardless of how the component is pipelined. Of course, usage of resources is usually bursty, so pipelining can be advantageous even when the average initiation rate is low. If the application performed 100 million operations on this component, what is the range of cost of these operations?

Answer Taking the operation count times the operation latency would give an upper bound of 10 seconds. Taking the operation count divided by the peak rate gives a lower bound of 1 second. The former is accurate if the program waited for each operation to complete before continuing. The latter assumes that the operations are completely overlapped with other useful work, so the cost is simply the cost to initiate the operation. Suppose that on average the program can do 50 ns of useful work after each operation issued to the component before it depends on the operations result. Then the cost to the application is 50 ns per operation—the 10 ns to issue the operation and the 40 ns spent waiting for it to complete—so the total cost is 5 seconds. ■

Since the unique property of parallel computer architecture is communication, the operations that we are concerned with most often are data transfers. The performance of these operations can be understood as a generalization of our basic pipeline example.

Data Transfer Time

The time for a data transfer operation is generally described by a linear model:

$$\text{Transfer Time}(n) = T_0 + \frac{n}{B} \quad (1.3)$$

where n is the amount of data (e.g., number of bytes), B is the transfer rate of the component moving the data in compatible units (e.g., bytes per second), and the constant term, T_0 , is the start-up cost. This is a very convenient model, and it is used to describe a diverse collection of operations, including messages, memory accesses, bus transactions, and vector operations. For message passing, the start-up cost can be thought of as the time for the first bit to get to the destination. For memory operations, it is essentially the access time. For bus transactions, it reflects the bus arbitration and command phases. For any sort of pipelined operation, including pipelined instruction processing or vector operations, it is the time to fill the pipeline.

Using this simple model, it is clear that the bandwidth of a data transfer operation depends on the transfer size. As the transfer size increases, it approaches the asymptotic rate of B , which is sometimes referred to as r_∞ . How quickly it approaches this rate depends on the start-up cost. It is easily shown that the size at which half of the peak bandwidth is obtained, the *half-power point*, is given by

$$n_{\frac{1}{2}} = \frac{T_0}{B} \quad (1.4)$$

Unfortunately, this linear model does not give any indication of when the next such operation can be initiated, nor does it indicate whether other useful work can be performed during the transfer. These other factors depend on how the transfer is performed.

Overhead and Occupancy

The data transfer in which we are most interested is the one that occurs across the network in parallel machines. It is initiated by the processor through the communication assist. The essential components of this operation can be described by the following simple model:

$$\text{Communication Time}(n) = \text{Overhead} + \text{Occupancy} + \text{Network Delay} \quad (1.5)$$

The *overhead* is the time the processor spends initiating the transfer. This may be a fixed cost, if the processor simply has to tell the communication assist to start, or it may be linear in n , if the processor has to copy the data into the assist. The key point is that this is time the processor is busy with the communication event; it cannot do other useful work or initiate other communication during this time. The remaining portion of the communication time is considered the *network latency*; it is the part that can be hidden by other processor operations.

The *occupancy* is the time it takes for the data to pass through the slowest component on the communication path. For example, each link that is traversed in the network will be occupied for time n/B , where B is the bandwidth of the link. The data will occupy other resources, including buffers, switches, and the communication assist. Often the communication assist is the bottleneck that determines the occupancy. The occupancy limits how frequently communication operations can be initiated. The next data transfer will have to wait until the critical resource is no longer occupied before it can use that same resource. If there is buffering between the processor and the bottleneck, the processor may be able to issue a burst of transfers at a frequency greater than $1/\text{Occupancy}$; however, once this buffer is full, the processor must slow to the rate set by the occupancy. A new transfer can start only when an older one finishes.

The remaining communication time is lumped into the *network delay*, which includes the time for a bit to be routed across the actual network as well as many other factors, such as the time to get through the communication assists. From the processor's viewpoint, the specific hardware components contributing to network delay are indistinguishable. What affects the processor is how long it must wait before it can use the result of a communication event, how much of this time it can use for other activities, and how frequently it can communicate data. Of course, the task of designing the network and its interfaces is very concerned with the specific components and their contribution to the aspects of performance that the processor observes.

In the simple case where the processor issues a request and waits for the response, the breakdown of the communication time into its three components is immaterial.

All that matters is the total round-trip time. However, in the case where multiple operations are issued in a pipelined fashion, each of the components has a specific influence on the delivered performance.

Indeed, every individual component along the communication path can be described by its delay and its occupancy. The network delay is simply the sum of the delays along the path. The network occupancy is the maximum of the occupancies along the path. For interconnection networks, an additional factor arises because many transfers can take place simultaneously. If two of these transfers attempt to use the same resource at once (e.g., they use the same wire at the same time), one must wait. This *contention* for resources increases the average communication time. From the processor's viewpoint, contention appears as increased occupancy. Some resource in the system is occupied for a time determined by the collection of transfers across it.

Equation 1.5 is a very general model. It can be used to describe data transfers in many places in modern, highly pipelined computer systems. As one example, consider the time to move a block between cache and memory on a miss. The cache controller spends a period of time inspecting the tag to determine that it is not a hit and then starting the transfer; this is the overhead. The occupancy is the block size divided by the bus bandwidth, unless there is some slower component in the system. The delay includes the normal time to arbitrate and gain access to the bus plus the time spent delivering data into the memory. Additional time spent waiting to gain access to the bus or waiting for the memory bank cycle to complete is due to contention. A second obvious example is the time to transfer a message from one processor to another.

Communication Cost

The bottom line is, of course, the time a program spends performing communication. A useful model connecting the program characteristics to the hardware performance is given by the following:

$$\text{Communication Cost} = \text{Frequency} \times (\text{Communication Time} - \text{Overlap}) \quad (1.6)$$

The *frequency of communication*, defined as the number of communication operations per unit of work in the program, depends on many programming factors (as we will see in Chapters 2 and 3) and many hardware design factors. In particular, hardware may limit the transfer size and thereby determine the minimum number of messages. It may automatically replicate data or migrate it to where it is used. However, a certain amount of communication is inherent to parallel execution since data must be shared and processors must coordinate their work. In general, for a machine to support programs with a high communication frequency, the other parts of the communication cost equation must be small—low overhead, low network delay, and small occupancy. The attention paid to communication costs essentially determines which programming models a machine can realize efficiently and what portion of the application space it can support. Any parallel computer with good computational performance can support programs that communicate infrequently, but as the

frequency or volume of communication increase, greater stress is placed on the communication architecture.

The *overlap* is the portion of the communication operation that is performed concurrently with other useful work, including computation or other communication. This reduction of the effective cost is possible because much of the communication time involves work done by components of the system other than the processor, such as the communication assist, the bus, the network, or the remote processor or memory. Overlapping communication with other work is a form of small-scale parallelism, as is the instruction-level parallelism exploited by fast microprocessors. In effect, we may invest some of the available parallelism in a program to hide the actual cost of communication.

1.3.5 Summary

The issues of naming, operation set, and ordering apply at each level of abstraction in a parallel architecture, not just the programming model. In general, a level of translation or run-time software may intervene between the programming model and the communication abstraction, and beneath this abstraction are key hardware abstractions. At any level, communication and replication are deeply related. Whenever two processes access the same data, the data either needs to be communicated between the two or replicated so each can access a copy of it. The ability to have the same name refer to two distinct physical locations in a meaningful manner at a given level of abstraction depends on the position adopted on naming and ordering at that level. Wherever data movement is involved, we need to understand its performance characteristics in terms of latency and bandwidth and, furthermore, how these are influenced by overhead and occupancy. As architects, we need to design against the frequency and type of operations that occur at the communication abstraction, understanding that trade-offs occur across this boundary, involving what is supported directly in hardware and what is supported in software. The position adopted on naming, operation set, and ordering at each of these levels has a qualitative impact on these trade-offs, as we will see throughout the book.

1.4 CONCLUDING REMARKS

Parallel computer architecture forms an important thread in the evolution of computer architecture, rooted essentially in the beginnings of computing. For much of this history it takes on a novel, even exotic role as the avenue for advancement over and beyond what the base technology can provide. Parallel computer designs have demonstrated a rich diversity of structure, usually motivated by specific higher-level parallel programming models. However, the dominant technological forces of the VLSI generation have pushed parallelism increasingly into the mainstream, making parallel architecture almost ubiquitous. All modern microprocessors are highly parallel internally, executing several bit-parallel instructions in every cycle and even reordering instructions within the limits of inherent dependences to mitigate the

costs of communication with hardware components external to the processor itself. These microprocessors have become the performance and price-performance leaders of the computer industry. From the most powerful supercomputers to departmental servers to the desktop, we see systems constructed by utilizing multiples of such processors integrated into a communications fabric. This technological focus, and increasing maturity of compiler technology, has brought about a dramatic convergence in the structural organization of modern parallel machines. The key architectural issue is how communication is integrated into the memory and I/O systems that form the remainder of the computational node. This communications architecture reveals itself functionally in terms of what can be named at the hardware level, what ordering guarantees are provided, and how synchronization operations are performed whereas, from a performance point of view, we must understand the inherent latency and bandwidth of the available communication operations. Thus, modern parallel computer architecture carries with it a strong engineering component, amenable to quantitative analysis of cost and performance trade-offs.

This book presents the conceptual foundations as well as the engineering issues of parallel computer architecture across a broad range of potential scales of design, all of which have an important role in computing today and in the future. Computer systems, whether parallel or sequential, are designed against the requirements and characteristics of intended workloads. For conventional computers, we assume that most practitioners in the field have a good understanding of what sequential programs look like, how they are compiled, and what level of optimization is reasonable to assume that the programmer has performed. Thus, we are comfortable taking popular sequential programs, compiling them for a target architecture, and drawing conclusions from running the programs or evaluating execution traces. When we attempt to improve performance through architectural enhancements, we assume that the program is reasonably good in the first place.

The situation with parallel computers is quite different. Much less general understanding exists about the process of parallel programming, and programmer and compiler optimizations have a wider scope, which can greatly affect the program characteristics exhibited at the machine level.

Chapter 2 provides an overview of parallel programs—what they look like and how they are constructed. Chapter 3 explains the issues that must be addressed by the programmer and compiler to construct a “good” parallel program, that is, one that is effective enough in using multiple processors to form a reasonable basis for architectural evaluation. Ultimately, we design parallel computers against the program characteristics at the machine level, so the goal of Chapter 3 is to draw a connection between what appears in the program text and how the machine spends its time. In effect, Chapters 2 and 3 take us from a general understanding of issues at the application level to a specific understanding of the character and frequency of operations at the communication abstraction level.

Chapter 4 establishes a framework for workload-driven evaluation of parallel computer designs. Two related scenarios are addressed. First, for a parallel machine that has already been built, we need a sound method of evaluating its performance. This proceeds by first determining the capability of individual aspects of the

machine in isolation and then measuring how well they perform collectively. The understanding of application characteristics is important to ensure that the workload run on the machine stresses the various aspects of interest. Second, we need a process for evaluating hypothetical architectural advancements. New ideas for which no machine exists need to be evaluated through simulations, which imposes severe restrictions on what can reasonably be executed. Again, an understanding of application characteristics and how they scale with problem and machine size is crucial to navigating the design space.

Chapters 5 and 6 study in detail the design of symmetric multiprocessors with a shared physical address space. Going deeply into the small-scale case before examining scalable designs is important for several reasons. First, small-scale multiprocessors are the most prevalent form of parallel architecture; they are likely to be the form most students are exposed to, most software developers are targeting, and most professional designers are dealing with. Second, the issues that arise in the small scale are indicative of what is critical in the large scale, but the solutions are often simpler and easier to grasp. Thus, these chapters provide a study in the small of what the following five chapters address in the large. Third, the small-scale multiprocessor design is a fundamental building block for the larger-scale machines. The available options for interfacing a scalable interconnect with a processor-memory node are largely circumscribed by the processor, cache, and memory structure of the small-scale machines. Finally, the solutions to key design problems in the small-scale case are elegant in their own right.

The fundamental building block for the designs in Chapters 5 and 6 is the shared bus between processors and memory. The basic problem that we need to solve is to keep the contents of the caches coherent and the view of memory provided to the processors consistent. A bus is a powerful mechanism. It provides any-to-any communication through a single set of wires; moreover, it can serve as a broadcast medium, since there is only one set of wires, and even provide global status via wired-OR signals. The properties of bus transactions are exploited in designing extensions of conventional cache controllers that solve the coherence problem. Chapter 5 presents the fundamental techniques for bus-based cache coherence at the logical level and presents the basic design alternatives. These design alternatives provide an illustration of how workload-driven evaluation can be brought to bear in making design decisions. Finally, Chapter 5 examines the parallel programming issues of the earlier chapters in terms of the aspects of machine design that influence software level, especially with regard to cache effects on sharing patterns and the design of robust synchronization routines. Chapter 6 focuses on the organizational structure and machine implementation of bus-based cache coherence. It examines a variety of more advanced designs that seek to reduce latency and increase bandwidth while preserving a consistent view of memory.

Chapters 7 through 11 form a closely interlocking study of the design of scalable parallel architectures. Chapter 7 makes the conceptual step from a bus transaction as a building block for higher-level abstractions to a network transaction as a building block. To cement this understanding, the communication abstractions that we have

surveyed in this introductory chapter are constructed from primitive network transactions. Then the chapter studies the design of the node-to-network interface in depth using a spectrum of case studies.

Chapters 8 and 9 go deeply into the design of scalable machines supporting a shared address space, both a shared physical address space and a shared virtual address space upon independent physical address spaces. The central issue is automatic replication of data while preserving a consistent view of memory and avoiding performance bottlenecks. The study of a global physical address space emphasizes hardware organizations that provide efficient, fine-grained sharing. The study of a global virtual address space provides an understanding of a minimal degree of hardware support required for most workloads.

Chapter 10 takes up the question of the design of the scalable network itself. As with processors, caches, and memory systems, the network design space has several dimensions, and often a design decision involves interactions along these dimensions. The chapter lays out the fundamental design issues for scalable interconnects, illustrates the common design choices, and evaluates them relative to the requirements established in Chapters 8 and 9. Chapter 11 draws together the material from the previous four chapters in the context of techniques for latency tolerance, including bulk transfer, write behind, and read ahead across the spectrum of communication abstractions. Finally, Chapter 12 looks at the overall concepts of the book in light of technological, application, and economic trends and forecasts the key ongoing developments in the field of parallel computer architecture.

1.5

HISTORICAL REFERENCES

Parallel computer architecture has a long, rich, and varied history that is deeply interwoven with advances in the underlying processor, memory, and network technologies. The first blossoming of parallel architectures occurs around 1960. This is a point where transistors have replaced tubes and other complicated and constraining logic technologies. Processors are smaller and more manageable. A relatively cheap, inexpensive storage technology exists (core memory), and computer architectures are settling down into meaningful “families.”

Small-scale shared memory multiprocessors took on an important commercial role at this point with the inception of what we call mainframes today, including the Burroughs B5000 (Lonergan and King 1961) and D825 (Anderson et al. 1962) and the IBM System 360 models 65 and 67 (Padegs 1981). Support for multiprocessor configurations was one of the key extensions in the evolution of the 360 architecture to System 370. These included atomic memory operations and interprocessor interrupts. In the scientific computing area, shared memory multiprocessors were also common. The CDC 6600 provided an asymmetric shared memory organization to connect multiple peripheral processors with the central processor, and a dual CPU configuration of this machine was produced. The origins of message-passing machines come about in the RW400, introduced in 1960 (Porter 1960). Data parallel machines also emerged, with the design of the Solomon computer (Ball et al. 1962; Slotnick, Borck, and McReynolds 1962).

Through the late 1960s, tremendous innovation occurred in the use of parallelism within the processor using pipelining and replication of function units to obtain a far greater range of performance within a family than could be obtained by simply increasing the clock rate. It was argued that these efforts were reaching a point of diminishing returns, so the University of Illinois and Burroughs undertook a major research project to design and build a 64-processor SIMD machine, called Illiac IV (Bouknight et al. 1972), based on the earlier Solomon work (and in spite of Amdahl's arguments to the contrary [Amdahl 1967]). This project was very ambitious, involving research in the basic hardware technologies, architecture, I/O devices, operating systems, programming languages, and applications. By the time a scaled-down, 16-processor system was working in 1975, the computer industry had undergone massive structural change.

First, the concept of storage as a simple linear array of moderately slow physical devices had been revolutionized, beginning with the idea of virtual memory and then with the concept of caching. Work on Multics and its predecessors (e.g., Atlas and CTSS) separated the concept of the user address space from the physical memory of the machine. This required maintaining a short list of recent translations, a translation lookaside buffer (TLB), in order to obtain reasonable performance. Maurice Wilkes, the designer of EDSAC, saw this as a powerful technique for organizing the addressable storage itself, giving rise to what we now call the cache. This proved an interesting example of locality triumphing over parallelism. The introduction of caches into the 360/85 yielded higher performance than the 360/91, which had a faster clock rate, faster memory, and elaborate pipelined instruction execution with dynamic scheduling. The use of caches was commercialized in the IBM 360/185, but this raised a serious difficulty for the I/O controllers as well as the additional processors. If addresses were cached and therefore not bound to a particular memory location, how was an access from another processor or controller to locate the valid data? One solution was to maintain a directory of the location of each cache line, an idea that has regained importance in recent years.

Second, storage technology itself underwent a revolution with semiconductor memories replacing core memories. Initially, this technology was most applicable to small cache memories. Other machines, such as the CDC 7600, simply provided a separate, small, fast, explicitly addressed memory. Third, integrated circuits took hold. The combined result was that uniprocessor systems enjoyed a dramatic advance in performance, which mitigated much of the added value of parallelism in the Illiac IV system, with its inferior technological and architectural base. Pipelined vector processing in the CDC STAR-100 addressed the class of numerical computations that Illiac was intended to solve but eliminated the difficult data movement operations. The final straw was the introduction of the CRAY-1 system, with an astounding 80-MHz clock rate owing to exquisite circuit design and the use of what we now call a RISC instruction set, augmented with vector operations using vector registers and offering high peak rate with very low start-up cost. The use of simple vector processing coupled with fast, expensive ECL circuits was to dominate high-performance computing for the next 15 years.

A fourth dramatic change occurred in the early 1970s, however, with the introduction of microprocessors. Although the performance of the early microprocessors was quite low, the improvements were dramatic as bit-slice designs gave way to 4-bit, 8-bit, 16-bit, and full-word designs. The potential of this technology motivated a major research effort at Carnegie-Mellon University to design a large shared memory multiprocessor using the LSI-11 version of the popular PDP-11 minicomputer. This project went through two phases. The first, called C.mmp, connected 16 processors through a specially designed circuit-switched crossbar to a collection of memories and I/O devices, much like the dancehall design in Figure 1.15 (Wulf, Levin, and Person 1975). The second, CM*, sought to build a 100-processor system by connecting 14-node clusters with local memory through a packet-switched network in a NUMA configuration (Swan, Fuller, and Siewiorek 1977; Swan et al. 1977), as in Figure 1.19.

This trend toward systems constructed from many small microprocessors literally exploded in the early to mid-1980s, resulting in the emergence of several disparate factions. On the shared memory side, it was observed that a confluence of caches and the properties of buses made modest multiprocessors very attractive. Buses have limited bandwidth but are a broadcast medium. Caches filter bandwidth and provide an intermediary between the processor and the memory system. Research at the University of California, Berkeley and elsewhere (Goodman 1983; Hill et al. 1986) introduced extensions of the basic bus protocol that allowed the caches to maintain a consistent state. This direction was picked up by several small companies, including Synapse (Nestle and Inselberg 1985), Sequent (Rodgers 1985), Encore (Bell 1985; Schanin 1986), Flex (Matelan 1985), and others, as the 32-bit microprocessor made its debut and the vast personal computer industry took off. A decade later, this general approach dominated the server and high-end workstation market and took hold in the PC servers and the desktop. The approach experienced a temporary setback as very fast RISC microprocessors took away the performance edge of multiple slower processors. Although the RISC micros were well suited to multiprocessor design, their bandwidth demands severely limited scaling until a new generation of shared bus designs emerged in the early 1990s.

Simultaneously, the message-passing direction took off with two major research efforts. At CalTech, a project was started to construct a 64-processor system using i8086/8087 microprocessors assembled in a hypercube configuration (Seitz 1985; Athas and Seitz 1988). From this baseline, several other designs were pursued at CalTech and JPL (Fox et al. 1988), and at least two companies pushed the approach into commercialization—Intel, with the iPSC series, and Ametek. A somewhat more aggressive approach was widely promoted by the INMOS Corporation in England in the form of the Transputer, which integrated four communication channels directly onto the microprocessor. This approach was adopted by nCUBE, with a series of very large-scale message-passing machines. Intel carried the commodity processor approach forward, replacing the i80386 with the faster i860, then replacing the network with a fast grid-based interconnect in the Delta and adding dedicated message processors in the Paragon. Meiko moved away from the Transputer to the i860 in

their computing surface. IBM also investigated an i860-based design in Vulcan before obtaining commercial success with the SP family, essentially a cluster of RS6000 workstations.

Data parallel systems also took off in the early 1980s, after a period of relative quiet. These included Batcher's MPP system for image processing developed by Goodyear and the Connection Machine promoted by Hillis for AI applications (Hillis 1985). The key enhancement was the provision of a general-purpose interconnect for problems demanding other than simple grid-based communication. These ideas saw commercialization with the emergence of Thinking Machines Corporation, first with the CM-1, which was close to Hillis's original conceptions, and then with the CM-2, which incorporated a large number of bit-parallel floating-point units. In addition, MasPar and Wavetracer carried the bit-serial or slightly wider organization forward in cost-effective systems.

A more formal development of highly regular parallel systems emerged in the early 1980s as systolic arrays, generally under the assumption that a large number of very simple processing elements would fit on a single chip. It was envisioned that these arrays would provide cheap, high-performance, special-purpose add-ons to conventional computer systems. To some extent, these ideas have been employed in programming data parallel machines. The iWARP project at CMU produced a more general, smaller-scale building block that has been developed further in conjunction with Intel. These ideas have also found their way into fast graphics, compression, and rendering chips.

The technological possibilities of the VLSI revolution also prompted the investigation of more radical architectural concepts, including dataflow architectures (Dennis 1980; Gurd, Kerkham, and Watson 1985; Papadopoulos and Culler 1990; Arvind and Culler 1986), which integrated the network very closely with the instruction scheduling mechanism of the processor. It was argued that very fast dynamic scheduling throughout the machine would hide the long communication latency and synchronization costs of a large machine and thereby vastly simplify programming. The evolution of these ideas tended to converge with the evolution of message-passing architectures in the form of message-driven computation (Dally, Keen, and Noakes 1993).

Large-scale shared memory designs took off as well. IBM pursued a high-profile research effort with the RP-3 (Pfister et al. 1985), which sought to connect a large number of early RISC processors (the 801) through a butterfly network. This was based on the NYU Ultracomputer work (Gottlieb et al. 1983), which was particularly novel for its use of combining operations. BBN developed two large-scale designs, the BBN Butterfly using Motorola 68000 processors and the TC2000 (Bolt Beranek and Newman 1989) using the 88100s. These efforts prompted a very broad investigation of the possibility of providing cache-coherent shared memory in a scalable setting. The DASH project at Stanford University sought to provide a fully cache-coherent distributed shared memory by maintaining a directory containing the disposition of every cache block (Lenoski et al. 1993; Lenoski et al. 1992). SCI represented an effort to standardize an interconnect and cache coherence protocol

(IEEE 1993). The Alewife project at MIT sought to minimize the hardware support for shared memory (Agarwal et al. 1995), which was pushed further by researchers at the University of Wisconsin (Wood et al. 1993). The Kendall Square Research KSR1 (Frank, Burkhardt, and Rothnie 1993; Saavedra, Gains, and Carlton 1993) went even further and allowed the home location of data in memory to migrate. Alternatively, the Denelcor HEP attempted to hide the cost of remote memory latency by interleaving many independent threads on each processor.

The 1990s have exhibited the beginnings of a dramatic convergence among these various factions. This convergence is driven by many factors. One is clearly that all of the approaches have common requirements. They all require a fast, high-quality interconnect. They all profit from avoiding latency where possible and reducing the absolute latency when it does occur. They all benefit from hiding as much of the communication cost as possible. They all must support various forms of synchronization. We have seen the shared memory work explicitly seek to better integrate message passing in Alewife (Agarwal et al. 1995) and FLASH (Kuskin et al. 1994) to obtain better performance where the regularity of the application can provide large transfers. We have seen data parallel designs incorporate complete commodity processors in the CM-5 (Leiserson et al. 1996), allowing very simple processing of messages at the user level, which provides much better efficiency for message-driven computing and shared memory (von Eicken et al. 1992; Spertus et al. 1993). There remains the additional support for fast global synchronization. We have seen fast global synchronization, message queues, and latency-hiding techniques developed in a NUMA shared memory context in the CRAY T3D (Kessler and Schwarzmeier 1993; Koeninger, Furtney, and Walker 1994), and the message-passing support in the Meiko CS-2 (Barton, Crownie, and McLaren 1994; Homewood and McLaren 1993) provides direct virtual-memory-to-virtual-memory transfers within the user address space. The new element that continues to separate the factions is the use of complete commodity workstation nodes, as in the SP-1, SP-2, and various workstation clusters using merging high-bandwidth networks (Anderson, Culler, and Patterson 1995; Kung et al. 1989; Pfister 1995). The costs of weaker integration into the memory system, imperfect network reliability, and general-purpose system requirements have tended to keep these systems more closely aligned with traditional message passing, although the future developments are far from clear.

1.6 EXERCISES

- 1.1 Compute the annual growth rate in number of transistors, die size, and clock rate by fitting an exponential to the technology leaders using the data in Table 1.1. Obtain more recent data from the Web, and see how well these trends have held.
- 1.2 Compute the annual performance growth rates for each of the benchmarks shown in Table 1.2. Comment on the differences that you observe.

Table 1.1 Basic Parameters for Several Microprocessors

Name	Year	Die (mm ²)	Total Transistors	Clock (MHz)
i4004	1971	9	2,300	0.5
i8008	1972	12.25	3,500	0.8
i8080	1974	20.25	5,000	3
M6800	1974	25	5,000	1
M68000	1979	43.56	68,000	12.5
i80286	1982	64	130,000	10
M68020	1984	84.64	180,000	25
i80386	1985	90.25	275,000	16
i80486	1988	160	1,200,000	50
MIPS R3000	1988	72	125,000	33
Motorola 68040	1989	126.4	1,200,000	25
Alpha 21064	1992	233.5	1,680,000	160
Pentium 66	1993	294	3,100,000	66.7
Alpha 21066	1994	209	1,750,000	133
MIPS R10000	1994	298	5,900,000	200
Alpha 21164	1995	298.7	9,300,000	300
UltracSparc	1995	315	3,800,000	167

Table 1.2 Performance of Leading Workstations

Machine	Year	SpecInt	SpecFP	LINPACK	n = 1,000	Peak FP
Sun 4/260	1987	9	6	1.1	1.1	3.3
MIPS M/120	1988	13	10.2	2.1	4.8	6.7
MIPS M/2000	1989	18	21	3.9	7.9	10
IBM RS6000/540	1990	24	44	19	50	60
HP 9000/750	1991	51	101	24	47	66
DEC Alpha AXP	1992	80	180	30	107	150
DEC 7000/610	1993	132.6	200.1	44	156	200
AlphaServer 2100	1994	200	291	43	129	190

- 1.3 Generally in evaluating performance trade-offs, we evaluate the improvement in performance, or speedup, due to some enhancement. Formally,

$$\text{Speedup due to enhancement } E = \frac{\text{Time}_{\text{without } E}}{\text{Time}_{\text{with } E}} = \frac{\text{Performance}_{\text{with } E}}{\text{Performance}_{\text{without } E}}$$

In particular, we will often refer to the speedup as a function of the machine parallel (e.g., the number of processors).

Suppose you are given a program that does a fixed amount of work, and some fraction s of that work must be done sequentially. The remaining portion of the work is perfectly parallelizable on P processors. Assuming T_1 is the time taken on one processor, derive a formula for T_p , the time taken on P processors. Use this to get a formula giving an upper bound on the potential speedup on P processors. (This is a variant of what is often called Amdahl's Law [Amdahl 1967].) Explain why it is an upper bound.

- 1.4 Given a histogram of available parallelism such as that shown in Figure 1.7, where f_i is the fraction of cycles on an ideal machine in which i instructions issue, derive a generalization of Amdahl's Law to estimate the potential speedup on a k -issue superscalar machine. Apply your formula to the histogram data in Figure 1.7 to produce the speedup curve shown in that figure.
- 1.5 Locate the current TPC performance data on the Web and compare the mix of system configurations, performance, and speedups obtained on those machines with the data presented in Figure 1.4.
- 1.6 In message-passing models, each process is provided with a special variable or function that gives its unique number or rank among the set of processes executing a program. Most shared memory programming systems provide a `fetch&inc` operation, which reads the value of a location and atomically increments the location. Write a little pseudocode to show how to use `fetch&add` to assign each process a unique number. Can you determine the number of processes comprising a shared memory parallel program in a similar way?
- 1.7 To move an n -byte message along H links in an unloaded store-and-forward network takes time $H \frac{n}{W} + (H - 1)R$, where W is the raw link bandwidth and R is the routing delay per hop. In a network with cut-through routing, this takes time $\frac{n}{W} + (H - 1)R$. Consider an 8×8 grid consisting of 40-MB/s links and routers with 250 ns of delay. What is the minimum, maximum, and average time to move a 64-byte message through the network? A 256-byte message?
- 1.8 Consider a simple 2D finite difference scheme where at each step every point in the matrix is updated by a weighted average of its four neighbors, $A[i, j] = A[i, j] - w(A[i - 1, j] + A[i + 1, j] + A[i, j - 1] + A[i, j + 1])$.

All the values are 64-bit floating-point numbers. Assuming one element per processor and $1,024 \times 1,024$ elements, how much data must be communicated per step? Explain how this computation could be mapped onto 64 processors so as to minimize the data traffic. Compute how much data must be communicated per step.

- 1.9 Consider the simple pipelined component described in Example 1.2. Suppose that the application alternates between bursts of m independent operations on the component and phases of computation lasting T ns that do not use the component. Develop an expression describing the execution time of the program based on these parameters. Compare this with the unpipelined and fully pipelined bounds. At what points do you get the maximum discrepancy between the models? How large is it as a fraction of overall execution time?
- 1.10 Show that Equation 1.4 follows from Equation 1.3.
- 1.11 What is the x -intercept of the line in Equation 1.3?
- 1.12 If we consider loading a cache line from memory, the transfer time is the time to actually transmit the data across the bus. The start-up includes the time to obtain access to the bus, convey the address, access the memory, and possibly place the data in the cache before responding to the processor. However, in a modern processor with dynamic instruction scheduling, the overhead may include only the portion spent accessing the cache to detect the miss and placing the request on the bus. The memory access portion contributes to latency, which can potentially be hidden by the overlap with execution of instructions that do not depend on the result of the load.

Suppose we have a machine with a 64-bit-wide bus running at 40 MHz. It takes two bus cycles to arbitrate for the bus and present the address. The cache line size is 32 bytes and the memory access time is 100 ns. What is the latency for a read miss? What bandwidth is obtained on this transfer?

- 1.13 Suppose this 32-byte line is transferred to another processor and the communication architecture imposes a start-up cost of 2 μ s and a data transfer bandwidth of 20 MB/s. What is the total latency of the remote operation?
- 1.14 If we consider sending an n -byte message to another processor, we may use the same model as in Exercise 1.12. The start-up can be thought of as the time for a zero-length message; it includes the software overhead on the two processors, the cost of accessing the network interface, and the time to actually cross the network. The transfer time is usually determined by the point along the path with the least bandwidth, that is, the bottleneck.

Suppose we have a machine with a message start-up of 100 μ s and an asymptotic peak bandwidth of 80 MB/s. At what size message is half of the peak bandwidth obtained?

- 1.15 In some cases, Equation 1.6 can be used for estimating data transfer performance based on design parameters. In other cases, it serves as an empirical tool for fitting measurements to a line to determine the effective start-up and peak bandwidth of a portion of a system. If data undergoes a series of copies as part of a transfer (assuming that before transmitting a message the data must be copied into a buffer), the basic message time is as in Exercise 1.14, but the copy is performed at a cost of 5 cycles per 32-bit word on a 100-MHz machine. Given an equation for the expected user-level message time, how does the cost of a copy compare with a fixed cost of, say, entering the operating system?

- 1.16 Consider a machine running at 100 MIPS on some workload with the following mix: 50% ALU, 20% loads, 10% stores, and 20% branches. Suppose the instruction miss rate is 1%, the data miss rate is 5%, and the cache line size is 32 bytes. For the purpose of this calculation, treat a store miss as requiring two cache line transfers, one to load the newly updated line and one to replace the dirty line. If the machine provides a 250-MB/s bus, how many processors can it accommodate at 50% of peak bus bandwidth? What is the bandwidth demand of each processor?
- 1.17 Exercise 1.16 looks only at the sum of the average bandwidths, leaving 50% headroom on the bus to make the calculation reasonable. As the bus approaches saturation, however, it takes longer to obtain access for the bus, so it looks to the processor as if the memory system is slower. The effect is to slow down all of the processors in the system, thereby reducing their bandwidth demand. Let's try an analogous calculation from the other direction.
Assume the instruction mix and miss rate as in Exercise 1.16, but ignore the MIPS since that depends on the performance of the memory system. Assume instead that the processor runs at 100 MHz and has an ideal CPI (with a perfect memory system) of one. The unloaded cache miss penalty is 20 cycles. You can ignore the write back for stores. (As a starter, you might want to compute the MIPS rate for this new machine.) Assume that the memory system (i.e., the bus and the memory controller) is utilized throughout the miss. What is the utilization of the memory system U_1 with a single processor? From this result, estimate the number of processors that could be supported before the processor demand would exceed the available bus bandwidth.
- 1.18 Of course, no matter how many processors you place on the bus, they will never exceed the available bandwidth. Explain what happens to processor performance in response to bus contention. Can you formalize your observations?



Parallel Programs

To understand and evaluate design decisions in a parallel machine, we must have an idea of the software that runs on the machine. Understanding program behavior has led to some of the most important advances in uniprocessors, including memory hierarchies and instruction set design. It is all the more important in multiprocessors, both because of the increase in degrees of freedom and because of the much greater performance penalties caused by mismatches between applications and systems.

Understanding parallel software is important for algorithm designers, for programmers, and for architects. As algorithm designers, it helps us focus on designing algorithms that can be run effectively in parallel on real systems. As programmers, it helps us understand the key performance issues and obtain the best performance from a system. And as architects, it helps us understand the workloads we are designing against and their important degrees of freedom. Parallel software and its implications will be the focus of the next three chapters of this book. This chapter describes the process of creating parallel programs in the major programming models. Chapter 3 focuses on the performance issues that must be addressed in this process, exploring some of the key interactions between parallel applications and architectures. Chapter 4 relies on this understanding of hardware/software interactions to develop guidelines for using parallel workloads to evaluate architectural trade-offs. In addition to being helpful to architects, the material in these chapters is useful for users of parallel machines as well: Chapters 2 and 3 for programmers and algorithm designers, and Chapter 4 for users making decisions about what types of machines to procure. However, the major focus is on issues that architects should understand before they get into the nuts and bolts of machine design.

As architects of sequential machines, we generally take programs for granted: the field is mature, and there is a large base of programs that can (or must) be viewed as fixed. We optimize the machine design against the requirements of these programs. Although we recognize that programmers may further optimize their code—for example, as caches become larger or floating-point support is improved—we usually evaluate new designs without anticipating such software changes. Compilers may evolve along with the architecture, but the source program is still treated as fixed. In parallel architecture, there is a much stronger and more dynamic interaction between the evolution of machine designs and that of parallel software. Since parallel computing is all about performance, programming tends to be oriented toward taking advantage of what machines provide. Parallelism offers a new degree of

freedom—the number of processors—and higher costs for data access and coordination, giving the programmer a wide scope for software optimizations. Even as architects, we therefore need to open the application “black box.” Understanding the important aspects of the process of creating parallel software (the focus of this chapter) helps us appreciate the role and limitations of the architecture. A deeper look at performance issues in the next chapter will shed greater light on hardware/software trade-offs.

Even after a problem and a good sequential algorithm to solve it are determined, a substantial process is involved in arriving at a parallel program and the execution characteristics that it offers to a multiprocessor architecture. This chapter presents general principles of the parallelization process and illustrates them with real examples. It begins by introducing four actual problems that serve as case studies throughout the next two chapters. Then it describes the four major steps in creating a parallel program—using the case studies to illustrate—followed by examples of how a simple parallel program might be written in each of the major programming models. As discussed in Chapter 1, the dominant models from a programming perspective narrow down to three: the data parallel model, a shared address space or shared memory, and message passing between private address spaces. This chapter illustrates the primitives provided by these models and how they might be used, without much concern for performance. After the performance issues in the parallelization process are understood in Chapter 3, the four application case studies will be treated in more detail to create high-performance versions of them.

2.1 PARALLEL APPLICATION CASE STUDIES

We saw in the previous chapter that multiprocessors are used for a wide range of applications—from multiprogramming and commercial computing to so-called Grand Challenge scientific problems—and that the most demanding of these applications tend to be from scientific and engineering computing. Of the four case studies referred to throughout this chapter and the next, two are from scientific computing, one is from computer graphics, and one is from commercial computing. Besides being from different application domains, the case studies are chosen to represent a range of important behaviors found in other parallel programs as well.

The first case study simulates the motion of ocean currents by discretizing the problem on a set of regular grids and solving a system of equations on the grids. This technique is very common in scientific computing and leads to a set of very common communication patterns. The second case study represents another important form of scientific computing, in which, rather than discretizing the domain on a grid, the computational domain is represented as a large number of bodies that interact with one another and move around as a result of these interactions. These so-called n -body problems are common in many areas, such as simulating galaxies in astrophysics (our specific case study), simulating proteins and other molecules in chemistry and biology, and simulating electromagnetic interactions. As in many other areas, hierarchical algorithms for solving these problems have become very popular. Hierarchical n -body algorithms, such as the one in our case study, have also been used to solve

important problems in computer graphics and some particularly difficult types of equation systems. Unlike the first case study, this one leads to irregular, long-range, and unpredictable communication.

The third case study is from computer graphics, a very important consumer of moderate-scale multiprocessors. It traverses a three-dimensional scene with highly irregular and unpredictable access patterns and renders it into a two-dimensional image for display. The first three case studies are part of a benchmark suite (Singh, Weber, and Gupta 1992) that is widely used in architectural evaluations in the literature, so a wealth of detailed information is available about them. They will be used to illustrate architectural trade-offs in this book as well.

The last case study represents an increasingly important class of commercial applications that analyze the huge volumes of data being produced by our information society to discover useful knowledge, categories, and trends. These information processing applications tend to be I/O intensive, so parallelizing the I/O activity effectively is very important.

2.1.1

Simulating Ocean Currents

To model the climate of the earth, it is important to understand how the atmosphere interacts with the oceans that occupy three-fourths of the earth's surface. This case study simulates the motion of water currents in the ocean. These currents develop and evolve under the influence of several physical forces, including atmospheric effects, wind, and friction with the ocean floor. Near the ocean walls, additional "vertical" friction is present as well, which leads to the development of eddy currents. The goal of this particular application case study is to simulate these eddy currents over time and understand their interactions with the mean ocean flow.

Good models for ocean behavior are complicated: predicting the state of the ocean at any instant requires the solution of complex systems of equations, which can only be performed numerically by computer. We are, additionally, interested in the behavior of the currents over time. The actual physical problem is continuous in both space (the ocean basin) and time, but to enable computer simulation we discretize it along both dimensions. To discretize space, we model the ocean basin as a grid of points. Every important variable—such as pressure, velocity, and various currents—has a value at each grid point in this discretization. This particular application uses not a three-dimensional grid but a set of two-dimensional, horizontal cross sections through the ocean basin, each represented by a two-dimensional grid of points (see Figure 2.1). For simplicity, the ocean is modeled as a rectangular basin and the grid points are assumed to be equally spaced. Each of the many variables is therefore represented by a separate two-dimensional array for each cross section through the ocean. For the time dimension, we discretize time into a series of finite time-steps. The equations of motion are solved at all the grid points in one time-step, the state of the variables is updated as a result, the equations of motion are solved again for the next time-step, and so on repeatedly.

Every time-step itself consists of several computational phases. Many of these are used to set up values for the different variables at all the grid points using the results

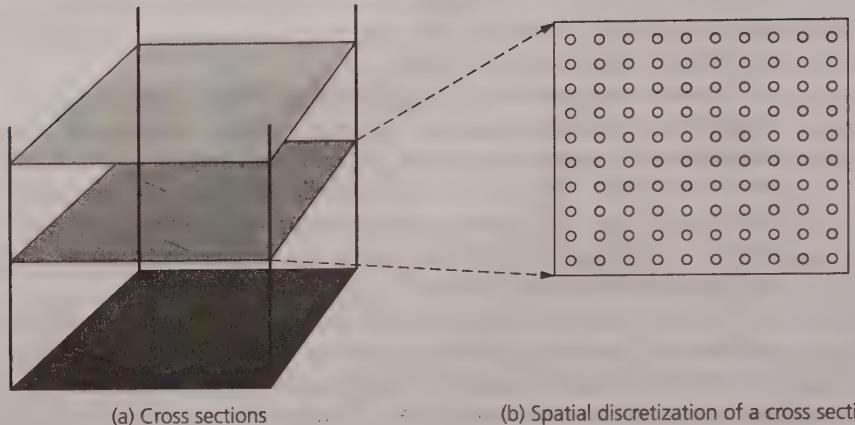


FIGURE 2.1 Horizontal cross sections through an ocean basin and their spatial discretization into regular grids

from the previous time-step. In other phases, the system of equations for a time-step is actually solved. All the phases, including the solver, involve sweeping through all points of the relevant arrays and manipulating their values. The solver phases are a little more complex, as we shall see when we discuss this case study in more detail in Chapter 3.

The more grid points we use in each dimension to represent a fixed-size ocean, the finer the spatial resolution of our discretization and the more accurate our simulation. For an ocean such as the Atlantic, with its roughly $2,000 \text{ km} \times 2,000 \text{ km}$ span, using a grid of 100×100 points implies a distance of 20 km between points in each dimension. This is not a very fine resolution, so we would like to use many more grid points. Similarly, shorter physical intervals between time-steps lead to greater simulation accuracy. For example, to simulate 5 years of ocean movement by updating the state every 8 hours, we would need about 5,500 time-steps. The computational demands for high accuracy are large, and the need for multiprocessing is clear.

Fortunately, the application naturally affords a lot of concurrency: many of the setup phases in a time-step are independent of one another and therefore can be done in parallel, and the processing of different grid points in each phase or grid computation can itself be done in parallel. For example, we might assign different parts of each ocean cross section to different processors and have the processors perform each phase of computation on their assigned parts of the cross section grids (a data parallel formulation).

2.1.2 Simulating the Evolution of Galaxies

The second case study is also from scientific computing. It seeks to understand the evolution of stars in a system of galaxies over time. For example, we may want to

study what happens when galaxies collide or how a random collection of stars folds into a defined galactic shape. This problem involves simulating the motion of a number of bodies (here stars) moving under forces exerted on each by all the others, an n -body problem. The computation is discretized in space by treating each star as a separate body or by sampling to use one body to represent many stars. Here again we discretize the computation in time and simulate the motion of the galaxies for many time-steps. In each time-step, we compute the gravitational forces exerted on each star by all the others and update the position, velocity, and other attributes of that star.

Computing the forces among stars is the most expensive part of a time-step. A simple method to compute forces is to calculate pairwise interactions among all stars. This has $O(n^2)$ computational complexity for n stars and is therefore prohibitive for the millions of stars that we would like to simulate. However, by taking advantage of insights into the force laws, smarter hierarchical algorithms are able to reduce the complexity to $O(n \log n)$. This makes it feasible to simulate problems with millions of stars in a reasonable time but only by using powerful multiprocessors. The hierarchical algorithms use the basic insight that, since the strength of the gravitational interaction falls off with distance as

$$G \frac{m_1 m_2}{r^2}$$

the influences of stars that are farther away are weaker and therefore do not need to be computed as accurately as those of stars that are close by. Thus, if a group of stars is sufficiently far from a given star, we can compute the effect of the group on the star by approximating the group as a single star at the center of the group with little loss in accuracy (see Figure 2.2). The farther away the stars are from a given star, the larger the group that can be thus approximated. In fact, the strength of many physical interactions falls off with distance, so hierarchical methods are becoming increasingly popular in many areas of computing.

The particular hierarchical force calculation algorithm used in this case study is the Barnes-Hut algorithm. (The case study is called Barnes-Hut in the literature, and this name is used here as well.) We shall see how the algorithm works in Section 3.5.2. Since galaxies are denser in some regions and sparser in others, the distribution of stars in space is highly irregular. The distribution also changes with time as the galaxy evolves. The nature of the hierarchical approach implies that stars in denser regions interact more with other stars and centers of mass—and hence have more work associated with them—than stars in sparser regions. Ample concurrency exists across stars within a time-step, but given their irregular and dynamically changing nature, the challenge is to exploit concurrency efficiently on a parallel architecture.

2.1.3 Visualizing Complex Scenes Using Ray Tracing

The third case study is the visualization of complex scenes in computer graphics. A common technique used to render such scenes into images is known as *ray tracing*.

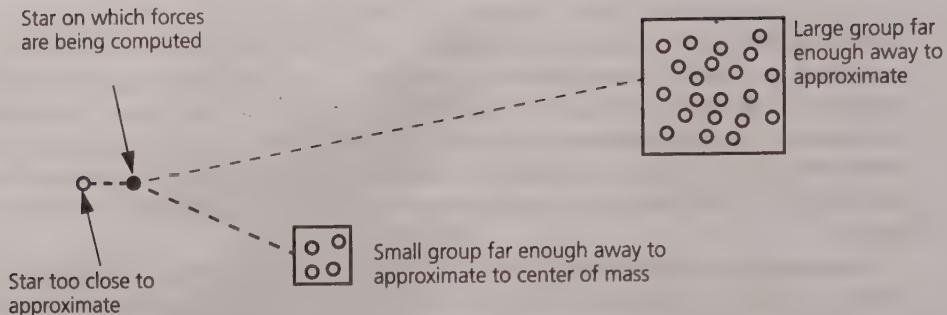


FIGURE 2.2 The insight used by hierarchical methods for *n*-body problems. A group of bodies that is far enough away from a given body may be approximated by the center of mass of the group. The farther apart the bodies, the larger the group that may be thus approximated.

The scene is represented as a set of objects in three-dimensional space, and the image being rendered is represented as a two-dimensional array of pixels (picture elements) whose color, opacity, and brightness values are to be computed. The pixels taken together represent the image, and the resolution of the image is determined by the distance between pixels in each dimension. The scene is rendered as seen from a specific viewpoint or position of the eye. Rays are shot from that viewpoint through every pixel in the image plane and into the scene. The algorithm traces the paths of these rays—computing their reflection, refraction, and lighting interactions as they strike and reflect off objects—and thus computes values for the color and brightness of the corresponding pixels. There is obvious parallelism across the rays shot through different pixels. This case study is referred to as Raytrace.

2.1.4 Mining Data for Associations

Information processing is rapidly becoming a major market for parallel systems. Businesses acquiring data about customers and products are devoting computational power to automatically extracting useful information or “knowledge” from this data. Examples from a customer database might include determining the buying patterns of demographic groups or segmenting customers according to relationships in their buying patterns. This process is called *data mining*. It differs from standard database queries in that its goal is to identify implicit trends and segmentations in data rather than simply look up the data requested by a direct, explicit query. For example, finding all customers who have bought cat food in the last week is not data mining; however, segmenting customers according to relationships in their age groups, their monthly incomes, and their preferences in cat food, in cars, and in kitchen utensils is.

A particular type of data mining is mining for associations. Here the goal is to discover relationships (associations) in the available information related to, say, differ-

ent customers and their transactions and to generate rules for the inference of customer behavior. For example, the database may store for every transaction the list of items purchased in that transaction. The goal of the mining may be to determine associations between sets of commonly purchased items that tend to be purchased together—for example, the conditional probability $P(S_1/S_2)$ that a certain set of items S_1 is found in a transaction given that a different set of items S_2 is found in that transaction, where S_1 and S_2 are sets of items that occur often in customer transactions. If this probability is high, then customers who have the set S_2 in their purchase transactions may be good advertising targets of items in set S_1 .

Consider the problem a little more concretely. We are given a database in which the records correspond to customer purchase transactions, as described above. Each transaction has a transaction identifier and a set of attributes, which in this case are the items purchased. The first goal in mining for associations is to examine the database and determine which sets of k items, say, are found to occur together in more than a certain threshold fraction of the transactions. A set of items (of any size) that occur together in a transaction is called an *itemset*, and an itemset that is found in more than this threshold fraction of transactions is called a *large itemset*. Once the large itemsets of size k are found, together with their frequencies of occurrence in the database, determining the association rules among them is quite easy. The problem we consider therefore focuses on discovering the large itemsets of size k and their frequencies. The database may be in main memory or more commonly on disk.

A simple way to solve the problem is to first determine the large itemsets of size one. From these, a set of candidate itemsets of size two items can be constructed—using the basic insight that an itemset can only be large if all its subsets are also large—and their frequency of occurrence in the transaction database can be counted. This results in a list of large itemsets of size two. The process is repeated until we obtain the large itemsets of size k . There is concurrency in examining large itemsets of size $k - 1$ to determine candidate itemsets of size k and in counting the number of transactions in the database that contain each of the candidate itemsets.

2.2

THE PARALLELIZATION PROCESS

The four case studies—Ocean, Barnes-Hut, Raytrace, and Data Mining—offer abundant concurrency and will help illustrate the process of creating effective parallel programs in this chapter and the next. For concreteness, we will assume that the sequential algorithm that we are to make parallel is given to us, perhaps as a description or as a sequential program. In many cases, as in these case studies, the best sequential algorithm for a problem lends itself easily to parallelization; in others, it may not afford enough parallelism, and a fundamentally different algorithm may be required. The rich field of parallel algorithm design is outside the scope of this book. However, whatever the chosen underlying sequential algorithm, a significant process of creating a good parallel program is present in all cases, and we must understand this process in order to program parallel machines effectively and evaluate architectures against parallel programs.

At a high level, the job of parallelization involves identifying the work that can be done in parallel, determining how to distribute the work and perhaps the data among the processing nodes, and managing the necessary data access, communication, and synchronization. Note that the work to be done includes computation, data access, and input/output activity. The goal is to obtain high performance while keeping programming effort and the resource requirements of the program low. In particular, we would like to obtain good speedup over the best sequential program that solves the same problem. This requires that we ensure a balanced distribution of work among processors, reduce the amount of interprocessor communication, which is expensive, and keep low the overhead of communication, synchronization, and parallelism management.

The steps in the process of creating a parallel program may be performed either by the programmer or by one of the many layers of system software that intervene between the programmer and the architecture. These layers include the compiler, the run-time system, and the operating system. In a perfect world, system software would allow users to write programs in the form they found most convenient (for example, as sequential programs in a high-level language or as an even higher-level specification of the problem) and would automatically perform the transformation into efficient parallel programs and executions. While much research is being conducted in parallelizing compiler technology and in programming languages, the goal of automatic parallelization is very ambitious and has not yet been fully achieved. In practice today, the vast majority of the process is still the responsibility of the programmer, with perhaps some help from the compiler and run-time system. Regardless of how the responsibility is divided among these parallelizing agents, the issues and trade-offs are similar, and it is important that we understand them. For concreteness, we shall assume for the most part that the programmer has to make all the decisions.

Let us now examine the parallelization process in a more structured way, by looking at the actual steps in it. Each step will address a subset of the issues needed to obtain good performance. These performance issues will be discussed in detail in Chapter 3 and only mentioned briefly here.

2.2.1 Steps in the Process

To understand the steps in creating a parallel program, let us first define three important concepts: tasks, processes, and processors. A *task* is an arbitrarily defined piece of the work done by the program. It is the smallest unit of concurrency that the parallel program can exploit; that is, an individual task is executed by only one processor, and concurrency among processors is exploited only across tasks. In the Ocean application, we can think of a single grid point in each phase of computation as being a task, or a row of grid points, or any arbitrary subset of a grid. We could even consider an entire grid computation to be a single task, in which case parallelism is exploited only across independent grid computations. In Barnes-Hut a task may be a body, in Raytrace a ray or a group of rays, and in Data Mining it may be

checking a single transaction for the occurrence of a particular itemset. What exactly constitutes a task is not prescribed by the underlying sequential program; it is a choice of the parallelizing agent, though it usually matches some natural granularity of work in the sequential program structure (for example, an iteration of a loop). If the amount of work a task performs is small, it is called a *fine-grained* task; otherwise, it is called *coarse-grained*.

A process (referred to interchangeably as a *thread* hereafter) is an abstract entity that performs tasks.¹ A parallel program is composed of multiple cooperating processes, each of which performs a subset of the tasks in the program. Tasks are assigned to processes by some assignment mechanism. For example, if the computation for each row in a grid in Ocean is viewed as a task, then a simple assignment mechanism may be to give an equal number of adjacent rows to each process, thus dividing the ocean cross section into as many horizontal slices as there are processes. In Data Mining, the assignment may be determined both by which portions of the database are assigned to each process and by how the candidate itemsets within a list are assigned to processes to look up the database. Processes may need to communicate and synchronize with one another to perform their assigned tasks. Finally, the way processes perform their assigned tasks is by executing them on the physical *processors* in the machine.

It is important to understand the difference between processes and processors from a parallelization perspective. While processors are physical resources, processes provide a convenient way of abstracting, or virtualizing, a multiprocessor: we initially write parallel programs in terms of processes, not physical processors; mapping processes to processors is a subsequent step. The number of processes does not have to be the same as the number of processors available to the program in a given execution. If there are more processes, they are multiplexed onto the available processors; if there are fewer processes, then some processors will remain idle.

Given these concepts, the job of creating a parallel program from a sequential one consists of four steps, illustrated in Figure 2.3:

1. *Decomposition* of the computation into tasks
2. *Assignment* of tasks to processes
3. *Orchestration* of the necessary data access, communication, and synchronization among processes
4. *Mapping* or binding of processes to processors

Together, decomposition and assignment are called *partitioning*, since they divide the work done by the program among the cooperating processes. Let us examine the steps and their individual goals a little further.

1. In Chapter 1 we used the correct operating systems definition of a process: an address space and one or more threads of control that share that address space. Thus, processes and threads are distinguished in that definition. To simplify our discussion of parallel programming in this chapter, we do not make this distinction but assume that a process has only one thread of control.

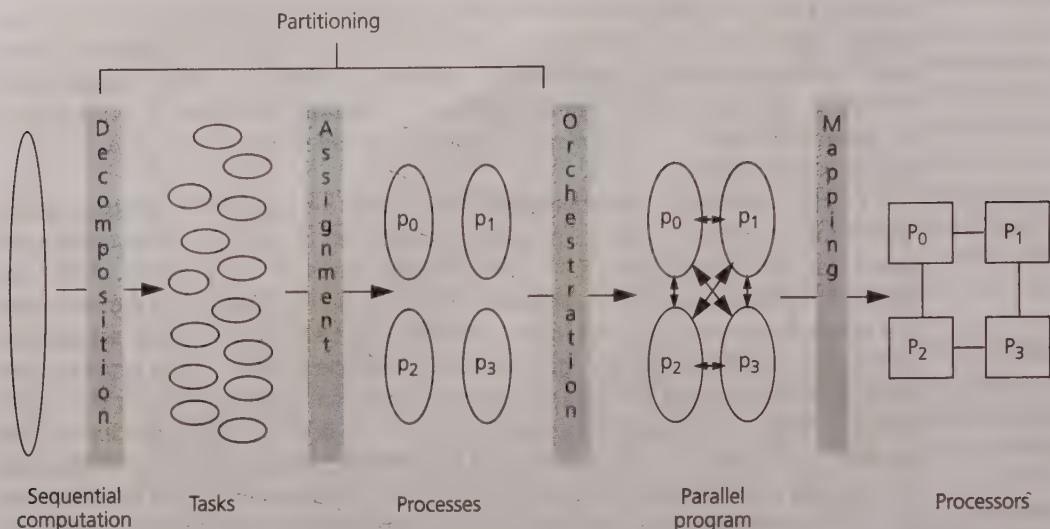


FIGURE 2.3 Steps in parallelization and the relationships among tasks, processes, and processors. The decomposition and assignment phases together are called *partitioning*. The orchestration phase coordinates data access, communication, and synchronization among processes (p), and the mapping phase maps them to physical processors (P).

Decomposition

Decomposition means breaking up the computation into a collection of tasks. In general, tasks may become available dynamically as the program executes, and the number of tasks available at a time may vary over the execution of the program. The maximum number of tasks available for execution at a time provides an upper bound on the number of processes (and hence processors) that can be used effectively at that time. Therefore, the major goal in decomposition is to expose enough concurrency to keep the processes busy at all times, yet not so much that the overhead of managing the tasks becomes substantial compared to the useful work done.

Limited concurrency is the most fundamental limitation on the speedup achievable through parallelism. It is not only the available concurrency in the underlying problem that matters but also how much of this concurrency is exposed in the decomposition. The impact of available concurrency is codified in one of the few “laws” of parallel computing, called Amdahl’s Law. If some portions of a program’s execution don’t have as much concurrency as the number of processors used, then some processors will have to be idle for those portions and speedup will be suboptimal. To see this in its simplest form, consider what happens if a fraction s of a program’s execution time on a uniprocessor is inherently sequential; that is, it cannot be parallelized. Even if the rest of the program is parallelized to run on a large number of processors in infinitesimal time, this sequential time will remain. The overall execution time of the parallel program will be at least s , normalized to a total

sequential time of 1, and the speedup limited to $1/s$. For example, if $s = 0.2$ (20% of the program's execution is sequential), the maximum speedup available is $1/0.2$, or 5, regardless of the number of processors used, even if we ignore all other sources of overhead. Example 2.1 provides a simple but more realistic example.

EXAMPLE 2.1 Consider an example program with two phases. In the first phase, a single operation is performed independently on all points of a two-dimensional n -by- n grid, as in Ocean. In the second phase, the sum of the n^2 grid point values is computed. If we have p processors, we can assign n^2/p points to each processor and complete the first phase in parallel in time n^2/p . In the second phase, each processor can add each of its assigned n^2/p values into a global sum variable. What is the problem with this assignment, and how can we expose more concurrency? Ignore the costs of data access and communication.

Answer The problem is that the accumulations into the global sum must be done one at a time, or *serialized*, to avoid corrupting the sum value by having two processors try to modify it simultaneously (see mutual exclusion in Section 2.3.5). Thus, the second phase is effectively serial and takes n^2 time regardless of p . The total time in parallel is $n^2/p + n^2$, compared to a sequential time of $2n^2$, so the speedup is at most

$$\frac{2n^2}{\frac{n^2}{p} + n^2}$$

or

$$\frac{2p}{p+1}$$

which is at best 2 even if a very large number of processors is used.

We can expose more concurrency by using a little trick. Instead of summing each value directly into the global sum and serializing all the summing, we divide the second phase into two phases. In the new second phase, a process sums its assigned values independently into a private sum. Then, in the third phase, processes sum their private sums into the global sum. The second phase is now fully parallel; the third phase is serialized as before, but there are only p operations in it, not n . The total parallel time is $n^2/p + n^2/p + p$, and the speedup is at best $p \times 2n^2/(2n^2 + p^2)$. If n is large relative to p , this speedup limit is almost linear in the number of processors used. Figure 2.4 illustrates the improvement and the impact of limited concurrency. ■

More generally, given a decomposition and a problem size, we can construct a *concurrency profile*, which depicts how many operations (or tasks) are available to be performed concurrently in the application at a given time. The concurrency profile is a function of the problem, the decomposition, and the problem size. However, it is independent of the number of processors, effectively assuming that an infinite number of processors is available. It is also independent of the assignment or orchestration. These concurrency profiles may be easy to provide analytically (as in Example 2.1 and as we shall see for matrix factorization in Exercise 3.8) or they may be quite irregular. For example, Figure 2.5 shows a concurrency profile of a parallel event-driven simulation for the synthesis of digital logic systems. The x-axis is time,

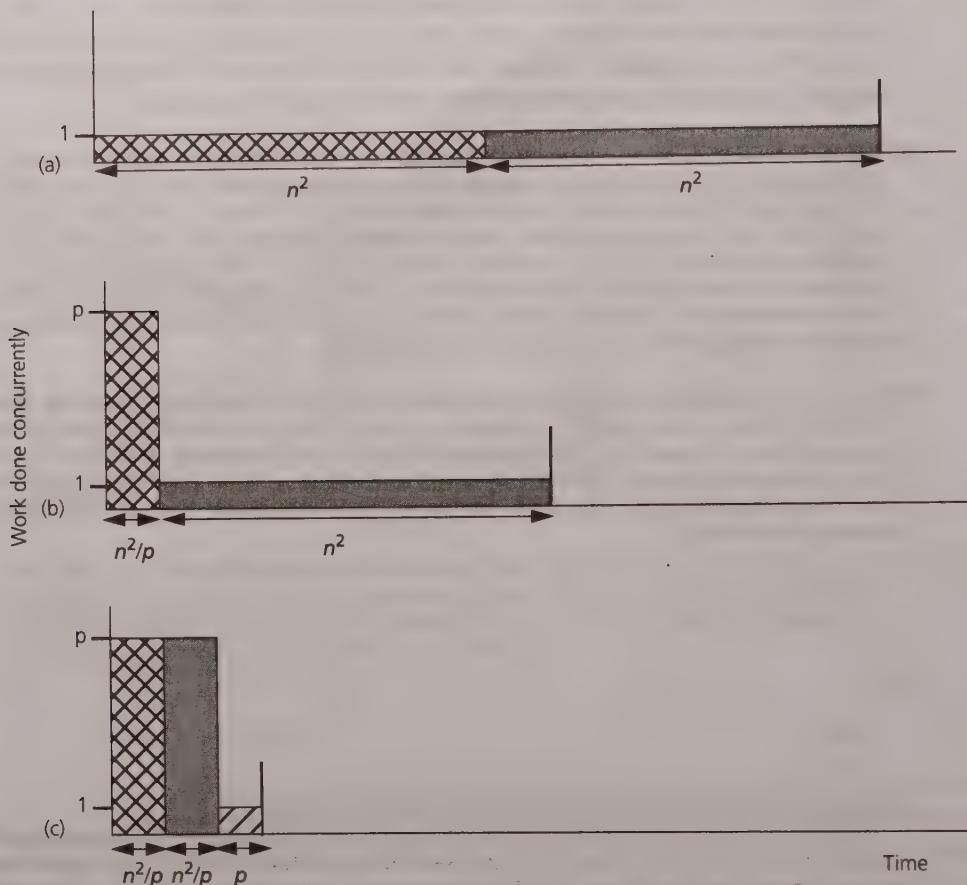


FIGURE 2.4 Illustration of the impact of limited concurrency: (a) one processor; (b) p processors, n^2 operations serialized; (c) p processors, p operations serialized. The x-axis is time, and the y-axis is the amount of work available (exposed by the decomposition) to be done in parallel at a given time. (a) shows the profile for a single processor. (b) shows the original case in the example, which is divided into two phases: one fully concurrent and one fully serialized. (c) shows the improved version, which is divided into three phases: the first two fully concurrent and the last fully serialized but with a lot less work in it ($O(p)$ rather than $O(n)$).

measured in clock cycles of the circuit being simulated. The y-axis or amount of concurrency is the number of logic gates in the circuit that are ready to be evaluated at a given time, which is a function of the circuit, the values of its inputs, and time. A wide range of unpredictable concurrency exists across clock cycles, with some cycles having almost no concurrency.

The area under the curve in the concurrency profile is the total amount of work done; that is, the number of operations or tasks computed or the “time” taken on a single processor. Its horizontal extent is a lower bound on the time that it would

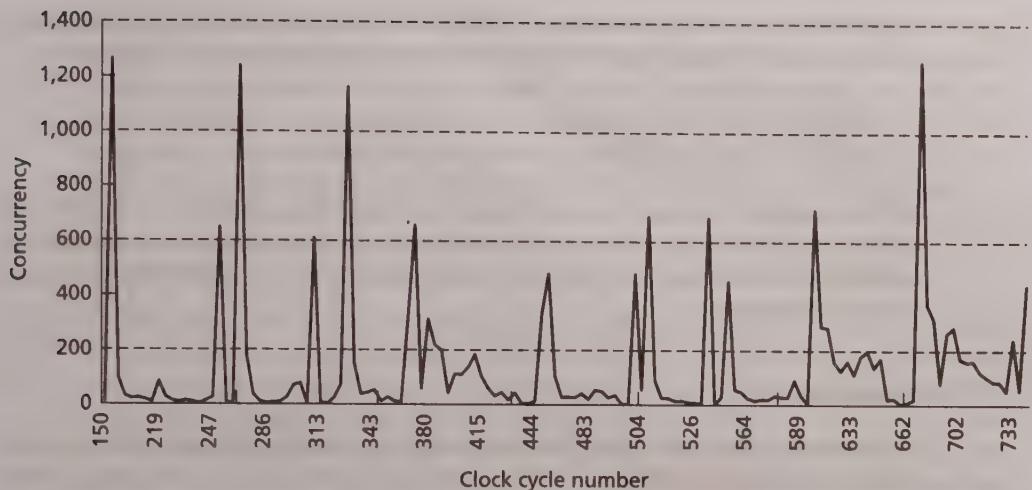


FIGURE 2.5 Concurrency profile for a distributed-time, discrete-event logic simulator. The circuit being simulated is a simple MIPS R6000 microprocessor. The y-axis shows the number of logic elements available for evaluation in a given simulated clock cycle.

take to run the best parallel program given that decomposition, assuming an infinitely large number of processors and that data access and communication are free. The area divided by the horizontal extent therefore gives us a limit on the achievable speedup with an unlimited number of processors, which is simply the average concurrency available in the application over time. A rewording of Amdahl's Law may therefore be

$$\text{Speedup} \leq \frac{\text{Area under Concurrency Profile}}{\text{Horizontal Extent of Concurrency Profile}}$$

For p processors, if f_k is the number of x-axis points in the concurrency profile that have concurrency k , then we can write Amdahl's Law as

$$\text{Speedup } (p) \leq \frac{\sum_{k=1}^{\infty} f_k k}{\sum_{k=1}^{\infty} f_k \left\lceil \frac{k}{p} \right\rceil} \quad (2.1)$$

It is easy to see that if the total work

$$\sum_{k=1}^{\infty} f_k k$$

is normalized to 1 and a fraction s of this is serial, then the speedup with an infinite number of processors is limited by $1/s$, and that with p processors it is limited by

$$\frac{1}{s + \frac{1-s}{p}}$$

In fact, Amdahl's Law can be applied to any overhead of parallelism (not just limited concurrency) that is not alleviated by using more processors. For now, Amdahl's Law quantifies the importance of exposing enough concurrency as a first step in creating a parallel program.

Assignment

Assignment means specifying the mechanism by which tasks will be distributed among processes. For example, which process is responsible for computing forces on which stars in Barnes-Hut? Which process will count occurrences of which item-sets, and in which parts of the database, in Data Mining?

The primary performance goals of assignment are to balance the workload among processes, to reduce the amount of interprocess communication, and to reduce the run-time overhead of managing the assignment. Balancing the workload is often referred to as *load balancing*. The workload to be balanced includes computation, input/output, and data access or communication; programs that are not balanced well among processes are said to be *load imbalanced*. Interprocess communication is expensive, especially when the processes run on different processors, and complex assignments of tasks to processes may incur overhead at run time.

Achieving these performance goals simultaneously can appear intimidating. However, most programs lend themselves to a fairly structured approach to partitioning (i.e., decomposition and assignment). For example, programs are often structured in phases, and candidate tasks for decomposition within a phase are often easily identified as seen in the case studies. The appropriate assignment of tasks is often discernible either by inspection of the code or from a higher-level understanding of the application. Where this is not so, well-known heuristic techniques are often applicable.

If the assignment is completely determined at the beginning of the program, or just after reading and analyzing the input, and does not change thereafter, it is called a *static* or *predetermined assignment*; if the assignment of work to processes is determined at run time as the program executes (perhaps to react to load imbalances), it is called a *dynamic assignment*. We shall see examples of both in Chapter 3. Note that this use of "static" is a little different from the compile-time meaning typically used in computer science. Compile-time assignment that does not change at run time would indeed be static, but the term is more general here.

Decomposition and assignment are the major algorithmic steps in parallelization. They are usually independent of the underlying architecture and programming model, although sometimes the cost and complexity of using certain primitives on a system can impact decomposition and assignment decisions. As architects, we

assume that the programs that will run on our machines are reasonably partitioned. There is nothing we can do if a computation is not parallel enough or not balanced across processes and little we may be able to do if it overwhelms the machine with communication. As programmers, we usually focus on decomposition and assignment first, independent of the programming model or architecture, though in some cases the properties of the latter may cause us to revisit our partitioning strategy.

Orchestration

Orchestration is the step in which the architecture and programming model, as well as the programming language itself, play a large role. To execute their assigned tasks, processes need mechanisms to name and access data, to exchange data (communicate) with other processes, and to synchronize with one another. Orchestration uses the available mechanisms to accomplish these goals correctly and efficiently. The choices made in orchestration are much more dependent on the programming model, and on the efficiencies with which the primitives of the programming model are supported, than the choices made in the previous steps. Some questions in orchestration include how to organize data structures, how to schedule the tasks assigned to a process temporally to exploit data locality, whether to communicate implicitly or explicitly and in small or large messages, and how exactly to organize and express the interprocess communication and synchronization that resulted from assignment. The programming language is important both because this is the step in which the program is actually written and because some of the trade-offs in orchestration are influenced strongly by available language mechanisms and their costs.

The major performance goals in orchestration are reducing the cost of the communication and synchronization as seen by the processors, preserving locality of data reference, scheduling tasks so that those on which many other tasks depend are completed early, and reducing the overhead of parallelism management. The job of architects is to provide the appropriate primitives with efficiencies that simplify successful orchestration. We shall discuss the major aspects of orchestration further when we see how programs are actually written.

Mapping

The cooperating processes that result from the decomposition, assignment, and orchestration steps constitute a full-fledged parallel program on modern systems. The program may choose to control the mapping of processes to processors, but if not, the operating system will take care of it, providing a parallel execution. Mapping tends to be fairly specific to the system or programming environment.

In the simplest case, the processors in the machine are partitioned into fixed subsets, possibly the entire machine, and only a single program runs at a time in a subset. This is called *space-sharing* of the machine. The program can bind, or *pin*, processes to processors to ensure that they do not migrate during the execution; it can even control exactly which processor a process runs on so as to preserve locality of communication in the network topology. Strict space-sharing schemes, together

with some simple mechanisms for time-sharing a subset among multiple applications, have so far been typical of large-scale multiprocessors. At the other extreme, the operating system may dynamically control which process runs where and when—without allowing the user any control over the mapping—to achieve better aggregate resource sharing and utilization. Each processor may employ the usual multiprogrammed scheduling criteria to manage processes from the same or different programs, and processes may be moved around among processors as the scheduler dictates. The operating system may extend the scheduling criteria to include multiprocessor-specific issues (for example, trying to have a process be scheduled on the same processor as much as possible so that the process can reuse its state in the processor cache and trying to schedule processes from the same application at the same time). In fact, most modern systems fall somewhere between these two extremes: the user may ask the system to preserve certain properties, giving the user program some control over the mapping, but the operating system is allowed to change the mapping dynamically for effective resource management.

Mapping and associated resource management issues in multiprogrammed systems are active areas of research. However, our goal here is to understand parallel programming in its basic form, so for simplicity we assume that a single parallel program has complete control over the resources of the machine. We also assume that the number of processes equals the number of processors and that neither changes during the execution of the program. By default, the operating system will place one process on every processor in no particular order. Processes are assumed not to migrate from one processor to another during execution. For this reason, the terms “process” and “processor” are used interchangeably in the rest of the chapter.

2.2.2 Parallelizing Computation versus Data

The view of the parallelization process described above has been centered on computation, or work, rather than on data. It is the computation that is decomposed and assigned. However, due to the programming model or performance considerations, we may be responsible for decomposing and assigning data to processes as well. In fact, in many important classes of problems, the decomposition of work and data are so strongly related that they are difficult or even unnecessary to distinguish. Ocean is a good example: each cross-sectional grid through the ocean is represented as an array, and we can view the parallelization as decomposing the data in each array and assigning parts of it to processes. The process that is assigned a portion of an array will then be responsible for the computation associated with that portion; this is known as an *owner computes* arrangement. A similar situation exists in Data Mining, where we can view the database as being decomposed and assigned; of course, here is also the question of assigning the itemsets to processes. Several language systems, including the high-performance Fortran standard (Koebel et al. 1994; High Performance Fortran Forum 1993), allow the programmer to specify the decomposition and assignment of data structures. The assignment of computation then follows the assignment of data in an owner computes manner. However, the distinction between

computation and data is stronger in many other more irregular applications, including the Barnes-Hut and Raytrace case studies, as we shall see. Since the computation-centric view is more general, we shall retain this view and consider data management to be part of the orchestration step.

2.2.3 Goals of the Parallelization Process

As stated previously, the major goal of using a parallel machine is to improve performance by obtaining speedup over the best uniprocessor execution. Each of the steps in creating a parallel program has a role to play in achieving this overall goal, and each step has its own subset of performance goals. These are summarized in Table 2.1; the next chapter discusses them in more detail.

Creating an effective parallel program requires evaluating cost as well as performance. In addition to the dollar cost of the machine itself, we must consider the resource requirements of the program on the architecture (for example, its memory usage) and the effort it takes to develop a satisfactory program. While costs and their impact are often more difficult to quantify than performance, they are very important, and we must not lose sight of them; in fact, we often decide to compromise performance to reduce them. As algorithm designers, we should favor high-performance solutions that keep the resource requirements of the algorithm small and that don't require inordinate programming effort. As architects, we should try to design high-performance systems that facilitate resource-efficient algorithms and reduce programming effort in addition to being low cost. For example, an architecture on which performance improves gradually with increased programming effort may be preferable to one that is capable of ultimately delivering better performance but requires inordinate programming effort to even achieve acceptable performance.

Table 2.1 Steps in the Parallelization Process and Their Goals

Step	Architecture-Dependent?	Major Performance Goals
Decomposition	Mostly no	Expose enough concurrency but not too much
Assignment	Mostly no	Balance workload Reduce communication volume
Orchestration	Yes	Reduce noninherent communication via data locality Reduce communication and synchronization cost as seen by the processor Reduce serialization at shared resources Schedule tasks to satisfy dependences early
Mapping	Yes	Put related processes on the same processor if necessary Exploit locality in network topology

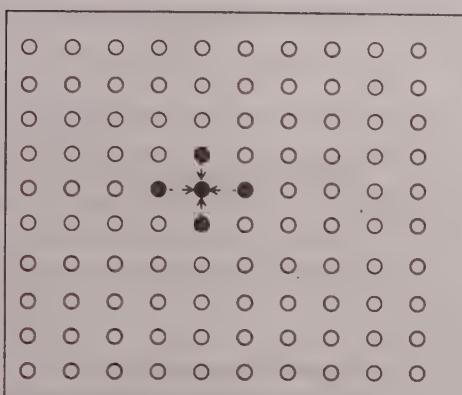
We can apply an understanding of the basic process and its goals to a simple but detailed example to see what the resulting parallel programs look like in the three major modern programming models introduced in Chapter 1: shared address space, message passing, and data parallel. Our focus will be on illustrating programs and programming primitives, not so much on performance, which is the subject of Chapter 3.

2.3 PARALLELIZATION OF AN EXAMPLE PROGRAM

The four case studies introduced at the beginning of the chapter all lead to parallel programs that are too complex and too long to serve as useful sample programs. Instead, this section presents a simplified version of a piece, or *kernel*, of Ocean: its equation solver. It uses the equation solver to dig deeper and to illustrate how to implement a parallel program using the three programming models. Except for the data parallel version, which necessarily uses a high-level data parallel language, the parallel programs are not written in an aesthetically pleasing language that relies on software layers to hide the orchestration and communication abstraction from the programmer. Rather, they are written in C or Pascal-like pseudocode augmented with simple extensions for parallelism, thus exposing the basic communication and synchronization primitives that a shared address space or message-passing communication abstraction must provide. Standard sequential languages augmented with primitives for parallelism also reflect the state of most real parallel programming today.

2.3.1 The Equation Solver Kernel

The equation solver kernel solves a simple partial differential equation on a grid, using what is referred to as a finite differencing method. It operates on a regular, two-dimensional grid or array of $(n + 2)$ -by- $(n + 2)$ elements, such as a single horizontal cross section of the ocean basin in Ocean. The border rows and columns of the grid contain boundary values that do not change, whereas the interior n -by- n points are updated by the solver, starting from their initial values. The computation proceeds over a number of sweeps. In each sweep, it operates on all the interior n -by- n points of the grid. For each point, it replaces its value with a weighted average of itself and its four nearest-neighbor points—above, below, left, and right (see Figure 2.6). The updates are done in place in the grid, so the update computation for a point sees the new values of the points above and to the left of it and the old values of the points below and to its right. This form of update is called the Gauss-Seidel method. During each sweep, the kernel also computes the average difference of an updated element from its previous value. If this average difference over all elements is smaller than a predefined “tolerance” parameter, the solution is said to have converged and the solver exits at the end of the sweep. Otherwise, it performs another sweep and tests for convergence again. The sequential pseudocode is shown in Figure 2.7. Let us now go through the steps to convert this simple equation solver



Expression for updating each interior point:

$$A[i,j] = 0.2 \times (A[i,j] + A[i,j - 1] + A[i - 1, j] + A[i,j + 1] + A[i + 1, j])$$

FIGURE 2.6 Nearest-neighbor update of a grid point in the simple equation solver. The black point is $A[i,j]$ in the two-dimensional array that represents the grid and is updated using itself and the four shaded points that are its nearest neighbors according to the equation at the right of the figure.

to a parallel program for each programming model. The decomposition and assignment steps are essentially the same for all three models, so these steps are examined in a general context. Once we enter the orchestration phase, the discussion will be organized explicitly by programming model.

2.3.2 Decomposition

For programs that are structured in successive loops or loop nests, a simple way to identify concurrency is to start from the loop structure itself. We examine the individual loops or loop nests in the program one at a time, see if their iterations can be performed in parallel, and determine whether this exposes enough concurrency. We can then look for concurrency across loops or take a different approach if necessary. Let us follow this program-structure-based approach in Figure 2.7.

Each iteration of the outermost while loop, beginning at line 15, sweeps through the entire grid. These iterations clearly are not independent since data modified in one iteration is accessed in the next. Consider the loop nest in lines 17–24, and ignore the lines containing `diff`. Look at the inner loop first (the `j` loop starting on line 18). Each iteration of this loop reads the grid point ($A[i,j - 1]$) that was written in the previous iteration. The iterations are therefore sequentially dependent, and we call this a *sequential loop*. The outer loop of this nest is also sequential, since the elements in row $i - 1$ were written in the previous ($i - 1$ th) iteration of this loop. So this simple analysis of existing loops and their dependences uncovers no concurrency in this example program.

In general, an alternative to relying on program structure to find concurrency is to go back to the fundamental dependences in the underlying algorithms used,

```

1. int n;                                /*size of matrix: (n + 2-by-n + 2) elements*/
2. float **A, diff = 0;

3. main()
4. begin
5.     read(n);                         /*read input parameter: matrix size*/
6.     A ← malloc (a 2-d array of size n + 2 by n + 2 doubles);
7.     initialize(A);                  /*initialize the matrix A somehow*/
8.     Solve (A);                     /*call the routine to solve equation*/
9. end main

10. procedure Solve (A)                 /*solve the equation system*/
11.    float **A;
12.    begin
13.        int i, j, done = 0;
14.        float diff = 0, temp;
15.        while (!done) do             /*outermost loop over sweeps*/
16.            diff = 0;               /*initialize maximum difference to 0*/
17.            for i ← 1 to n do      /*sweep over nonborder points of grid*/
18.                for j ← 1 to n do
19.                    temp = A[i,j];   /*save old value of element*/
20.                    A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                        A[i,j+1] + A[i+1,j]); /*compute average*/
22.                    diff += abs(A[i,j] - temp);
23.                end for
24.            end for
25.            if (diff/(n*n) < TOL) then done = 1;
26.        end while
27.    end procedure

```

FIGURE 2.7 Pseudocode describing the sequential equation solver kernel. The main body of work to be done in each sweep is in the nested for loop in lines 17–23. This is what we would like to parallelize. (Italics indicate keywords of the sequential programming language.)

regardless of program or loop structure. In the equation solver, we might look at the fundamental dependences in the generation and usage of data values (*data dependences*) at the granularity of individual grid points. As discussed earlier, since the computation proceeds from left to right and top to bottom in the grid, computing a particular grid point in the sequential program uses the updated values of the grid points directly above and to the left. This data dependence pattern is shown in Figure 2.8. The result is that the elements along a given anti-diagonal (southwest to northeast) have no dependences among them and can be computed in parallel, whereas the points in the next anti-diagonal depend on some points in the previous one. From this diagram, we can observe that of the $O(n^2)$ work involved in each

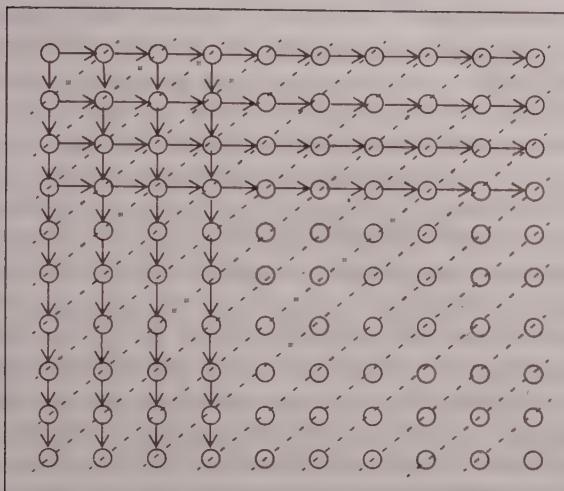


FIGURE 2.8 Dependences and concurrency in the Gauss-Seidel equation solver computation. The horizontal and vertical lines with arrows indicate dependences; the anti-diagonal, dashed lines connect points with no dependences among them that can be computed in parallel.

sweep, there is an inherent concurrency proportional to n along anti-diagonals and a sequential dependence proportional to n along the diagonal.

Suppose we decide to decompose the work into individual grid points so that updating a single grid point is a task. We can exploit the concurrency this exposes in several ways. First, we can leave the loop structure of the program as it is and insert point-to-point synchronization to ensure that the new value for a grid point has been produced in the current sweep before it is used by the points below or to its right. Thus, different loop nests (of the sequential program) and even different sweeps might be in progress simultaneously on different elements, as long as the element-level dependences are not violated. But the overhead of this synchronization at grid-point level may be too high. Second, we can change the loop structure: the first for loop (line 17) can be over anti-diagonals and the inner for loop can be over elements within an anti-diagonal. The inner loop can then be executed completely in parallel, with global synchronization between iterations of the outer for loop (to preserve dependences conservatively across anti-diagonals). Communication would be orchestrated very differently in the two cases, particularly if communication is in explicit messages. However, this approach also has problems. Global synchronization is still very frequent—once per anti-diagonal. In addition, the number of iterations in the parallel (inner) loop changes with successive outer loop iterations, as the size of the anti-diagonals changes, causing load imbalances among processes especially in the shorter anti-diagonals. Because of the frequency of synchronization, load imbalances, and programming complexity, neither of these approaches is used much on modern architectures.

The third and most common approach is based on exploiting knowledge of the problem beyond the dependences in the sequential program itself. The order in which the grid points are updated in the sequential algorithm (left to right and top to bottom) is in fact not fundamental to the Gauss-Seidel solution method; it is

simply one possible ordering that is convenient to program sequentially. Since the Gauss-Seidel method is not an exact solution method (unlike Gaussian elimination, for example) but rather iterates until convergence, we can update the grid points in a different order as long as we use updated values for grid points frequently enough.² One such ordering that is used often for parallel versions is called *red-black* ordering. The idea here is to separate the grid points into alternating red points and black points as on a checkerboard (see Figure 2.9) so that no red point is adjacent to a black point or vice versa. Since each point reads only its four nearest neighbors, to compute a given red point, we do not need the updated value of any other red point; we need only the updated values of the black points above it and to its left (in a standard sweep) and vice versa for computing black points. We can therefore divide a grid sweep into two phases, first computing all red points and then computing all black points. Within each phase no dependences exist among grid points, so we can compute all $n^2/2$ red points in parallel, synchronize globally, and then compute all $n^2/2$ black points in parallel. Global synchronization is conservative and can be replaced by point-to-point synchronization at the level of grid points since not all black points need to wait for all red points to be computed; but global synchronization is convenient.

Since the red-black ordering is different from our original sequential ordering, it can converge in fewer or more sweeps. It can also produce different final values for the grid points (though still within the convergence tolerance). While the red-point updates do not see updated values of any black points, the black points will see the updated values of all their red neighbors from the first phase of the current sweep, not just the ones to the left and above. Whether the new order is sequentially better or worse than the old one depends on the problem. The red-black ordering also has the advantage that the produced values and the convergence properties are independent of the number of processors used since no dependences occur within a phase. If the sequential program itself uses a red-black ordering, then parallelism does not change the results or convergence properties at all, thus making the parallel program deterministic.

Red-black ordering itself produces a longer kernel of code than is appropriate for this illustration of parallel programming. Let us examine a simpler but still common asynchronous method that does not separate points into red and black. This method simply ignores dependences among grid points within a sweep. Global synchronization is used between grid sweeps as in the preceding approach, but the loop structure for a process within a sweep is not changed from the top-to-bottom, left-to-right order. Instead, within a sweep a process simply updates the values of all its assigned grid points, accessing its nearest neighbors whether they have been updated in the current sweep by their assigned processes or not. When only a single process is used, this defaults to the original sequential ordering of updates. When multiple processes are used, the ordering is unpredictable; it depends on the assignment of

2. Even if we don't use updated values from the current sweep (i.e., while loop iteration) for any grid points but always use the values as they were at the end of the previous sweep, the system will still converge, only much slower. This is called Jacobi, rather than Gauss-Seidel, iteration.

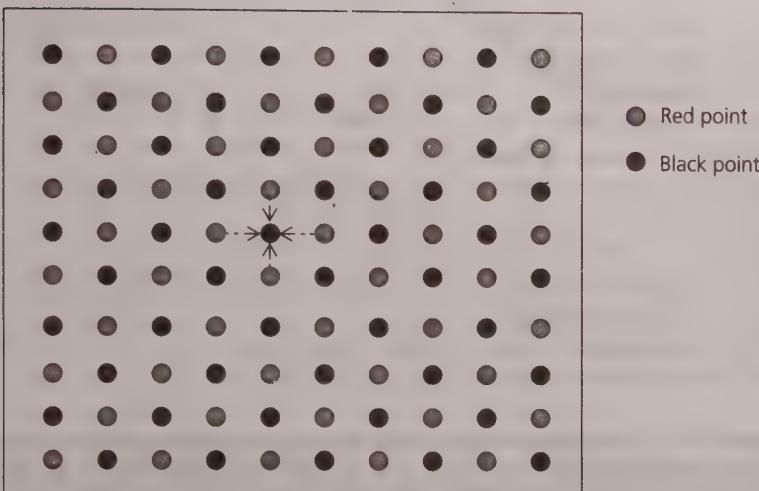


FIGURE 2.9 Red-black ordering for the equation solver. The sweep over the grid is broken up into two subsweeps: the first computes all the red points and the second all the black points. Since red points depend only on black points and vice versa, no dependences occur within a subsweep.

points to processes, the number of processes used, and how quickly different processes execute relative to one another at run time. The execution is no longer deterministic, and the number of sweeps required to converge may depend on the number of processes used; however, for most reasonable assignments the number of sweeps will not vary much.

If we choose a decomposition into individual inner loop iterations (grid points), we can express the program by revising lines 15–26 of Figure 2.7. Figure 2.10 highlights the changes to the code in boldface: all we have done is replace the keyword `for` in the parallel loops with `for_all`. A `for_all` loop simply tells the underlying hardware/software system that all iterations of the loop can be executed in parallel without worrying about dependences, but it says nothing about assignment. A loop nest with both nesting levels being `for_all` means that all iterations in the loop nest ($n \times n$ or n^2 here) can be executed in parallel. The system can assign and orchestrate the parallelism in any way it chooses; the program does not take a position on this. All it assumes is an implicit global synchronization after a `for_all` loop nest.

In fact, we can decompose the computation not just into individual inner loop iterations but into any aggregated groups of iterations we desire. Notice that decomposing the computation corresponds very closely to decomposing the grid itself. Suppose we wanted to decompose into rows of grid points instead so that the work for an entire row is an indivisible task that must be assigned to the same process. We could express this by making the inner loop on line 18 a sequential loop, changing its `for_all` back to a `for`, but leaving the loop over rows on line 17 as a parallel

```

15. while (!done) do           /*a sequential loop*/
16.     diff = 0;
17.     for_all i ← 1 to n do   /*a parallel loop nest*/
18.         for_all j ← 1 to n do
19.             temp = A[i,j];
20.             A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                     A[i,j+1] + A[i+1,j]);
22.             diff += abs(A[i,j] - temp);
23.         end for_all
24.     end for_all
25.     if (diff/(n*n) < TOL) then done = 1;
26. end while

```

FIGURE 2.10 Parallel equation solver kernel with decomposition into grid points and no explicit assignment. Since both for loops are made parallel by using `for_all` instead of `for`, the decomposition is into individual grid elements. Other than this change, the code is the same as the sequential code.

`for_all` loop. The parallelism, or *degree of concurrency*, exploited under this decomposition is reduced from n^2 inherent in the problem to n : instead of n^2 independent tasks of duration 1 unit each, we now have n independent tasks of duration n units each. If each task is executed on a different processor, we will have approximately $2n$ words of communication (accesses to grid points that were computed by other processors) for n points, which results in a communication-to-computation ratio of $O(1)$.

2.3.3 Assignment

Using the row-based decomposition, let us see how we might assign rows to processes explicitly. The simplest option is a static (predetermined) assignment in which each process is responsible for a contiguous block of rows, as shown in Figure 2.11. Interior row i is assigned to process

$$\left[\begin{array}{c} i \\ \bar{p} \end{array} \right]$$

where p is the number of processes. Alternative static assignments to this so-called block assignment are also possible, such as a *cyclic* assignment in which rows are interleaved among processes (process i is assigned rows i , $i + p$, and so on). We might also consider a dynamic assignment where each process repeatedly grabs the next available (not yet computed) row after it finishes with a row task, so that it is not predetermined which process computes which rows. For now, we will work with the static block assignment. This simple partitioning of the problem exhibits good load balance across processes as long as the number of interior rows is divisible by the number of processes, since the work per row is uniform across rows. Observe

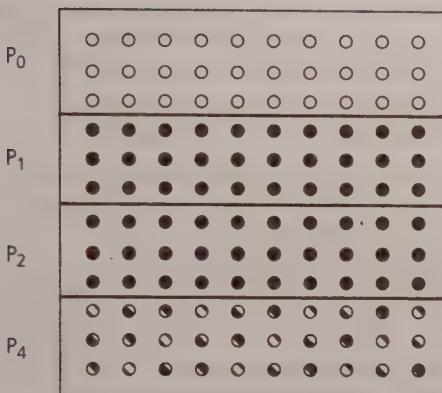


FIGURE 2.11 A simple assignment for the parallel equation solver. Each of the four processors is assigned a contiguous, equal number of rows of the grid. In each sweep, a processor will perform the work needed to update the elements of its assigned rows. Only the interior rows, which are updated in a sweep, are shown in the figure.

that the static assignments have further reduced the parallelism or degree of concurrency, from n to p , by making tasks larger, and the block assignment has reduced the communication required by assigning adjacent rows to the same processor. The communication-to-computation ratio is now only

$$O\left(\frac{p}{n}\right)$$

Having examined decomposition and assignment, we are ready to dig into the orchestration phase. This requires that we pin down the programming model. We begin with a high-level, data parallel model and then look at the two major programming models that the data parallel and other models might compile down to: shared address space and explicit message passing.

2.3.4 Orchestration under the Data Parallel Model

The data parallel model is convenient for the equation solver kernel since it is natural to view the computation as a single thread of control performing global transformations on a large array data structure (Hillis 1985; Hillis and Steele 1986). Computation and data are quite interchangeable, a simple decomposition and assignment of the data leads to good load balance across processes, and the appropriate assignments (partitions) are very regular in shape and can be described by simple expressions. Pseudocode for the data parallel equation solver is shown in Figure 2.12. We assume that global declarations (outside any procedure) describe shared data and that all other data (for example, data on a procedure's stack) is private to a process. Dynamically allocated shared data, such as the array A , is allocated with a `G_MALLOC` (global malloc) call rather than a regular `malloc`. The `G_MALLOC` allocates data in a shared region of the heap storage, which can be accessed and modified by any process. Other than this, the main differences (shown in boldface) from the sequential program are the use of `for_all` loops instead of

```

1. int n, nprocs;           /*grid size (n + 2-by-n + 2) and number of processes*/
2. float **A, diff = 0;

3. main()
4. begin
5.     read(n); read(nprocs); /*read input grid size and number of processes*/
6.     A ← G_MALLOC (a 2-d array of size n+2 by n+2 doubles);
7.     initialize(A);      /*initialize the matrix A somehow*/
8.     Solve (A);          /*call the routine to solve equation*/
9. end main

10. procedure Solve(Ā)        /*solve the equation system*/
11.    float **A;            /*A is an (n + 2-by-n + 2) array*/
12.    begin
13.        int i, j, done = 0;
14.        float mydiff = 0, temp;
15.        DECOMP A[BLOCK, *, nprocs];
16.        while (!done) do      /*outermost loop over sweeps*/
17.            mydiff = 0;          /*initialize maximum difference to 0*/
18.            for_all i ← 1 to n do /*sweep over non-border points of grid*/
19.                for_all j ← 1 to n do
20.                    temp = A[i,j]; /*save old value of element*/
21.                    A[i,j] ← 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
22.                        A[i,j+1] + A[i+1,j]); /*compute average*/
23.                    mydiff += abs(A[i,j] - temp);
24.                end for_all
25.            end for_all
26.            REDUCE (mydiff, diff, ADD);
27.            if (diff/(n*n) < TOL) then done = 1;
28.        end while
29.    end procedure

```

FIGURE 2.12 Pseudocode describing the data parallel equation solver. Differences from the sequential code are shown in boldface. Italicized boldface indicates constructs designed to achieve parallelism. The decomposition is still into individual elements, as indicated by the nested **for_all** loop. The assignment, indicated by the (unfortunately) labeled **DECOMP** statement, is into blocks of contiguous rows (the first, or column, dimension is partitioned into blocks, and the second, or row, dimension is not partitioned). The **REDUCE** statement sums the locally computed **mydiffs** into a global **diff** value. The while loop is still serial.

for loops, the use of a **DECOMP** statement, the use of a private **mydiff** variable per process, and the use of a **REDUCE** statement.

We have already seen that **for_all** loops specify that the iterations can be performed in parallel. The parallel processes other than the one executing the main thread of control are implicit in the data parallel model and are active only during

these parallel loops. The `DECOMP` statement has a twofold purpose. First, it specifies the assignment of the iterations to processes (`DECOMP` is in this sense an unfortunate word choice). Here, it is a `[BLOCK, *, nprocs]` assignment, which means that the first dimension (rows) is partitioned into contiguous blocks among the `nprocs` processes, and the second dimension is not partitioned at all. Specifying `[CYCLIC, *, nprocs]` would have implied a cyclic or interleaved partitioning of rows among `nprocs` processes, specifying `[BLOCK, BLOCK, nprocs]` would have implied a 2D contiguous block partitioning, and specifying `[*, CYCLIC, nprocs]` would have implied an interleaved partitioning of columns. The second and related purpose of `DECOMP` is that it also specifies how the grid data should be distributed among memories on a distributed-memory machine. (This is restricted to be the same as the assignment of computation in most current data parallel languages, following the owner computes rule, which works well in this example.) The `mydiff` variable is used to allow each process to first independently compute the sum of the difference values for its assigned grid points. Then, the `REDUCE` statement directs the system to add all the partial `mydiff` values together into the shared `diff` variable. This increases concurrency, as was discussed in Example 2.1. The `REDUCE` operation implements a *reduction*, which is a scenario in which many processes (all, in a global reduction) perform associative operations (such as addition, taking the maximum, etc.) on the same logically shared data. Associativity implies that the order of the operations does not matter. Floating-point operations such as the ones here are, strictly speaking, not associative since the way in which rounding errors accumulate depends on the order of operations. However, the effects are small and we usually ignore them, especially in iterative calculations that are approximate anyway. The reduction operation may be implemented in a library in a manner best suited to the underlying architecture.

While the data parallel programming model is well suited to specifying partitioning and data distribution for regular computations on large arrays of data (such as the equation solver kernel or the Ocean application), the suitability does not always hold true for more irregular applications, particularly those in which the communication pattern or the distribution of work among tasks changes unpredictably with time. (For example, think of the stars in Barnes-Hut or the rays in Raytrace, where assigning equal numbers of rays to processes would lead to severe load imbalances.) Let us look at the more flexible, lower-level programming models in which processes are explicit, have their own individual threads of control, and communicate with each other when they please.

2.3.5

Orchestration under the Shared Address Space Model

In a shared address space, we can simply declare the matrix `A` as a single shared array—as we did in the data parallel model—and processes can reference the parts of it they need using loads and stores with exactly the same array indices as in a sequential program. Communication is generated implicitly as necessary. With explicit parallel processes, we now need mechanisms to create the processes, coordinate them through synchronization, and control the assignment of work to

Table 2.2 Key Shared Address Space Primitives

Name	Syntax	Function
CREATE	CREATE(p, proc, args)	Create p processes that start executing at procedure proc with arguments args
G_MALLOC	G_MALLOC(size)	Allocate shared data of size bytes
LOCK	LOCK(name)	Acquire mutually exclusive access
UNLOCK	UNLOCK(name)	Release mutually exclusive access
BARRIER	BARRIER(name, number)	Global synchronization among number processes: none gets past BARRIER until number have arrived
WAIT_FOR_END	WAIT_FOR_END(number)	Wait for number processes to terminate
wait for flag	while (!flag); or WAIT(flag)	Wait for flag to be set (spin or block); used for point-to-point event synchronization
set flag	flag = 1; or SIGNAL(flag)	Set flag; wakes up process that is spinning or blocked on flag, if any

processes. The primitives we use are typical of low-level programming environments such as *parmacs* (Boyle et al. 1987) and are summarized in Table 2.2.

Pseudocode for the parallel equation solver in a shared address space is shown in Figure 2.13. The special primitives for parallelism are shown in boldface. They are typically implemented as library calls or macros, each of which expands to a number of instructions that accomplishes its goal. Although the code for the *Solve* procedure is remarkably similar to the sequential version, let's go through it one step at a time.

A single process is first started up by the operating system to execute the program, starting from the procedure called *main*. Let's call it the main process. It reads the input, which specifies the size of the grid A (recall that input n denotes an $(n + 2)$ -by- $(n + 2)$ grid of which n-by-n points are updated by the solver). It then allocates the grid A as a two-dimensional array in the shared address space using the G_MALLOC call (see Section 2.3.4) and initializes the grid. For data that is not dynamically allocated on the heap, different systems make different assumptions about what is shared and what is private to a process. Let us make the same assumptions as in the earlier data parallel example. Data declared outside any procedure, such as nprocs and n in Figure 2.13, is shared. Data on a procedure's stack (such as mymin, mymax, mydiff, temp, i, and j) is private to a process that executes the procedure, as is data allocated with a regular malloc call (and data that is explicitly declared to be private, not used in this program).

Having allocated data and initialized the grid, the program is ready to start solving the system. It creates (`nprocs - 1`) “worker” processes, which begin executing at the procedure called `Solve`. The main process then also calls the `Solve` procedure so that all `nprocs` processes enter the procedure in parallel as equal partners. All created processes execute the same code image until they exit from the program and terminate. That is, we use a structured, single-program-multiple-data (SPMD) style of programming. This does not mean that they proceed in lockstep or even execute the same instructions (as in the single-instruction-multiple-data or SIMD model) since, in general, they may follow different control paths through the code. Control over the assignment of work to processes—as well as what data they access—is maintained by a few private variables that acquire different values for different processes (e.g., `mymin` and `mymax`) and by simple manipulations of loop control variables. For example, we assume that every process upon creation automatically obtains a unique process identifier (`pid`) between 0 and `nprocs - 1` in its private address space and that it uses this `pid` (in lines 14a–b) to determine which rows are assigned to it. Processes synchronize through calls to synchronization primitives, which will be discussed shortly.

We assume for simplicity that the total number of interior rows `n` is an integer multiple of the number of processes `nprocs` so that every process is assigned the same number of rows. Each process calculates the indices of the first and last rows of its assigned block in the private variables `mymin` and `mymax`. It then proceeds to the actual solution loop.

The outermost while loop (line 15) is still over successive grid sweeps. Although the iterations of this loop proceed sequentially, each iteration or sweep is itself executed in parallel by all processes. The decision of whether to execute the next sweep is taken separately by each process or thread of control (by setting the `done` variable and computing the `while (!done)` condition) even though in this case each will make the same decision: the redundant work performed here is very small compared to the cost of communicating a completion flag or the `diff` value among the processors.

The code that performs the actual updates (lines 19–22) is essentially identical to that in the sequential program. Other than the bounds in the loop control statements, which control assignment, the only difference is that each process maintains its own private variable `mydiff`. As in the data parallel example, this private variable keeps track of the total difference between new and old values for only its assigned grid points. It is accumulated once into the shared `diff` variable at the end of the sweep, rather than adding directly into the shared variable for every grid point. In addition to the serialization and concurrency reason discussed in Section 2.2.1 (Example 2.1), all processes repeatedly modifying and reading the same shared variable cause a lot of expensive communication, so we do not want to do this once per grid point.

The interesting aspect of the rest of the program (line 25 onward) is synchronization—both mutual exclusion and event synchronization. First, the accumulations into the shared variable by different processes have to be mutually exclusive. To see why, consider the sequence of instructions that a processor executes to add its

```

1.     int n, nprocs;          /*matrix dimension and number of processors to be used*/
2a.    float **A, diff;       /*A is global (shared) array representing the grid*/
3.        /*diff is global (shared) maximum difference in current
4.        sweep*/
2b.    LOCKDEC(diff_lock); /*declaration of lock to enforce mutual exclusion*/
2c.    BARDEC (bar1);      /*barrier declaration for global synchronization between
5.        sweeps*/
6.
7.    main()
8.    begin
9.        read(n); read(nprocs); /*read input matrix size and number of processes*/
10.       A  $\leftarrow$  G_MALLOC (a two-dimensional array of size n+2 by n+2 doubles);
11.       initialize(A);           /*initialize A in an unspecified way*/
12.       CREATE (nprocs-1, Solve, A);
13.       Solve(A);              /*main process becomes a worker too*/
14.       WAIT_FOR_END (nprocs-1); /*wait for all child processes created to terminate*/
15.    end main
16.
17.    procedure Solve(A)
18.    float **A;                /*A is entire n+2-by-n+2 shared array,
19.                                as in the sequential program*/
20.    begin
21.        int i,j, pid, done = 0;
22.        float temp, mydiff = 0;           /*private variables*/
23.        int mymin = 1 + (pid * n/nprocs); /*assume that n is exactly divisible by*/
24.        int mymax = mymin + n/nprocs - 1 /*nprocs for simplicity here*/
25.
26.        while (!done) do                  /*outer loop over all diagonal elements*/
27.            mydiff = diff = 0;             /*set global diff to 0 (okay for all to do it)*/
28.            BARRIER(bar1, nprocs);        /*ensure all reach here before anyone modifies diff*/
29.            for i  $\leftarrow$  mymin to mymax do /*for each of my rows*/
30.                for j  $\leftarrow$  1 to n do          /*for all nonborder elements in that row*/
31.                    temp = A[i,j];
32.                    A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
33.                        A[i,j+1] + A[i+1,j]);
34.                    mydiff += abs(A[i,j] - temp);
35.                endfor
36.            endfor
37.            LOCK(diff_lock);             /*update global diff if necessary*/
38.            diff += mydiff;
39.            UNLOCK(diff_lock);
40.            BARRIER(bar1, nprocs);        /*ensure all reach here before checking if done*/

```

```

25e.    if (diff/(n*n) < TOL) then done = 1;      /*check convergence; all get
25f.    BARRIER(bar1, nprocs);
26. endwhile
27. end procedure

```

FIGURE 2.13 Pseudocode describing the parallel equation solver in a shared address space.

Line numbers followed by a letter denote lines that were not present in the sequential version. The numbers are chosen to match the line or control structure in the sequential code with which the new lines are most closely related. The design of the data structures does not have to change from the sequential program. Processes are created with the CREATE call, and the main process waits for them to terminate at the end of the program with the WAIT_FOR_END call. The decomposition is into rows, since the inner loop is unmodified, and the outer loop specifies the assignment of rows to processes. Barriers are used to separate sweeps (and to separate the convergence test from further modification of the global diff variable), and locks are used to provide mutually exclusive access to the global diff variable.

mydiff variable (maintained, say, in register r2) into the shared diff variable (i.e., to execute the source statement `diff += mydiff`):

```

load the value of diff into register r1
add the register r2 to register r1
store the value of register r1 into diff

```

Suppose the value in the variable diff is 0 to begin with and the value of mydiff in each process is 1. After two processes have executed this code, we would expect the value in diff to be 2. However, it may turn out to be 1 instead if the processes happen to execute their operations interleaved in the following order (shown vertically):

P_1	P_2
$r1 \leftarrow \text{diff}$ { P_1 gets 0 in its $r1$ }	
	$r1 \leftarrow \text{diff}$ { P_2 also gets 0}
$r1 \leftarrow r1 + r2$. { P_1 sets its $r1$ to 1}	$r1 \leftarrow r1 + r2$ { P_2 sets its $r1$ to 1}
$\text{diff} \leftarrow r1$ { P_1 stores 1 into diff}	$\text{diff} \leftarrow r1$ { P_2 also stores 1 into diff}

This is not what we intended. The problem is that a process (here P_2) may be able to read the value of the logically shared diff between the time that another process (P_1) reads it and writes it back. To prohibit this interleaving of operations, we would like the sets of operations from different processes to execute *atomically* (i.e., to achieve mutual exclusion) with respect to one another. The set of operations we want to execute atomically is called a *critical section*: once a process starts to execute the first of its three instructions above (its critical section), no other process can execute any of the instructions in its corresponding critical section until the former

process has completed its last instruction of the critical section. The LOCK-UNLOCK pair around line 25b achieves mutual exclusion for the critical section composed of `diff += mydiff`.

A lock such as `cell_lock` can be viewed as a shared token that confers an exclusive right. Acquiring the lock through the LOCK primitive gives a process the right to execute the critical section. The process that holds the lock frees it by issuing an UNLOCK command when it has completed the critical section. At this point, the lock is free for another process to either acquire or be granted, depending on the implementation. The LOCK and UNLOCK primitives must be implemented in a way that guarantees mutual exclusion. Locks are expensive, and even a given lock can cause contention and serialization if multiple processes try to access it at the same time. Our LOCK primitive takes as its argument the name of the lock being used. Associating names with locks allows us to use different locks to protect unrelated critical sections, reducing contention and serialization.

Once a process has added its `mydiff` into the global `diff`, it waits until all processes have done so and the value contained in `diff` is indeed the total difference over all grid points. This requires global event synchronization, implemented here with a BARRIER. A barrier operation takes as an argument the name of the barrier and the number of processes involved in the synchronization, and it is issued by all those processes. When a process calls the barrier, it registers the fact that it has reached that point in the program. The process is not allowed to proceed past the barrier call until the specified number of processes participating in the barrier have issued the barrier operation. That is, the semantics of `BARRIER(name, p)` are as follows: wait until `p` processes get here and only then proceed. The need for the other two barriers in the program is discussed in Exercise 2.6.

Barriers are often used to separate distinct phases of computation in a program. For example, in the Barnes-Hut galaxy simulation we use a barrier between updating the positions of the stars at the end of one time-step and using them to compute forces at the beginning of the next one, and in Data Mining we may use a barrier between counting occurrences of candidate itemsets and using the resulting large itemsets to generate the next list of candidates. Since barriers implement all-to-all event synchronization, they are usually a conservative way of preserving dependences; usually, not all operations (or processes) after the barrier actually need to wait for all operations before the barrier to complete. More specific event synchronization between pairs or groups of processes would enable some processes to get past their synchronization operation earlier; however, from a programming viewpoint it is often more convenient to use a single barrier than to orchestrate the actual dependences through point-to-point synchronization among processes.

When point-to-point synchronization is needed, one way to orchestrate it in a shared address space is with wait and signal operations on semaphores, with which we are familiar from operating systems. A more common way in parallel programs is by using normal shared variables as flags for event synchronization, as shown in Figure 2.14. Since `P1` simply spins around in a tight while loop waiting for the flag variable to be set to 1, keeping the processor busy during this time, we call this `spinning` or `busy-waiting`. Recall that in the case of a semaphore the waiting proce

P₁	P₂
	A = 1;
	b: flag
a: while (flag is 0) do nothing;	= 1;
print A;	

FIGURE 2.14 Point-to-point event synchronization using flags. Suppose we want to ensure that a process P_1 does not get past a certain point (say, a) in the program until some other process P_2 has already reached another point (say, b). Assume that the variable flag (and A) was initialized to 0 before the processes arrived at this scenario. If P_1 gets to statement a after P_2 has already executed statement b , P_1 will simply pass point a . If, on the other hand, P_2 has not yet executed b , then P_1 will remain in the “idle” while loop until P_2 reaches b and sets flag to 1, at which point P_1 will exit the idle loop and proceed. If we assume that the writes by P_2 are seen by P_1 in the order in which P_2 issues them, then this synchronization will ensure that P_1 prints the value 1 for A .

does not spin and consume processor resources but rather blocks (suspends) itself and is awakened when another process signals the semaphore.

In event synchronization among subsets of processes, or *group event synchronization*, one or more processes may wait for an event and one or more processes may notify them of its occurrence. Group event synchronization can be orchestrated either using ordinary shared variables as flags or by using barriers among subsets of processes.

Returning to the equation solver in Figure 2.13, once a process is past the barrier, it reads the value of `diff` and examines whether the average difference over all grid points ($diff / (n*n)$) is less than the error tolerance used to determine convergence. If so, it sets the `done` flag to exit from the while loop; if not, it goes on to perform another sweep.

Finally, the `WAIT_FOR_END` called by the main process at the end of the program (line 8b) is a particular form of all-to-one synchronization. Through it, the main process waits for all the worker processes it created to terminate. The other processes do not call `WAIT_FOR_END` but implicitly participate in the synchronization by terminating when they exit the `Solve` procedure that was their entry point into the program.

In summary, for this simple equation solver the parallel program in a shared address space is not too different in structure from the sequential program. The major differences in the control flow are implemented by changing the bounds on some loops. Additional differences are due to creating processes, partitioning the work among them, and synchronizing through the use of simple and generic primitives. The body of the computational loop is mostly unchanged, as are the major data structures and the references to them. Given a strategy for decomposition, assignment, and synchronization, inserting the necessary primitives and making the necessary modifications to produce a correct parallel program is quite mechanical in this example. Changes to decomposition and assignment are also easy to incorporate

```

17. for i ← pid+1 to n by nprocs do      /*for my interleaved set of rows*/
18.   for j ← 1 to n do                  /*for all elements in that row*/
19.     temp = A[i,j];
20.     A[i,j] = 0.2 * (A[i,j] + A[i,j-1] + A[i-1,j] +
21.                       A[i,j+1] + A[i+1,j]);
22.     mydiff += abs(A[i,j] - temp);
23.   endfor
24. endfor

```

FIGURE 2.15 Cyclic assignment of row-based solver in a shared address space. All that changes in the code from the block assignment of rows in Figure 2.13 is the first for statement in line 17. The data structures or accesses to them do not have to be changed.

as shown in Example 2.2. Although many simple programs have these properties in a shared address space, we will see later that more substantial changes are needed as we seek to obtain higher parallel performance and as we address more complex parallel programs.

EXAMPLE 2.2 How would the code for the shared address space parallel version of the equation solver (Figure 2.13) change if we retained the same decomposition into rows but changed to a cyclic (interleaved) assignment of rows to processes?

Answer Figure 2.15 shows the relevant pseudocode. All that has changed in the code is the control arithmetic in line 17. The same global data structure is used with the same indexing, and the rest of the parallel program stays exactly the same. ■

2.3.6 Orchestration under the Message-Passing Model

We now examine a possible implementation of the parallel solver using explicit message passing between private address spaces, employing the same decomposition and assignment as before. Since we no longer have a shared address space, we cannot simply declare the matrix A to be shared and have processes reference parts of it as they would in a sequential program. Rather, the logical data structure A must be represented by a collection of smaller per-process data structures, which are allocated among the private address spaces of the cooperating processes in accordance with the assignment of work. In particular, the process that is assigned a block of rows allocates those rows as an array in its private address space.

A set of simple primitives for message-passing programming are shown in Table 2.3. The message-passing program shown in Figure 2.16 uses some of these primitives and is structurally very similar to the shared address space program in Figure 2.13 (more complex programs will reveal further differences in Section 3.6). Here too a main process is started by the operating system when the program executable is invoked, and this main process creates $nprocs - 1$ other processes to collaborate with it. We assume again that every created process automatically acquires a

Table 2.3 Some Basic Message-Passing Primitives

Name	Syntax	Function
CREATE	CREATE(procedure)	Create process that starts at procedure
SEND	SEND(src_addr, size, dest, tag)	Send size bytes starting at src_addr to the dest process, with tag identifier
RECEIVE	RECEIVE(buffer_addr, size, src, tag)	Receive a message with the tag identifier from the src process, and put size bytes of it into buffer starting at buffer_addr
SEND_PROBE	SEND_PROBE(tag, dest)	Check if message with identifier tag has been sent to process dest (only for asynchronous message passing, and meaning depends on semantics, as discussed in this section)
RECV_PROBE	RECV_PROBE(tag, src)	Check if message with identifier tag has been received from process src (only for asynchronous message passing, and meaning depends on semantics)
BARRIER	BARRIER(name, number)	Global synchronization among number processes: none gets past BARRIER until number have arrived
WAIT_FOR_END	WAIT_FOR_END(number)	Wait for number processes to terminate

process identifier (pid) between 0 and nprocs - 1, and that the CREATE call automatically communicates the program's input parameters (n and nprocs) to the address space of each process.³ The outermost loop of the Solve routine (line 15) still iterates over grid sweeps until convergence, and in every iteration, a process performs the computation for its assigned rows and communicates as necessary. The major differences are in orchestration: in the data structures used to represent the logically shared matrix A and in how interprocess communication and synchronization are implemented. We shall focus on these differences.

Instead of representing the matrix to be factored as a single global $(n + 2)$ -by- $(n + 2)$ array A, each process in the message-passing program allocates an array called myA of size $(\text{nprocs}/n + 2)$ -by- $(n + 2)$ in its private address space. This array represents its assigned nprocs/n rows of the logically shared matrix A, plus two rows at the edges to hold the boundary data from its neighboring partitions (for use in the grid point updates). The boundary rows from its neighbors must be communicated to it explicitly and copied into these extra, or *ghost*, rows since their elements cannot be directly referenced otherwise as they are not in the process's

3. An alternative organization is to use what is called a "hostless" model, in which there is no single main process. The number of processes to be used is specified to the system when the program is invoked. The system then starts up that many processes and distributes the code to the relevant processing nodes. There is no need for a CREATE primitive in the program itself; every process reads the program inputs (n and nprocs) separately, though processes still acquire unique user-level pids.

```

1. int pid, n, nprocs;           /*process id, matrix dimension and number of
                                processors to be used*/
2. float **myA;
3. main()
4. begin
5.   read(n);  read(nprocs);    /*read input matrix size and number of processes*/
8a.   CREATE (nprocs-1, Solve); /*main process becomes a worker too*/
8b.   Solve();                 /*wait for all child processes created to terminate*/
8c.   WAIT_FOR_END (nprocs-1);
9. end main

10. procedure Solve()
11. begin
13.   int i,j, pid, n' = n/nprocs, done = 0;
14.   float temp, tempdiff, mydiff = 0; /*private variables*/
6.   myA ← malloc(a 2-d array of size [n/nprocs + 2] by n+2);
                                /*my assigned rows of A*/
7.   initialize(myA);           /*initialize my rows of A, in an unspecified way*/

15. while (!done) do
16.   mydiff = 0;               /*set local diff to 0*/
16a.  if (pid != 0) then SEND(&myA[1,0],n*sizeof(float),pid-1,ROW);
16b.  if (pid = nprocs-1) then
        SEND(&myA[n',0],n*sizeof(float),pid+1,ROW);
16c.  if (pid != 0) then RECEIVE(&myA[0,0],n*sizeof(float),pid-1,ROW);
16d.  if (pid != nprocs-1) then
        RECEIVE(&myA[n'+1,0],n*sizeof(float), pid+1,ROW);
                                /*border rows of neighbors have now been copied
                                into myA[0,*] and myA[n'+1,*]*/
17.   for i ← 1 to n' do      /*for each of my (nonghost) rows*/
18.     for j ← 1 to n do    /*for all nonborder elements in that row*/
19.       temp = myA[i,j];
20.       myA[i,j] = 0.2 * (myA[i,j] + myA[i,j-1] + myA[i-1,j] +
21.                           myA[i,j+1] + myA[i+1,j]);
22.       mydiff += abs(myA[i,j] - temp);
23.     endfor
24.   endfor
                                /*communicate local diff values and determine if
                                done; can be replaced by reduction and broadcast*/

```

```

25a. if (pid != 0) then          /*process 0 holds global total diff*/
25b.   SEND(mydiff,sizeof(float),0,DIFF);
25c.   RECEIVE(done,sizeof(int),0,DONE);
25d. else                      /*pid 0 does this*/
25e.   for i ← 1 to nprocs-1 do /*for each other process*/
25f.     RECEIVE(tempdiff,sizeof(float),*,DIFF);
25g.     mydiff += tempdiff;    /*accumulate into total*/
25h.   endfor
25i.   if (mydiff/(n*n) < TOL) then done = 1;
25j.   for i ← 1 to nprocs-1 do /*for each other process*/
25k.     SEND(done,sizeof(int),i,DONE);
25l.   endfor
25m. endif
26. endwhile
27. end procedure

```

FIGURE 2.16 Pseudocode describing parallel equation solver with explicit message passing.

Now the meaning of the data structures and the indexing of them changes in going to the parallel code. Each process has its own `myA` data structure that represents its assigned part of the grid, and `myA[i,j]` referenced by different processes refers to different parts of the logical overall grid. The communication is all contained in lines 16a–16d and 25a–25f. No locks or barriers are needed since the synchronization is implicit in the send/receive pairs. Several extra lines of code are added to orchestrate the communication with simple sends and receives.

address space. Ghost rows are used because without them the communicated data would have to be received into separate one-dimensional arrays with different names created specially for this purpose, which would complicate the referencing of the data when they are read in the inner loop (lines 20–21). Since communicated data has to be copied into the receiver's private address space anyway, programming is made easier by extending the existing data structure rather than allocating new ones.

Recall from Chapter 1 that both communication and synchronization in a message-passing program are based on two primitives: `SEND` and `RECEIVE`. The program event that initiates data transfer is the `SEND` operation, unlike in a shared address space where data transfer is usually initiated by the consumer or receiver using a read (load) instruction. When a message arrives at the destination processor, it is either kept in the network queue or temporarily stored in a system buffer until a process running on the destination processor posts a `RECEIVE` for it. With a `RECEIVE`, a process reads an incoming message from the network or system buffer into a specified portion of the private (application) address space. A `RECEIVE` does not in itself cause any data to be transferred across the network.

The simple `SEND` and `RECEIVE` primitives used in the example program assume that the data being transferred is in a contiguous region of the virtual address space. The arguments in our simple `SEND` call are: the start address of the data to be sent,

which is in the sending process's private address space; the size of the message in bytes; the pid of the destination process, which we must be able to name explicitly now (unlike in a shared address space); and an optional tag or type associated with the message for matching at the receiver. The arguments to the RECEIVE call are a local address at which to place the received data, the size of the message, the sender's pid, and the optional message tag or type. The specified sender's pid and the tag, if present, are used to perform a match with the messages that have arrived and are in the system buffer, to see which one corresponds to the receive. Either or both of these fields may be wild cards, in which case they will match a message from any source process or with any tag, respectively. SEND and RECEIVE primitives are usually implemented in a library on a specific architecture, just like BARRIER and LOCK in a shared address space. A full set of message-passing primitives commonly used in real programs is part of a standard called the Message Passing Interface, or MPI (described at different levels of detail in Pacheco 1996; MPI Forum 1993; Gropp, Lusk, and Skjellum 1994). A significant extension is transfers of noncontiguous regions of memory, either with regular stride—such as every tenth word between addresses a and b , or four words every sixth word—or by using index arrays to specify unstructured addresses from which to gather data on the sending side or to which to scatter data on the receiving side. Another is a large degree of flexibility in specifying tags to match messages and in the potential complexity of a match. For example, processes may be divided into groups that communicate certain types of messages only within their group, and collective communication operations may be provided as described in the following.

Semantically, the simplest forms of SEND and RECEIVE we can use in our program are the so-called synchronous forms. A synchronous SEND operation returns control to the calling process only when it is clear that the corresponding RECEIVE has been performed. A synchronous RECEIVE returns control when the data has been received into the destination process's address space. With synchronous messages, our implementation of the communication in lines 16a–16d is actually deadlocked. All the processes issue their SEND first and stall until the corresponding receive is performed, so none will ever get to actually perform their RECEIVE! In general, synchronous message passing can easily deadlock on pairwise exchanges of data if we are not careful. One way to avoid this problem is to have every alternate process do its SENDS first followed by its RECEIVES, and the others do their RECEIVES first followed by their SENDS. The alternative is to use different semantic flavors of send and receive, as we shall see shortly.

The communication is done all at once at the beginning of each iteration, rather than grid point by grid point as needed in a shared address space. It could be done grid point by grid point, but the overhead of send and receive operations is usually too large to make this approach perform reasonably. As a result, unlike in the shared address space version, the message-passing program is deterministic. Even though one process updates its boundary rows while its neighbor is computing in the same sweep, the neighbor is guaranteed not to see the updates in the current sweep since they are not in its address space. A process therefore sees, in its neighbors' boundary rows, the values as they were at the end of the previous sweep, which may cause

more sweeps to be needed for convergence as per our earlier discussion (red-black ordering would have been particularly useful here).

Once a process has received its neighbors' boundary rows into its ghost rows, it can update its assigned points using code almost exactly like that in the sequential and shared address space programs. (Although we use a different name, `myA`, for a process's local array than the `A` used in the sequential and shared address space programs, this is just to distinguish it from the logically shared entire grid `A`, which here is only conceptual; we could just as well have used the name `A`.) The loop bounds are different, extending from 1 to `nprocs/n` (substituted by `n'` in the code) for all processes rather than 0 to `n - 1` as in the sequential program or `mymin` to `mymax` as in the shared address space program. In fact, the indices used to reference `myA` are local indices, which are different than the global indices that would be used if the entire logically shared grid `A` could be referenced as a single shared array. For example, a reference, `myA[1, 1]`, by different processes refers to different rows of the logically shared grid `A`. The use of local index spaces can be somewhat trickier in cases where a global index must also be used explicitly, as seen in Exercise 2.7.

Synchronization, including the accumulation of private `mydiff` variables into a logically shared `diff` variable and the evaluation of the done condition that follows, is performed very differently here than in a shared address space. Given our simple synchronous sends and receives that block the issuing process until they complete, the send/receive match encapsulates a synchronization event and no special operations (like locks and barriers) or additional variables are needed to orchestrate synchronization. Consider mutual exclusion. The logically shared `diff` variable must be allocated in some process's private address space (here process 0). The identity of this process must be known to all the others. Every process sends its `mydiff` value to process 0, which receives them all and adds them to the logically shared global `diff`. Since only process 0 can manipulate this logically shared variable, mutual exclusion and serialization occur naturally and no locks are needed. In fact, process 0 can simply use its own `mydiff` variable as the global `diff`.

Now consider the global event synchronization needed for determining the done condition. Once process 0 has received the `mydiff` values from all the other processes and accumulated them, it tests the done condition and then sends the `done` variable to all the other processes, which are waiting for it with receive calls. There is no need for a barrier because the completion of the synchronous receive implies that process 0 has sent out the `done` result and therefore that all processes' `mydiffs` have been accumulated. The processes then test the `done` condition locally to determine whether or not to proceed with another sweep. We could also, of course, implement lock and barrier calls using messages if that is more convenient for programming, although that may lead to request-reply communication and therefore more round-trip messages. More complex send/receive semantics than the synchronous ones we have used here may require additional synchronization beyond the messages themselves, as we shall see.

Notice that the code for the accumulation and `done` condition evaluation has expanded to several lines when using point-to-point sends and receives as communication operations. In practice, programming environments would provide library

functions like REDUCE (accumulate values from private variables in multiple processes to a single variable in a given process) and BROADCAST (send from one process to all processes) to the programmer, which the application processes could use directly to simplify the code in these stylized situations. Using these functions, lines 25a–25m in Figure 2.16 could be replaced by the five lines in Figure 2.17. The system may provide special support to improve the performance of these and other collective communication operations (such as multicast from one-to-several or even several-to-several processes, or all-to-all communication in which every process transfers data to every other process), for example, by reducing the software overhead at the sender to that of a single message, or these operations may be built on top of the usual point-to-point send and receive in user-level libraries for programming convenience only.

Finally, it was mentioned earlier that SEND and RECEIVE operations come in different semantic flavors, which we can use to solve our deadlock problem. Let us examine this a little further. The main axis along which these flavors differ is their completion semantics—that is, when they return control to the user process that issued the send or receive. These semantics affect when the data structures or buffers they operate on can be reused without compromising correctness. The two major kinds of SEND/RECEIVE are synchronous and asynchronous; within the asynchronous class are two types: blocking and nonblocking. Let us examine these options and see how they might be used in our program.

Synchronous SENDS and RECEIVES are what we have assumed previously because they have the simplest semantics for a programmer. A synchronous SEND returns control to the calling process only when the corresponding synchronous RECEIVE at the destination end has completed successfully and returned an acknowledgment to the sender. Until the acknowledgment is received, the sending process cannot execute any code that follows the SEND. Receipt of the acknowledgment implies that the receiver has retrieved the entire message from the system buffer into the applica-

```
/*communicate local diff values and determine if done, using reduction and broadcast*/
25b. REDUCE(0,mydiff,sizeof(float),ADD);
25c. if (pid == 0) then
25i.   if (mydiff/(n*n) < TOL) then done = 1;
25k.   endif
25m.   BROADCAST(0,done,sizeof(int),DONE);
```

FIGURE 2.17 Accumulation and convergence determination in the solver using **REDUCE** and **BROADCAST** instead of **SEND** and **RECEIVE**. The first argument to the REDUCE call is the destination process. All other processes will do a send to this process in the implementation of REDUCE while this process will do a receive. The next argument is the private variable to be reduced from (in all processes other than the destination) and to (in the destination process), and the third argument is the size of this variable. The last argument is the function to be performed on the variables in the reduction. Similarly, the first argument of the BROADCAST call is the sender; this process does a send and all others do a receive. The second argument is the variable to be broadcast and received into, and the third is its size. The final argument is the optional message type.

tion space. Thus, the completion of the SEND guarantees (barring hardware errors) that the message has been successfully received and that all associated data structures and buffers can be reused.

A *blocking asynchronous* (or simply *blocking*) SEND returns control to the calling process when the message has been taken from the sending application's source data structure and is therefore in the care of the system. This means that when control is returned, the sending process can modify the source data structure without affecting that message. Compared to a synchronous SEND, this allows the sending process to resume much sooner, but the return of control does not guarantee that the message has been or will actually be delivered to the appropriate process. Obtaining such a guarantee would require additional handshaking between the processes. A blocking asynchronous RECEIVE is similar to a synchronous RECEIVE in that it returns control to the calling process only when the data it is receiving has been successfully removed from the system buffer and placed at the designated application address. Once it returns, the application can immediately use the data in the specified application buffer. Unlike a synchronous RECEIVE, however, a blocking RECEIVE does not send an acknowledgment to the sender.

The *nonblocking asynchronous* (or simply *nonblocking*) SEND and RECEIVE allow the greatest overlap between computation and message passing by returning control most quickly to the calling process. A nonblocking SEND returns control immediately. A nonblocking RECEIVE returns control after simply posting the intent to RECEIVE; the actual receipt of the message and placement into a specified application data structure is performed asynchronously at an undetermined time by the system on the basis of the posted receive. In both the nonblocking SEND and RECEIVE, however, the return of control does not imply anything about the state of the message or the application data structures it uses, so it is the user's responsibility to determine that state when necessary. Separate primitives are provided to probe (query) the state. Nonblocking messages are thus typically used in a two-phase manner: first the SEND/RECEIVE operation itself and then, when needed, the probes. The probes, which must be provided by the message-passing library, might either block until the desired state is observed or might return control immediately and simply report what state was observed.

The kind of SEND/RECEIVE semantics we choose depends on how the program uses its data structures and to what degree we wish to trade off ease of programming and portability to systems with other semantics for performance. The semantics mostly affects event synchronization, since mutual exclusion falls out naturally from having only private address spaces. In the equation solver example, using asynchronous SENDs and blocking asynchronous RECEIVES would avoid the deadlock problem since processes would proceed past the SEND and to the RECEIVE. However, if we used nonblocking asynchronous RECEIVES, we would have to use a probe before actually using the data structure specified in the RECEIVE. Note that a blocking SEND/RECEIVE is equivalent to a nonblocking SEND/RECEIVE followed immediately by a blocking probe.

To better appreciate the differences between the shared address space and message-passing programming models, it will be instructive to perform an exercise to transform the message-passing version of the equation solver to use a cyclic assignment as we did for the shared address space version in Example 2.2. The point to observe in this case is that, although the two message-passing versions will look syntactically similar, the meaning of the `myA` data structure will be completely different. In one case it is a contiguous section of the global array, and in the other it is a set of widely separated rows. Only by careful inspection of the data structures and communication patterns can you determine how a given message-passing version corresponds to the original sequential program or its shared address space counterpart.

2.4

CONCLUDING REMARKS

The process of parallelizing a sequential application is quite structured: we decompose the work into tasks; assign the tasks to processes; orchestrate data access, communication, and synchronization among processes; and optionally map processes to processors. For many applications, including the simple equation solver used in this chapter, the initial decomposition and assignment are similar or even identical regardless of whether a shared address space or message-passing programming model is used. The differences are in orchestration, particularly in the way data structures are organized and accessed and the way communication and synchronization are performed. A shared address space allows us to use the same major data structures as in a sequential program to produce a correct parallel program. Communication is implicit through data accesses, and the decomposition of data is not required at least for correctness. In the message-passing case, we must synthesize the logically shared data structure from per-process private data structures. Communication is explicit, decomposition of data explicitly among private address spaces (processes) is necessary, and processes must be able to name one another to communicate. On the other hand, whereas a shared address space program requires additional synchronization primitives separate from the reads and writes used for implicit communication, synchronization is bundled into the explicit send and receive communication in many forms of message passing. As we examine the parallelization of more complex applications, such as the four case studies introduced in this chapter, we will understand the implications of these differences for ease of programming as well as the additional considerations imposed by the desire for high performance.

The parallel versions of the simple equation solver described here were designed to illustrate programming primitives. Although these versions will not perform terribly (e.g., we reduced communication by using a block rather than cyclic assignment of rows, and we reduced both communication and synchronization dramatically by first accumulating into local `mydiffs` and only then into a global `diff`), the programs can be improved. We shall see how in the next chapter, as we turn our attention to the performance issues in parallel programming and how positions taken on these issues affect the workload presented to the architecture.

2.5 EXERCISES

- 2.1 Describe two examples where a good parallel algorithm must be based on a serial algorithm that is different from the best serial algorithm since the latter does not afford enough concurrency.
- 2.2 Which of our case study applications (Ocean, Barnes-Hut, Raytrace, and Data Mining) do you think are amenable to decomposing data rather than computation and using an owner computes rule in parallelization? What do you think the problem(s) would be with using a strict data distribution and owner computes rule in the others?
- 2.3 There are two dominant models for how parent and children processes relate to each other in a shared address space. In the heavyweight so-called process model, when a process creates another process, the child gets a private copy of the parent's image; that is, if the parent had allocated a variable x , then the child also finds a variable x in its address space that is initialized to the value that the parent had for x when it created the child. However, any modifications that either process makes subsequently are to its own copy of x and are not visible to the other process. In the lightweight threads model, the child process or thread gets a pointer to the parent's image, so that it and the parent now see the same storage location for x . All data that any process or thread allocates is shared in this model, except that on a procedure's stack.
 - a. Consider the problem of a process having to reference its process identifier pid in various parts of a program, in different routines called (in a call chain) by the routine at which the process begins execution. How would you implement this in the first model? In the second? Do you need private data per process, or could you do this with all data being globally shared?
 - b. A program written in the former (process) model may rely on the fact that a child process gets its own private copies of the parents' data structures. What changes would you make to port the program to the latter (threads) model for data structures that are (i) only read by processes after the creation of the child and (ii) are both read and written?
- 2.4 The classic bounded buffer problem provides an example of point-to-point event synchronization. Two processes communicate through a finite buffer. One process, the producer, adds data items to a buffer when it is not full; another, the consumer, reads data items from the buffer when it is not empty. If the consumer finds the buffer empty, it must wait until the producer inserts an item. When the producer is ready to insert an item, it checks to see if the buffer is full, in which case it must wait until the consumer removes something from the buffer. If the buffer is empty when the producer tries to add an item, then depending on the implementation the consumer may be waiting for notification, so the producer may need to notify the consumer. Can you implement a bounded buffer with only point-to-point event synchronization, or do you need mutual exclusion as well? Design an implementation, including pseudocode.

- 2.5 Would you use spinning on a flag or blocking of processes for interprocess synchronization in uniprocessor operating systems? What do you think the trade-offs are between blocking and spinning on a multiprocessor?
- 2.6 In the shared address space parallel equation solver (Figure 2.13), why do we need the first and third barriers in a while loop iteration (lines 16a and 25f)? Can you eliminate them without inserting any other synchronization, perhaps altering when certain operations are performed? Think about all possible scenarios.
- 2.7 Gaussian elimination is a well-known technique for solving simultaneous linear systems of equations. Variables are eliminated one by one until there is only one left, and then the discovered values of variables are back-substituted to obtain the values of other variables. In practice, the coefficients of the unknowns in the equation system are represented as a matrix A , and the matrix is first converted to an upper-triangular matrix (a matrix in which all elements below the main diagonal are 0). Then back-substitution is used. Let us focus on the conversion to an upper-triangular matrix by successive variable elimination. Pseudocode for sequential Gaussian elimination is shown in Figure 2.18. The diagonal element for a particular iteration of the k loop is called the *pivot element*, and its row is called the *pivot row*.
 - a. Draw a simple figure illustrating the dependences among matrix elements.
 - b. Assuming a decomposition into rows and an assignment into blocks of contiguous rows, write a shared address space parallel version using the primitives used for the equation solver in this chapter.
 - c. Write a message-passing version for the same decomposition and assignment, first using synchronous message passing and then any form of asynchronous message passing.
 - d. Can you see obvious performance problems with this partitioning? (We will discuss this further in the next chapter.)
 - e. Modify both the shared address space and message-passing versions to use an interleaved assignment of rows to processes.
 - f. Discuss the trade-offs (programming difficulty and any likely major performance differences) in programming the shared address space and message-passing versions.
- 2.8 Suppose that a system supporting a shared address space did not support barriers but only semaphores. Even global event synchronization would have to be constructed through semaphores or ordinary flags. The use of semaphores can be illustrated as follows. Suppose process P_2 has to indicate to process P_1 (using semaphores) that P_2 has reached a point b in the program so that P_1 can proceed past a point a (where it was waiting). P_1 performs a wait (also called P or down) operation on a semaphore when it reaches point a , and P_2 performs a signal (or V or up) operation on the same semaphore when it reaches point b . If P_1 gets to a before P_2 gets to b , P_1 suspends or blocks itself and is awakened by P_2 's signal operation.
 - a. How might you orchestrate the synchronization in the shared address space parallel Gaussian elimination with (i) flags and (ii) semaphores replacing the

```

procedure Eliminate (A)          /*triangularize the matrix A*/
begin
for k ← 0 to n-1 do           /*loop over all diagonal (pivot) elements*/
begin
  for j ← k+1 to n-1 do      /*for all elements in the row of, and to the right of,
                                the pivot element*/
    Ak,j = Ak,j / Ak,k;   /*divide by pivot element*/
    Ak,k = 1;
    for i ← k+1 to n-1 do    /*for all rows below the pivot row*/
      for j ← k+1 to n-1 do  /*for all elements in the row*/
        Ai,j = Ai,j - Ai,k* Ak,j;
      endfor
      Ai,k = 0;
    endfor
  endfor
end procedure

```

FIGURE 2.18 Pseudocode describing sequential Gaussian elimination

barriers? Could you use point-to-point or group event synchronization instead of global event synchronization?

- b. Answer the same for the equation solver example.
- 2.9 In the straightforward, loop-based approach to parallelizing Gaussian elimination discussed so far, parallelism is exploited only within an iteration of the outermost, k , loop. Since the pivot element and its row (called the *pivot row*) are effectively broadcast directly to all processes that need it, this is called the *broadcast version*. Gaussian elimination can also be parallelized in a form that is more aggressive in exploiting the available concurrency, even across outer loop iterations. During the k th iteration, the process assigned the pivot row can simply pass the pivot row on to the next process instead of broadcasting it. This process can use the pivot row to update its assigned rows immediately, as well as pass it on to the next process, and so on. As soon as this process has done its computation for the k th iteration of that loop in the sequential program, it can immediately perform its pivot row computation for the $(k + 1)$ th iteration without waiting for all other processes to receive the k th row and perform their work for the k th iteration. It can then pass this $(k + 1)$ th row on to the next process as well, which can use it right away instead of waiting for the entire previous k loop iteration to complete. Multiple k loop iterations are in progress at once; rows are passed down the processor pipeline as soon as they are computed and are computed as soon as the rows needed have arrived through the pipeline. We call this the *pipelined* form of parallelization.
- a. Write a shared address space pseudocode, at a similar level of detail as Figure 2.13, for a version that implements pipelined parallelism at the granularity of individual elements. Show all synchronization necessary. Do you need barriers?

- b. Write a message-passing pseudocode at the level of detail of Figure 2.16 for the pipelined case in part (a). Assume that the only communication primitives you have are synchronous and asynchronous (blocking and nonblocking) sends and receives. Which versions of send and receive would you use, and why wouldn't you choose the others?
 - c. Discuss the trade-offs in programming the loop-based versus pipelined parallelism.
- 2.10 Multicast (sending a message from one process to a named list of other processes) is a useful mechanism for communicating among subsets of processes.
- a. How would you implement the message-passing, interleaved assignment version of Gaussian elimination with multicast rather than broadcast? Make up a multicast primitive, write pseudocode, and compare the programming ease of the two versions.
 - b. Which do you think will perform better and why?
 - c. What group communication primitives other than multicast do you think might be useful for a message-passing system to support? Give examples of computations in which they might be used.

Programming for Performance

The goal of using multiprocessors is to obtain high performance. With a concrete understanding of how the decomposition, assignment, and orchestration of a parallel program are incorporated in the code that runs on the machine, we are ready to examine the key factors that limit parallel performance and how they are addressed in a wide range of problems. We will see how decisions made in different steps of the programming process affect the run-time characteristics presented to the architecture, as well as how the characteristics of the architecture influence programming decisions. Understanding programming techniques and these interdependencies is important not only for parallel software designers but also for architects. Besides understanding parallel programs as workloads for the systems we build, we learn to appreciate hardware/software trade-offs. In particular, we learn which aspects of programmability and performance the architecture can positively impact and which aspects are best left to software. The interdependencies of program and system are more fluid, more complex, and more important to performance in multiprocessors than in uniprocessors; hence, this understanding is critical to our goal of designing high-performance systems that reduce cost and programming effort. We carry it with us throughout the book, starting with concrete guidelines for workload-driven architectural evaluation in Chapter 4.

The space of performance issues and techniques in parallel software is very rich: different goals trade off with one another, and techniques that further one goal may cause us to revisit the techniques used to address another. This is what makes the creation of parallel software so interesting and challenging. As in uniprocessors, most performance issues can be addressed either by algorithmic and programming techniques in software or by architectural techniques or both. The focus of this chapter is on performance issues and software techniques. Architectural techniques, sometimes hinted at here, are the subject of the rest of the book.

Although several interacting performance issues must be considered, they are not dealt with all at once. The process of creating a high-performance program is one of successive refinement. As discussed in Chapter 2, the partitioning steps—decomposition and assignment—are often largely independent of the underlying architecture or programming model and concern themselves with major algorithmic issues that depend only on the inherent properties of the problem. In particular, these steps view the multiprocessor as simply a set of processors that communicate with one another. Their goal is to resolve the tension between balancing the workload across processes, reducing the interprocess communication inherent in the program, and

reducing the extra work needed to compute and manage the partitioning. We focus our attention first on addressing these partitioning issues.

Next, we open up the architecture and examine the new performance issues it raises for the orchestration and mapping steps. Opening up the architecture means recognizing two facts. The first fact is that a multiprocessor is not only a collection of processors but also a collection of memories, which an individual processor can view as an extended memory hierarchy. The management of data in these memory hierarchies can cause more data to be transferred across the network than the inherent communication mandated by the partitioning in the parallel program. The actual communication that occurs therefore depends both on the partitioning and on how the program's access patterns and locality of data reference interact with the organization and management of the extended memory hierarchy. The second fact is that the cost of communication as seen by the processor—and hence the contribution of communication to the execution time of the program—depends not only on the amount of communication but also on how it is structured to interact with the architecture. Section 3.2 discusses the relationship between communication, data locality, and the extended memory hierarchy. Then Section 3.3 examines the software techniques to address the major performance issues in orchestration and mapping: reducing the extra communication by exploiting data locality in the extended memory hierarchy and structuring communication to reduce its cost.

Of course, the architectural interactions and communication costs that we must deal with in orchestration sometimes cause us to go back and revise our partitioning methods, which is an important part of the refinement in parallel programming. Whereas interactions and trade-offs take place among all the performance issues we discuss, this chapter addresses each issue independently as far as possible and identifies trade-offs as they are encountered. Examples are drawn throughout from the four case study applications, and the impact of some individual programming techniques is illustrated through measurements on a cache-coherent machine with physically distributed memory, the Silicon Graphics Origin2000 (which is described in detail in Chapter 8). The equation solver kernel is also carried through the discussion, and performance techniques are applied to it as relevant; by the end of the discussion we will have created a high-performance parallel version of the solver.

As we examine the performance issues, we will develop simple analytical expressions for the speedup of a parallel program and illustrate how each performance issue affects the speedup equation. However, from an architectural perspective, a more concrete way of looking at performance is to examine the different components of execution time as seen by an individual processor in a machine—that is, how much time the processor spends executing instructions, accessing data in the extended memory hierarchy, and waiting for synchronization events to occur. In fact, these components of execution time can be mapped directly to the performance issues that software must address in the steps of creating a parallel program. Examining this view of performance helps us understand very concretely what a parallel execution looks like as a workload presented to the architecture, and the mapping helps us understand how programming techniques can alter this profile. These topics are discussed in Section 3.4.

Once we have studied the performance issues and techniques, we will be ready to understand how to create high-performance parallel versions of real applications—namely, the four case studies. Section 3.5 applies the parallelization process and performance techniques to each case study in turn. It illustrates how the techniques are employed together as well as the range of resulting execution characteristics that are presented to an architecture, reflected in varying profiles of execution time. We will also be ready to consider the implications of realistic applications for trade-offs between the two major lower-level programming models: a shared address space and explicit message passing. The trade-offs are in ease of programming and in performance and are discussed in Section 3.6. Let us begin with the algorithmic performance issues in the decomposition and assignment steps.

3.1 PARTITIONING FOR PERFORMANCE

For these steps, we can view the machine as simply a set of cooperating processors, largely ignoring its programming model and organization. All we need to know at this stage is that communication between processors is expensive. The three primary algorithmic issues are

- *balancing the workload* and reducing the time spent waiting at synchronization events
- *reducing communication*
- *reducing the extra work* done to determine and manage a good assignment

Unfortunately, even the three primary algorithmic goals are at odds with one another and must be traded off. A singular goal of minimizing communication would be satisfied by running the program on a single processor, as long as the necessary data fits in the local memory, but this would yield the ultimate load imbalance. On the other hand, near perfect load balance could be achieved—at a tremendous communication and task management penalty—by making each primitive operation in the program a task and assigning tasks randomly. And in many complex applications, load balance and communication could be improved by spending more time determining a good assignment, which results in extra work. The goal of decomposition and assignment is to achieve a good compromise between these conflicting demands as we see illustrated in the case studies and the equation solver kernel.

3.1.1 Load Balance and Synchronization Wait Time

In its simplest form, balancing the workload means ensuring that every processor does the same amount of work. It extends exposing enough concurrency (which we saw in Chapter 2 when discussing Amdahl's Law) with proper assignment and reduced serialization, and it gives the following simple limit on potential speedup:

$$\text{Speedup}_{\text{problem}}(p) \leq \frac{\text{Sequential Work}}{\max \text{Work on Any Processor}}$$

Work in this context should be interpreted liberally because what matters is not only how many calculations are done but also the time spent doing them, which involves data accesses and communication as well.

In fact, load balancing is a little more complicated than simply equalizing work. Not only should different processors do the same amount of work, they should also be working at the same time. The extreme point would be if the work were evenly divided among processes but only one process were active at a time so there would be no speedup at all! The real goal of load balance is to minimize the time processes spend waiting at synchronization points, including an implicit one at the end of the program. This also involves minimizing the serialization of processes because of either mutual exclusion (waiting to enter critical sections) or dependences. The assignment step should ensure that low serialization is possible, and orchestration should ensure that it happens.

The process of balancing the workload and reducing synchronization wait time consists of four parts:

1. Identifying enough concurrency in decomposition and overcoming Amdahl's Law
2. Deciding how to manage the concurrency (statically or dynamically)
3. Determining the granularity at which to exploit the concurrency
4. Reducing serialization and synchronization cost

This section examines some techniques for each, using examples from the four case studies and other applications as well.

Identifying Enough Concurrency: Data and Function Parallelism

We saw in parallelizing the equation solver kernel that concurrency may be found by examining the loops of a program, by looking more deeply at the fundamental dependences, or by exploiting an understanding of its underlying problem to discover algorithms that afford more concurrency. Parallelizing loops often leads to similar (not necessarily identical) operation sequences or functions being performed on elements of a large data structure, as in the equation solver kernel. This is called *data parallelism* and is a more general form of the parallelism that inspired data parallel architectures discussed in Chapter 1. Computing forces on different particles in Barnes-Hut is another example.

In addition to data parallelism, applications often exhibit *function parallelism* as well: entirely different calculations can be performed concurrently on either the same or different data. Function parallelism is often referred to as control parallelism or task parallelism, though these are overloaded terms. For example, setting up an equation system for the solver in Ocean requires many different computations on ocean cross sections, each using a few cross-sectional grids. Analyzing dependences at the level of entire grids or arrays reveals that several of these computations are independent of one another and can be performed in parallel. Pipelining is another form of function parallelism in which different functions or stages of the pipeline are

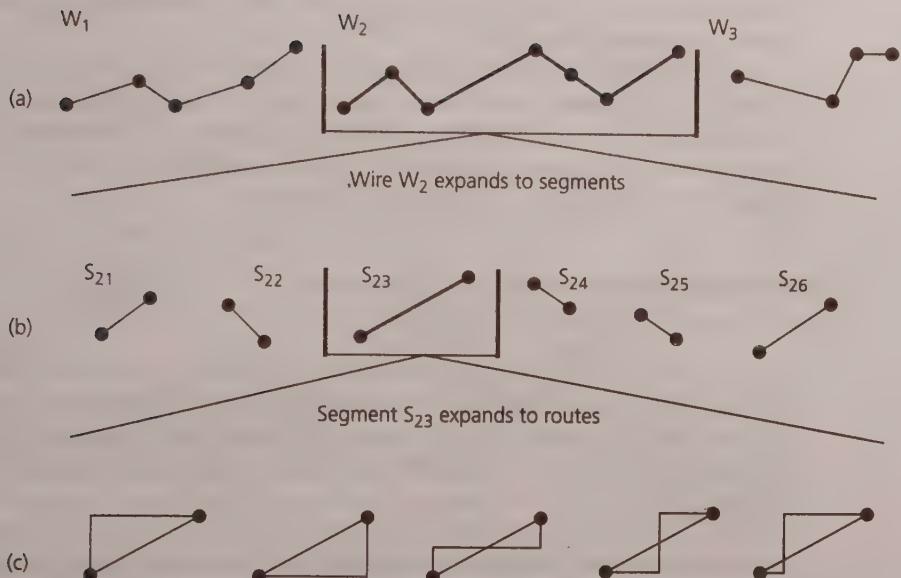


FIGURE 3.1 The three axes of parallelism in a VLSI wire-routing application: (a) wire parallelism; (b) segment parallelism; (c) route parallelism. The filled circles indicate the pins that are connected by wires.

performed concurrently on different data. For example, in encoding a sequence of video frames, each block of each frame passes through several stages: prefiltering, convolution from the time to the frequency domain, quantization, entropy coding, and so on. Pipeline parallelism is available across these stages (for example, a few processes could be assigned to each stage and operate concurrently), as is data parallelism between frames, among blocks in a frame, and within an operation on a block.

Function parallelism and data parallelism are often available together in an application and provide a hierarchy of levels of parallelism from which we must choose (e.g., function parallelism across grid computations and data parallelism within grid computations in Ocean, and the video encoding example). Orthogonal levels of data or function parallelism are found in many other applications as well; for example, applications that route wires in VLSI circuits exhibit parallelism across the wires to be routed, across the two-pin segments within a wire, and across the many routes evaluated for each segment (see Figure 3.1).

The degree of available function parallelism is usually modest and does not grow much with the size of the problem being solved. The degree of data parallelism, on the other hand, usually grows with data set size. Function parallelism is also usually more difficult to exploit in a load-balanced way, since different functions involve different amounts of work and have different scaling characteristics. Most parallel programs that run on large-scale machines are data parallel according to our loose definition of the term, and exploit function parallelism mainly to reduce the amount

of global synchronization required between data parallel computations (as illustrated in Ocean in Section 3.5.1).

By identifying the different types of concurrency available in an application, we often find much more concurrency than we need for load balancing. The next step in decomposition is to restrict the available concurrency by determining the granularity of tasks. However, the choice of task size also depends on how we expect to manage the concurrency, so let us discuss this next.

Determining How to Manage Concurrency: Static versus Dynamic Assignment

A key issue in exploiting concurrency is whether a good load balance can be obtained by a static or predetermined assignment (introduced in Chapter 2) or whether more dynamic means are required. A static assignment is typically an algorithmic mapping of tasks to processes, as in the simple equation solver kernel discussed in the previous chapter. Exactly which tasks (grid points or rows) are assigned to which processes may depend on the problem size, the number of processes, and other parameters, but once it is determined, the assignment does not change again at run time. Since the assignment is predetermined, static techniques do not incur much task management overhead at run time. However, to achieve good load balance, they require that the relative amounts of work in different tasks be adequately predictable or that enough tasks exist to ensure a balanced distribution by virtue of the statistics of large numbers. In addition to the program itself, it is also important that other environmental conditions—such as interference from other applications—not perturb the relationships among processors, thus limiting the robustness of static load balancing.

Dynamic partitioning techniques adapt to load imbalances at run time. They come in two forms. In *semistatic* techniques, the assignment for a phase of computation is determined algorithmically before that phase, but assignments are recomputed periodically to restore load balance based on profiles of the actual workload distribution gathered at run time. For example, we can profile (measure) the work that each task does in one phase and use that as an estimate of the work associated with it the next time that phase is executed. This repartitioning technique is used to assign stars to processes in Barnes-Hut (Section 3.5.2) by using profiles to recompute the assignment between time-steps of the galaxy's evolution. The galaxy evolves slowly, so the workload distribution among stars does not change much between successive time-steps. Figure 3.2(a) illustrates the advantage of semistatic partitioning over a static assignment of particles to processors, for a 512-K particle execution measured on the Origin2000, despite the cost of periodic repartitioning. It is clear that the performance difference grows with the number of processors used.

The second dynamic technique, *dynamic tasking*, is used to handle cases in which either the work distribution or the system environment is too unpredictable even to

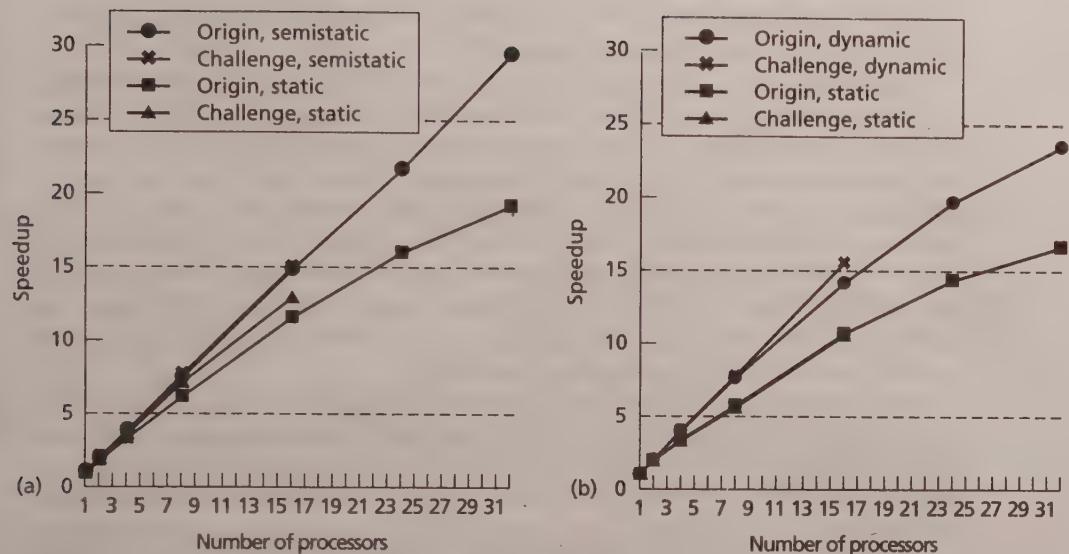


FIGURE 3.2 Illustration of the performance impact of dynamic partitioning for load balance.

The graph in (a) shows the speedups of the Barnes-Hut application with and without semistatic partitioning, and the graph in (b) shows the speedups of Raytrace with and without dynamic tasking. Even in these applications that have a lot of parallelism, dynamic partitioning is important for improving load balance over static partitioning.

periodically recompute a load-balanced assignment.¹ For example, in Raytrace the work associated with each ray is impossible to predict. Even if the rendering is repeated from different viewpoints, the change in viewpoints may not be gradual. The dynamic tasking approach divides the computation into tasks and maintains a pool of available tasks (in Raytrace a task may be a ray or a set of rays). Each process repeatedly takes a task from the pool and executes it—possibly inserting new tasks into the pool—until no tasks are left. Of course, the management of the task pool must preserve the dependences among tasks—for example, by inserting a task only when it is ready for execution. Since dynamic tasking is widely used, let us look at some specific techniques to implement the task pool. Figure 3.2(b) illustrates the advantage of dynamic tasking over a static assignment of rays to processors in the

1. The applicability of static or semistatic assignment depends not only on the computational properties of the program but also on its interactions with the memory and communication systems and on the predictability of the execution environment. For example, differences in memory or communication stall time (due to cache misses, page faults, or contention) can cause imbalances observed at synchronization points even when the workload is computationally load balanced. Static assignment also may not be appropriate for time-shared or heterogeneous systems.

Raytrace application, for a data set consisting of number of balls arranged like a bunch of grapes, measured on the Origin2000.

A simple example of dynamic tasking in a shared address space is *self-scheduling* of a parallel loop. The loop counter is a shared variable accessed by all the processes that execute iterations of the loop. Processes obtain a loop iteration by incrementing the counter atomically; they repeatedly execute an iteration and access the counter again until no iterations remain. The task size can be increased by taking multiple iterations at a time, that is, adding a value larger than one to the shared loop counter. However, this can increase load imbalance. In guided *self-scheduling* (Aiken and Nikolau 1988), processes start by taking large chunks and taper down the chunk size as the loop progresses, hoping to reduce the number of accesses to the shared counter without compromising load balance.

More general dynamic task pools are usually implemented by a collection of queues into which tasks are inserted and from which tasks are removed and executed by processes. This may be a single centralized queue or a set of distributed queues, typically one per process, as shown in Figure 3.3. A *centralized queue* is simpler but has the disadvantage that every process accesses the same task queue, potentially increasing communication and causing processors to contend for queue access. Modifications to the queue (enqueueing or dequeuing tasks) must be mutually exclusive, further increasing contention and causing serialization. Unless tasks are large, and therefore queue accesses are few relative to computation, a centralized queue can quickly become a performance bottleneck as the number of processors increases.

With *distributed queues*, every process is initially assigned a set of tasks in its local queue. This initial assignment may be done intelligently to reduce interprocess communication, thus providing more control than self-scheduling and centralized queues. A process removes and executes tasks from its local queue as far as possible. If it creates tasks, it inserts them in its local queue. When no more tasks are in its local queue, it queries other processes' queues to obtain tasks from them, a mechanism known as *task stealing*. Because task stealing implies communication and can generate contention, several interesting issues arise in implementing stealing: for example, how to minimize stealing, whom to steal from, how many and which tasks to steal at a time, and so on. Stealing also introduces the important issue of *termination detection*: how do we decide when to stop searching for tasks to steal and assume that they're all done, given that tasks generate other tasks that are dynamically inserted in the queues? Simple heuristic solutions to this problem work well in practice, although a robust solution can be quite subtle and communication intensive (Dijkstra and Sholten 1968; Chandy and Misra 1988). Task queues are used both in a shared address space, where the queues are shared data structures that are manipulated using locks, and with explicit message passing, where the owners of queues service requests for them.

Although dynamic techniques generally provide good load balancing despite unpredictability or environmental conditions, they make the management of parallelism more expensive. Dynamic tasking techniques also compromise the explicit control over which tasks are executed by which processes, thus potentially increas-

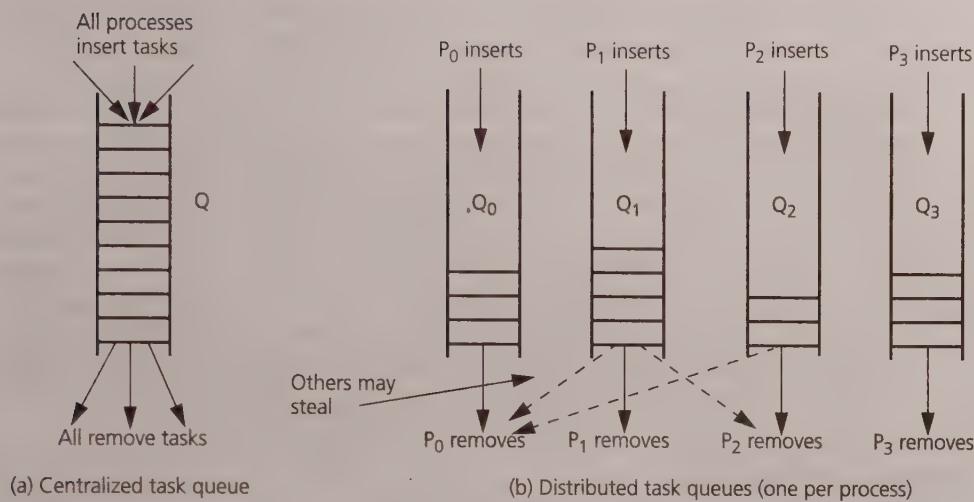


FIGURE 3.3 Implementing a dynamic task pool with a system of task queues

ing communication and compromising data locality. Static techniques are therefore usually preferable when they can provide good load balance for an application and environment.

Determining the Granularity of Tasks

If no load imbalances occur due to dependences among tasks (for example, if all tasks are ready to be executed at the beginning of a phase of computation), then the maximum load imbalance possible with a task-queue strategy is equal to the granularity of the largest task. By *task granularity*, we mean the amount of work associated with a task, which is measured by the number of instructions or, more appropriately, the execution time. The general rule for choosing a granularity at which to actually exploit concurrency is that fine-grained or small tasks have the potential for better load balance (more tasks to divide among processes and hence more concurrency), but they lead to higher task management overhead, more contention, and more interprocessor communication than coarse-grained or large tasks. Let us see why, first in the context of dynamic task queuing where the definitions and trade-offs are clearer.

Task Granularity with Dynamic Task Queuing Here, a task is explicitly defined as an entry placed on a task queue, so task granularity is the work associated with such an entry. The larger task management (queue manipulation) overhead with small tasks is clear. At least with a centralized queue, the more frequent need for queue access generally leads to greater contention as well. Finally, breaking up a task into

two smaller tasks might cause the two tasks to be executed on different processors, thus increasing communication if the tasks access the same logically shared data.

Task Granularity with Static Assignment With static assignment, tasks are not explicit in the program, so it is less clear what should be called a task or a unit of concurrency. For example, in the equation solver, is a task a group of rows, a single row, or an individual element? We can define a task as the largest unit of work such that even if the assignment of tasks to processes is changed, the code that implements a task need not change. With static assignment, task size has a much smaller effect on task management overhead compared to dynamic task queuing since there are no queue accesses. Communication and contention are affected by the assignment of tasks to processors, not their size. The major impact of task size is usually on load imbalance and on exploiting data locality in processor caches.

Reducing Serialization

Finally, to reduce serialization at synchronization points, whether it is due to mutual exclusion or dependences among tasks, we must be careful about how we assign tasks as well as how we orchestrate synchronization and schedule tasks. For event synchronization, an example of excessive serialization is the use of more conservative synchronization than necessary, such as barriers instead of point-to-point or group synchronization. Even if point-to-point synchronization is used, it may preserve data dependences at a coarser grain than is required; for example, a process waits for another to produce a whole row of a matrix when the actual dependences are at the level of individual matrix elements. However, finer-grained synchronization is often more complex to program; it also implies the execution of more synchronization operations (say, one per word rather than one per larger data structure), the overhead of which may turn out to be more expensive than the savings in serialization. As usual, trade-offs abound.

For mutual exclusion, we can reduce serialization by using separate locks for separate data items and making the critical sections protected by locks smaller and less frequent if possible. Consider the former technique. In a database application, we may want to lock when we update certain fields of records that are assigned to different processes. The question is how to organize the locking. Should we use one lock per process, one per record, or one per field? The finer the granularity, the lower the contention, but the greater the space overhead and the less frequent the reuse of locks. An intermediate solution is to use a fixed number of locks and share them among records using a simple hashing function from records to locks. Another way to reduce serialization is to stagger the critical sections in time, that is, to arrange the computation so that multiple processes do not try to access the same lock at the same time.

Implementing task queues provides an interesting example of making critical sections smaller and less frequent. Suppose each process adds a task to a queue, then searches the queue for another task with a particular characteristic, and then removes this latter task from the queue. The task insertion and deletion may need to

be mutually exclusive—or may not if they are done at different ends of the queue—but the searching of the queue does not. Thus, instead of using a single critical section for the whole sequence of operations, we can break it up into two critical sections (insertion and deletion) and use code that is not mutually exclusive to search the list in between.

More generally, checking (reading) the state of a protected data structure usually does not have to be done with mutual exclusion; only modifying the data structure does. If the common case is to check but not to modify, as for the tasks we search through in the task queue, we can check without locking and, only if the check returns the appropriate condition, then lock and recheck within the critical section (to ensure the state hasn't changed) before modifying. In addition, instead of using a single lock for the entire queue, we can use a lock per queue element so that elements in different parts of the queue can be inserted or deleted in parallel (without serialization). As with event synchronization, the correct trade-offs in performance and programming ease depend on the costs and benefits of the choices on a system.

We can extend our simple limit on speedup to reflect both load imbalance and time spent waiting at synchronization points as follows, where *max* in the denominator is the maximum over all processes:

$$\text{Speedup}_{\text{problem}}(p) \leq \frac{\text{Sequential Work}}{\max(\text{Work} + \text{Synch Wait Time})}$$

In general, the different aspects of balancing the workload are the responsibility of software. An architecture cannot do very much about a program that does not have enough concurrency or is not load balanced. However, an architecture can help in some ways. First, it can provide efficient support for load-balancing techniques, such as task stealing, that are used widely by parallel software (applications, libraries, and operating systems). An access to a remote task queue for stealing is usually a probe or query, involving a small amount of data transfer and perhaps mutual exclusion. The more efficient the support for fine-grained communication and for low-overhead, mutually exclusive access to data, the smaller we can make our tasks and thus improve load balance. Second, the architecture can make it easy to name or access the logically shared data that a stolen task needs. Third, the architecture can provide efficient support for point-to-point synchronization, making it more attractive to use this form of synchronization instead of conservative barriers and hence allowing better load balance to be achieved.

3.1.2 Reducing Inherent Communication

Load balancing by itself is conceptually quite easy as long as the application affords enough concurrency: we can simply make tasks small and use dynamic tasking. Perhaps the most important performance goal to be traded off with load balance is reducing interprocessor communication. Decomposing a problem into multiple tasks usually means that communication will be required among tasks. If these tasks are assigned to different processes, we incur communication among processes and

hence processors. The focus in this section is on reducing communication that is *inherent* to the parallel program (i.e., one process produces data values that another needs) while still preserving load balance, thus retaining the view of the machine as a set of cooperating processors. However, in a real system communication occurs for other reasons, as Section 3.2 shows.

The impact of communication is best estimated not by the absolute amount of communication but by a quantity called the *communication-to-computation ratio*. This is defined as the amount of communication (in bytes, say) divided by the computation time (or because time is influenced by many factors, by the number of instructions executed). For example, a gigabyte of communication has a much greater impact on the execution time and communication bandwidth requirements of an application if the time required for the application to execute is 1 second than if it is 1 hour! The communication-to-computation ratio may be computed as a per-process number or accumulated over all processes.

The inherent communication-to-computation ratio is primarily controlled by the assignment of tasks to processes. To reduce communication, we should try to ensure that tasks accessing the same data or requiring frequent communication with one another are assigned to the same process. For example, in a database application, communication would be reduced if queries and updates that access the same database records are assigned to the same process.

One partitioning principle that has worked very well in practice for load balancing and inherent communication is *domain decomposition*. It was initially used in data parallel scientific computations such as Ocean but has since been found applicable to many other areas. If the data set on which the application operates can be viewed as a physical domain, then it is often the case that a point in the domain requires information either directly from only a small localized region around that point or from a longer range, with the requirements falling off with increasing distance from the point. We saw an example of the latter in Barnes-Hut. For the former, consider a video application in which algorithms for motion estimation in video encoding and decoding examine only the areas of a scene that are close to the current pixel; similarly, a point in the equation solver kernel needs to access only its four nearest-neighbor points directly. The goal of partitioning in these cases is to give every process a contiguous region of the domain, while of course retaining load balance, and to shape the domain so that most of the process's information requirements are satisfied within its assigned partition. As Figure 3.4 shows, in many such cases the communication requirements for a process grow proportionally to the size of a partition's boundary, whereas computation grows proportionally to the size of its entire partition. The communication-to-computation ratio is thus a surface-area-to-volume ratio in three dimensions and a perimeter-to-area ratio in two dimensions. It can be reduced by either increasing the data set size (n^2 in the figure) or reducing the number of processors (p).

Of course, the ideal shape for partitions in a domain decomposition is application dependent, depending primarily on the information requirements of and work associated with the points in the domain. For the equation solver kernel, in Chapter 2 we

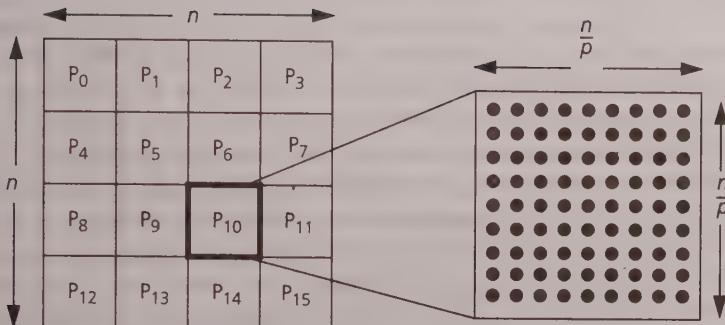


FIGURE 3.4 The perimeter-to-area relationship of communication to computation in a two-dimensional domain decomposition. The example shown is for an algorithm with localized, nearest-neighbor information exchange like the simple equation solver kernel. Every point on the grid needs information from its four nearest neighbors. Thus, the darker internal points in processor P_{10} 's partition do not need to communicate directly with any points outside the partition. Computation for processor P_{10} is thus proportional to the sum of all n^2/p points, whereas communication is proportional to the number of lighter boundary points, which is $4n/\sqrt{p}$.

chose to partition the grid into blocks of contiguous rows. Figure 3.5 shows that partitioning the grid into squarelike subgrids leads to a lower inherent communication-to-computation ratio. The impact becomes greater as the number of processors increases relative to the grid size. We shall therefore carry forward this partitioning into square subgrids (or simply “subgrids”) as we continue to discuss performance. As a simple exercise, think about what the communication-to-computation ratio would be if we assigned rows to processes in an interleaved or cyclic fashion instead (row i assigned to process $i \bmod nprocs$).

How do we find a suitable domain decomposition that is load balanced and also keeps communication low? This can be accomplished statically or semistatically, depending on the nature and predictability of the computation:

- *Statically, by inspection*, as in the equation solver kernel and in Ocean. This requires predictability and usually leads to regularly shaped partitions, as in Figures 3.4 and 3.5.
- *Statically, by analysis*. The computation and communication characteristics may depend not only on the size of the input but also on the structure of the input presented to the program at run time, thus requiring an analysis of the input. However, the partitioning may need to be done only once after the input analysis—before the actual computation starts—so we still consider it static. Partitioning sparse matrix computations used in aerospace and automobile simulations is an example: the matrix structure is fixed but is highly irregular and requires sophisticated graph partitioning. Another example is Data Mining. Here, we may divide the database of transactions statically among processors, but a balanced assignment of itemsets to processes requires some analysis

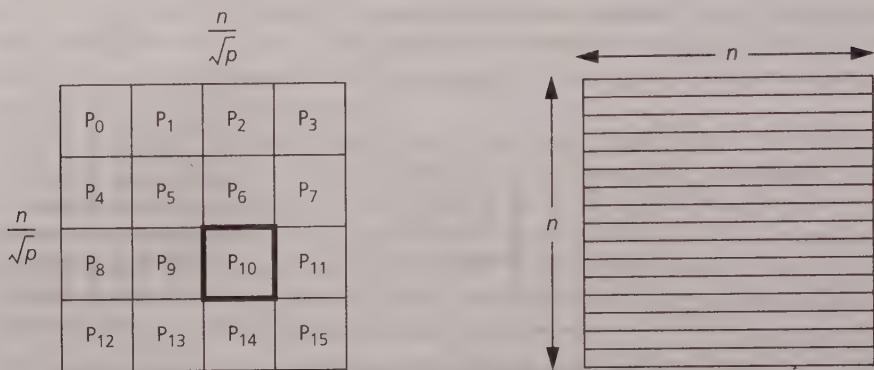


FIGURE 3.5 Choosing among domain decompositions for a simple nearest-neighbor computation on a regular two-dimensional grid. Since the work per grid point is uniform, equally sized partitions yield good load balance. But we still have choices. We might partition the elements of the grid into either strips of contiguous rows (right) or block-structured partitions that are as close to square as possible (left). The perimeter-to-area (and hence communication-to-computation) ratio in the block decomposition case is

$$\frac{4 \times n / \sqrt{p}}{n^2 / p} \text{ or } \frac{4 \times \sqrt{p}}{n}$$

whereas that in strip decomposition is $\frac{2 \times n}{n^2 / p}$ or $\frac{2 \times p}{n}$.

As p increases, block decomposition incurs less inherent communication for the same computation than strip decomposition.

since the work associated with different itemsets is not equal. A simple static assignment of itemsets and the database by inspection keeps communication low but does not provide load balance.

- *Semistatically, with periodic repartitioning.* This was discussed earlier for applications like Barnes-Hut whose characteristics change slowly with time. Domain decomposition is still important to reduce communication, as we see in the profiling-based Barnes-Hut case study in Section 3.5.2.
- *Statically or semistatically, with dynamic task stealing.* Even when the computation is highly unpredictable and dynamic task stealing must be used, domain decomposition may be useful in initially assigning tasks to processes. Raytrace is an example. Here there are two domains: the three-dimensional scene being rendered and the two-dimensional image plane. Since the natural tasks are rays shot through the image plane, it is much easier to manage domain decomposition of that plane than of the scene itself. We partition the image domain much like the grid in the equation solver kernel (Figure 3.4), with image pixels corresponding to grid points, and initially assign rays to the corresponding processes. This is useful because rays shot through adjacent pixels tend to access much of the same scene data. Processes then steal rays (pixels) or groups of rays dynamically for load balancing.

Of course, partitioning into a contiguous subdomain per processor is not always appropriate for high performance in all applications, as illustrated by the Gaussian elimination example in Exercise 3.9. Even Raytrace may benefit from dividing the image into more blocks than there are processors and assigning blocks to processors in an interleaved manner, trading off increased communication for better initial load balance. Different phases of the same application may also call for different partitioning. The range of techniques is very large, but common principles like domain decomposition can be found. For example, even when stealing tasks for load balancing in very dynamic applications, we can reduce communication by searching other queues in the same order every time or by preferentially stealing large tasks or several tasks at once to reduce the number of times we have to access nonlocal queues.

In addition to reducing communication volume, it is also important to keep communication (not just computation) balanced among processors. Since communication is expensive, imbalances in communication can translate directly to imbalances in execution time among processors. Overall, whether trade-offs in partitioning should be resolved in favor of load balance or communication volume depends on the cost of communication on a given system. Including communication as an explicit performance cost refines our basic speedup limit to

$$\text{Speedup}_{\text{problem}}(p) \leq \frac{\text{Sequential Work}}{\max(\text{Work} + \text{Synch Wait Time} + \text{Comm Cost})}$$

Compared to the previous expression, this expression separates communication from work, which now includes instructions executed plus local data access costs.

The amount of communication in parallel programs clearly has important implications for architecture. In fact, architects examine the needs of applications to determine what communication latencies and bandwidths are worth spending extra money for (see Exercise 3.14); for example, the bandwidth provided by a machine can usually be increased by throwing hardware (and hence money) at the problem, but this is only worthwhile if applications will exercise the increased bandwidth. As architects, we assume that the programs delivered to us are reasonable in their load balance and their communication demands, and we strive to make them perform better by providing the necessary support. Let us now examine the last of the algorithmic issues that we can resolve in partitioning itself without addressing the underlying architecture.

3.1.3 Reducing the Extra Work

The preceding discussion of domain decomposition suggests that when a computation is irregular, computing a good assignment that both provides load balance and reduces communication can be quite expensive. This extra work is not required in a sequential execution and is an overhead of parallelism. Consider the sparse matrix example that was discussed previously to illustrate static partitioning by analysis. The sparse matrix can be represented as a graph, such that each node represents a row or column of the matrix and an edge exists between two nodes i and j if the matrix entry (i,j) is nonzero. The goal in partitioning is to assign each process a set

of nodes such that the computation is load balanced and the number of edges that cross partition boundaries is minimized. Many clever partitioning techniques have been developed, but the ones that result in a better balance between load balance and communication require more time to partition the graph. We see this illustrated in the Barnes-Hut case study later in this chapter.

In addition to partitioning, another common source of extra work is redundant computation: multiple processes computing data values redundantly rather than having one process compute them and communicate them to the others, which may be a favorable trade-off when the cost of communication is high. Examples include all processes computing their own copy of the same shading table in computer graphics applications or of trigonometric tables in scientific computations. If the redundant computation can be performed while the processor is otherwise idle due to load imbalance, its cost can be hidden.

Finally, many aspects of orchestrating parallel programs involve extra work as well, such as creating processes, managing dynamic tasking, distributing code and data throughout the machine, executing synchronization operations and parallelism control instructions, structuring communication appropriately for a machine, and packing and unpacking data to and from communication messages. For example, the high cost of creating processes is what causes us to create them once up front and have them execute tasks until the program terminates, rather than creating and terminating processes as parallel sections of code are encountered and exited by a single main thread of computation (a *fork-join* approach, which is sometimes used with lightweight threads instead of processes). For example, in the Data Mining case study (Section 3.5.4), substantial extra work done to transform the database pays off in reducing communication, synchronization, and expensive input/output activity.

The trade-offs between extra work, load balance, and communication must be considered carefully when making partitioning decisions. The architecture can help reduce the need for extra work by making communication and task management more efficient. Based only on these algorithmic partitioning issues, the speedup limit can now be refined to

$$\text{Speedup}_{\text{problem}}(p) \leq \frac{\text{Sequential Work}}{\max (\text{Work} + \text{Synch Wait Time} + \text{Comm Cost} + \text{Extra Work})} \quad (3.1)$$

3.1.4 Summary

The analysis of parallel algorithm performance requires a characterization of a multiprocessor and a characterization of the parallel algorithm. Historically, the analysis of parallel algorithms has focused on algorithmic aspects like partitioning and mapping to network topologies and has not taken other architectural interactions into account. In fact, the most common model used to characterize a multiprocessor for algorithm analysis has been the Parallel Random Access Memory (PRAM) model (Fortune and Wyllie 1978). In its most basic form, the PRAM model

assumes that data access is free, regardless of whether it is local or involves communication. That is, communication cost is zero in the speedup expression of Equation 3.1, and work is treated simply as instructions executed:

$$\text{Speedup-PRAM}_{\text{problem}}(p) \leq \frac{\text{Sequential Instructions}}{\max (\text{Instr} + \text{Synch Wait Time} + \text{Extra Instr})} \quad (3.2)$$

A natural way to think of a PRAM model is as a shared address space machine in which all data access is free. The performance factors that matter in parallel algorithm analysis using this model are load balance (including serialization) and extra work. The goal of algorithm development for PRAMs is to expose enough concurrency so the workload may be well balanced without needing too much extra work.

While the PRAM model is useful in discovering the concurrency available in an algorithm, which is the first step in parallelization, it is clearly unrealistic for modeling performance on real parallel systems. This is because communication, which it ignores, can easily dominate the cost of a parallel execution in modern systems, and imbalances in communication cost can dominate imbalances in instructions executed. In fact, analyzing algorithms while ignoring communication can easily lead to a poor choice of decomposition and assignment, to say nothing of orchestration. More recent models have been developed to include communication costs as explicit parameters that algorithm designers can use (Valiant 1990; Culler et al. 1993). We return to this issue after we have a better understanding of communication costs.

The treatment of communication costs in this section is simplified in two respects relative to real systems. First, communication inherent to the parallel program and its partitioning is not the only form of communication that is important: substantial noninherent or *artifactual* communication may occur that is caused by interactions of the program with the architecture on which it runs. Thus, we have not yet modeled the amount of communication generated by a parallel program satisfactorily. Second, the communication cost term in Equation 3.1 is determined not only by the amount of communication caused, whether inherent or artifactual, but also by the structure of the communication in the program and how it interacts with the costs of the basic communication operations in the machine. Both artifactual communication and communication structure are important performance issues that are usually addressed in the orchestration step since they are architecture dependent. To understand them we first need a deeper understanding of some critical interactions of parallel architectures with parallel software.

3.2

DATA ACCESS AND COMMUNICATION IN A MULTIMEMORY SYSTEM

In our discussion of partitioning, we have viewed a multiprocessor as a collection of cooperating processors. However, multiprocessor systems are also multimemory, multicache systems, and the role of these components is essential to performance. The role is essential regardless of programming model, though the latter may influence the nature of the specific performance trade-offs. Our discussion turns to the

remaining performance issues for parallel programs, which are primarily concerned with accessing data in this multimemory system. It is useful for us to now take a different view of a multiprocessor.

3.2.1 A Multiprocessor as an Extended Memory Hierarchy

From an individual processor's perspective, we can view all the memory of the machine, including the caches of other processors, as forming levels of an extended memory hierarchy. The communication architecture glues together the parts of the hierarchy that are on different nodes. In a uniprocessor system, consider how interactions with different levels of the memory hierarchy (e.g., cache size, associativity, block size) can cause some accesses to be much faster than others and can also cause the transfer of more data between levels than is inherently necessary for the program. Similarly, in multiprocessors, interactions with the organization of the extended memory hierarchy can cause more communication (transfer of data across the network) than is inherently necessary to satisfy the processes in a parallel program. Since communication is expensive, it is particularly important that we exploit data locality in the extended hierarchy, both to improve node performance and to reduce the extra communication between nodes.

Even in uniprocessor systems, a processor's performance depends heavily on the performance of the memory hierarchy. Cache effects are so important that it hardly makes sense to talk about performance without taking caches into account. We can look at the performance of a system in terms of the time needed to complete a program, which has two components: the time the processor is busy executing instructions and the time it spends waiting for data from the memory system. (Input/output activity can be grouped with data access or treated separately.)

$$\text{Time}_{\text{prog}}(1) = \text{Busy}(1) + \text{Data Access}(1) \quad (3.3)$$

As architects, we often normalize this formula by dividing each term by the number of instructions executed and measuring time in clock cycles. We then have a convenient, machine-oriented metric of performance, cycles per instruction (CPI), which is composed of an ideal CPI plus the average number of data access stall cycles per instruction. On a modern microprocessor capable of issuing, say, four instructions per cycle, dependences within the program might limit the average issue rate to 2.5 instructions per cycle, or an ideal CPI of 0.4. If only 1% of these instructions causes a cache miss, and a cache miss causes the processor to stall for 80 cycles on average, then these stalls will account for an additional 0.8 cycles per instruction. The processor will be busy doing "useful" work only one-third of its time! Of course, the other two-thirds of the time is in fact useful: it is the time spent communicating with memory to access data. Recognizing this data access cost, we may elect to optimize either the program or the machine to perform the data access more efficiently. For example, we may change the program data layout to enhance temporal or spatial locality, or we might provide a bigger cache or mechanisms to tolerate latency.

In multiprocessors, an idealized view of this extended memory hierarchy would be local cache hierarchies connected to a single centralized memory at the next level. In reality, the picture is a bit more complex. Even on machines with centralized shared memories, beyond the local caches are a multibanked memory as well as the caches of other processors. With physically distributed memories, a part of the main memory too is local, a larger part is remote, and what is remote to one processor is local to another.

Differences in programming models reflect a difference in how certain levels of the hierarchy are managed. We take for granted that the registers in the processor are managed by the compiler. We also take for granted that the first couple of levels of caches are managed transparently by the hardware. In the shared address space model, data movement between a remote node and the local node is managed transparently to the user program as well. The message-passing model has this movement managed explicitly by the program. Regardless of the management, levels of the hierarchy that are closer to the processor provide higher bandwidth and lower latency access to data. Here too we can improve data access performance either by improving the architecture of the extended memory hierarchy or by improving the locality in the program.

Exploiting locality exposes a trade-off with parallelism similar to reducing communication. Parallelism may cause more processors to access the same data and hence move that data toward each of themselves, whereas each individual processor desires that its own data stays close to it. A high-performance parallel program needs to obtain performance from each individual processor (by exploiting locality in the extended memory hierarchy) in addition to being well parallelized.

3.2.2 Artifactual Communication in the Extended Memory Hierarchy

Data accesses that are not satisfied in the local (on-node) portion of the extended memory hierarchy generate communication. Inherent communication can be seen as part of this: the data moves from one processor through the memory hierarchy to another processor, regardless of whether it does this through explicit messages or reads and writes. However, the amount of communication that occurs in an execution of the program is usually greater than the inherent interprocess communication in the parallel algorithm. The additional communication is an *artifact* of how the program is actually implemented and how it interacts with the machine's extended memory hierarchy. There are many sources of this artifactual communication:

- *Poor allocation of data.* Data accessed by one node may happen to be allocated in the local memory of another. Accesses to remote data involve communication even if the data is not modified by other nodes. Such transfer can be eliminated by a better assignment or better distribution of data or reduced by replicating the data locally when it is accessed.
- *Unnecessary data in a transfer.* More data than needed may be communicated in a transfer. For example, a receiver may not use all the data in a message since it may have been easier for the sender to send extra data conservatively

than to determine exactly what to send. Similarly, if data is transferred implicitly in units larger than a word (e.g., cache blocks), part of the block may not be used by the requester. This artifactual communication can be eliminated with smaller transfers.

- *Unnecessary transfers due to other system granularities.* In cache-coherent machines, data is typically kept coherent at a granularity larger than a single word, which may lead to extra communication to keep data coherent, as we shall see in later chapters.
- *Redundant communication of data.* Data may be communicated multiple times (for example, every time the value of the data changes), but only the last value may actually be used. On the other hand, data may be communicated to a process that already has the latest values, again because it was too difficult to determine this.
- *Finite replication capacity.* Communicated data is usually replicated locally to avoid repeated communication when the data is accessed again by the processor. However, the capacity for replication on a node is finite—whether it be in the cache or the main memory—so data that has already been communicated from process A to process B may be replaced from B's local memory system and hence need to be transferred again even if it has not since been modified by A.

In contrast, inherent communication is what occurs given unlimited capacity for local replication, transfers as small as would be required by the program, and perfect knowledge of what logically shared data has been updated or already transferred. We will understand the sources of artifactual communication better when we get deeper into architecture. Let us look a little further at the last source of artifactual communication—finite replication capacity—which has particularly far-reaching consequences.

3.2.3 Artifactual Communication and Replication: The Working Set Perspective

The relationship between finite replication capacity and artifactual communication is quite fundamental in parallel systems, just like the relationship between cache size and memory traffic in uniprocessors; it is almost inappropriate to speak of the amount of communication without reference to replication capacity. The extended memory hierarchy perspective is useful in viewing this relationship. We may view our generic multiprocessor as a memory hierarchy with three levels: local cache is inexpensive to access, local memory is more expensive, and any remote memory is much more expensive. We can think of any level as a cache whether it is actually managed like a hardware cache or managed by system or application software. We can then classify the “misses” at any level, which generate traffic to the next level, just as we do for uniprocessors. A fraction of the traffic at any level results from *cold-start misses*, resulting from the first time data is accessed by the processor. This component, also called compulsory traffic in uniprocessors, is independent of cache size. Such cold-start misses diminish in importance as programs run longer. Then there is

traffic due to *capacity* misses, which clearly decrease with increases in cache size. A third fraction of traffic may be *conflict* misses, which are reduced by greater associativity, a greater number of blocks, or changing the data access pattern. These three types of misses or traffic are called the three C's in uniprocessor architecture—cold start (or compulsory), capacity, and conflict. The new form of traffic in multiprocessors is a fourth C, a *communication* miss, caused by the inherent communication between processors or by some of the sources of artifactual communication discussed previously. Like cold-start misses, communication misses do not diminish with cache size. Each of these components of traffic may be helped or hurt by large granularities of data transfer, depending on spatial locality.

If we were to determine the traffic for a parallel program that results from each type of miss at a given level of the hierarchy as the replication capacity (i.e., the cache size) at that level is increased, we could expect to obtain a curve such as the one shown in Figure 3.6. The curve has a small number of knees, or points of inflection. These knees correspond to the *working sets* of the algorithm relevant to that level of the hierarchy.² For the first-level cache, they are the working sets of the algorithm itself; for others, they depend on how references have been filtered by other levels of the hierarchy and on how the levels are managed. We speak of this curve for a first-level cache (assumed fully associative with a one-word block size) as the *working set curve* for the algorithm.

Traffic resulting from any of these types of misses may cause communication across the machine's interconnection network, for example, if the backing storage happens to be in a remote node. Similarly, any type of miss may contribute to local traffic and local data access cost if the backing storage happens to be local. Thus, we might expect that many of the techniques used to reduce artifactual communication are similar to those used to exploit locality in uniprocessors. With processes running on different processors, inherent communication misses almost always generate actual communication in the machine (except if the data needed has become local in the meanwhile, as we shall see). These misses can only be reduced by changing the logical sharing patterns in the algorithm. In addition, we are strongly motivated to reduce the artifactual communication that arises either because of transfer size or limited replication capacity, which we can do by exploiting spatial and temporal locality in a process's data accesses in the extended hierarchy. Changing the assignment and orchestration can dramatically change locality characteristics, including the shape of the working set curve.

Finally, for a given amount of communication, its cost as seen by the processor is also affected by how the communication is structured. By "structure," we mean whether messages are large or small, how bursty the communication is, whether

2. The working set model of program behavior (Denning 1968) is based on the temporal locality exhibited by the data referencing patterns of programs. Under this model, a program (or a process in a parallel program) has a set of data that it reuses substantially for a period of time before moving on to other data. The shifts between one set of data and another may be abrupt or gradual. In either case, there is at most times a "working set" of data that a processor should be able to maintain in a fast level of the memory hierarchy in order to use that level effectively.

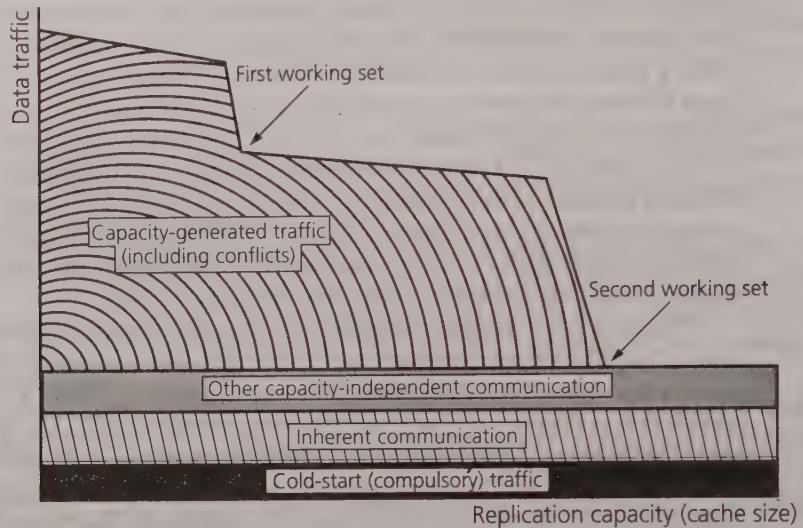


FIGURE 3.6 The data traffic between a cache (replication store) and the rest of the system and the components of the data traffic as a function of cache size. The points of inflection in the total traffic curve indicate the working sets of the program.

communication cost can be overlapped with other computation or communication (all of which are addressed in the orchestration step), and how well the communication patterns match the topology of the interconnection network, which is addressed in the mapping step. Reducing the amount of communication—*inherent* or *artifactual*—is important because it reduces the demand placed on both the system and the programmer to reduce communication cost. Now that we understand the machine as an extended hierarchy and the major issues this raises, let us see how to address these architecture-related performance issues in software—that is, how to program for performance once partitioning issues are resolved.

3.3 ORCHESTRATION FOR PERFORMANCE

We begin by discussing how we might exploit temporal and spatial locality to reduce the amount of artifactual communication and then move on to structuring communication—*inherent* or *artifactual*—to reduce its cost.

3.3.1 Reducing Artifactual Communication

In the message-passing model, both communication and replication are explicit, so even artifactual communication is explicitly coded in program messages. In a shared address space, artifactual communication is more interesting architecturally since it occurs transparently due to interactions between the program and the machine orga-

nization. Of particular interest are the finite cache size and the granularities at which data is allocated, communicated, and kept coherent. We therefore use a shared address space to illustrate issues in exploiting locality, which is done both to improve node performance and to reduce artifactual communication.

Exploiting Temporal Locality

A program is said to exhibit temporal locality if it tends to access the same memory locations repeatedly in a short time frame. Given a memory hierarchy, the goal in exploiting temporal locality is to structure an algorithm so that its working sets map well to the sizes of the different levels of the hierarchy. For a programmer, this typically means keeping working sets small, yet not so small as to lose performance for other reasons. Working sets can be reduced by several techniques. One is the same technique that reduces inherent communication—assigning tasks that tend to access the same data to the same process—which further illustrates the relationship between communication and locality. Once assignment is done, a process's assigned computation can be organized so that tasks that access the same data are scheduled close to one another in time and so that we reuse a set of data as much as possible before moving on to other data, rather than moving back and forth between sections of data.

When multiple data structures are accessed in the same phase of a computation, we must decide which are the most important candidates for exploiting temporal locality. Since communication is more expensive than local access, we might prefer to exploit temporal locality on nonlocal rather than local data. Consider a database application in which a process wants to compare all its records of a certain type with all the records of other processes. There are two choices here: (1) for each of its own records, the process can sweep through all other (nonlocal) records and compare and (2) for each nonlocal record, the process can sweep through its own records and compare. The latter exploits temporal locality on nonlocal data and is therefore likely to yield better overall performance. Example 3.1 discusses temporal locality for the equation solver kernel.

EXAMPLE 3.1 To what extent is temporal locality exploited in the equation solver kernel? How might the temporal locality be increased?

Answer The equation solver kernel traverses only a single data structure. A typical grid element in the interior of a process's partition is accessed at least five times by that process during each sweep: at least once to compute its own new value and once each to compute the new values of its four nearest neighbors. If a process sweeps through its partition of the grid in row-major order (i.e., row by row and left to right within each row, as in Figure 3.7(a)), then reuse of $A[i,j]$ is guaranteed across the updates of the three elements in the same row whose updates touch it: $A[i,j-1]$, $A[i,j]$, and $A[i,j+1]$. However, between the times that the new values for $A[i,j]$ and $A[i+1,j]$ are computed, three whole subrows of elements in that process's partition are accessed by that process. If the three subrows don't fit together in the cache, then $A[i,j]$ will no longer be in the cache when it is accessed again to compute $A[i+1,j]$.

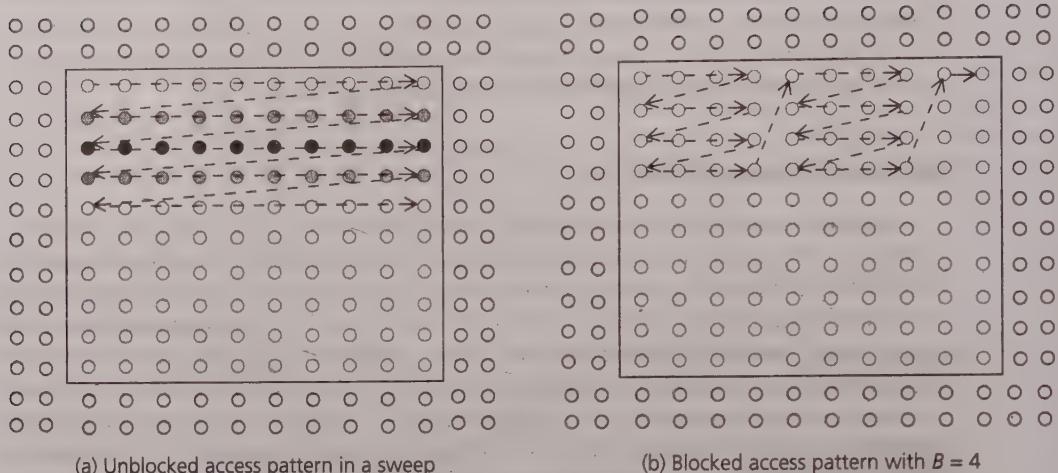


FIGURE 3.7 Blocking to exploit temporal locality in the equation solver kernel. The figure shows the access patterns for a process traversing its partition during a sweep, with the arrow-headed lines showing the order in which grid points are updated. Updating the subrow of bold elements requires accessing that subrow as well as the two subrows of shaded elements. Updating the first element of the next (shaded) subrow requires accessing the first element of the bold subrow again, but these three whole subrows (the black and the shaded) have been accessed since the last time that first bold element was accessed. By changing the update order, the blocked access pattern improves reuse by a constant factor.

If the backing store for the data is nonlocal, artifactual communication will result. The problem can be addressed by changing the order in which elements are computed, as shown in Figure 3.7(b). Essentially, a process proceeds left to right not for the length of a whole subrow of its partition but only for a certain length B before it moves on to the corresponding portion of the next subrow. It performs its sweep in subsweeps over B -by- B blocks of its partition. The block size B is chosen so that at least three B -length rows of a partition fit in the cache. Of course, this changes the order of the updates and hence perhaps the convergence properties unless red-black ordering is used. ■

This technique, called *blocking*, structures computation so that it accesses a subset of data that fits in a level of the hierarchy, uses that data as much as possible, and then moves on to the next such set of data. In the equation solver kernel, the reduction in miss rate due to blocking is only a small constant factor (about a factor of two). The reduction is only seen when three subrows of a process's partition of a grid do not fit in the cache, so blocking is not always useful. However, blocking is used very successfully in linear algebra computations like matrix multiplication or matrix factorization, where $O(n^{k+1})$ computation is performed on a data set of size $O(n^k)$, so each data item is accessed $O(n)$ times. Using blocking effectively with B -by- B blocks in these cases can reduce the miss rate by a factor of B , which is particularly impor-

tant since much of the data accessed is nonlocal. Not surprisingly, many of the same types of restructuring are used to improve temporal locality in sequential programs as well; for example, blocking is critical for high performance in sequential matrix computations, as in Exercise 3.10. Techniques for temporal locality can be used at any level of the hierarchy where data is replicated—including main memory—and for both explicit or implicit replication.

The temporal locality and data referencing patterns of applications have important implications for parallel architecture. For example, they help determine which programming model and communication abstraction a system should support, an issue we consider in Section 3.6. The sizes and scaling of working sets have obvious implications for the amounts of replication capacity needed at different levels of the memory hierarchy and for the number of levels that make sense in this hierarchy. In a cache-coherent shared address space, the sizes and compositions of working sets (i.e., whether they hold local or remote data or both) help determine whether it is useful to replicate communicated data in local main memory as well or simply to rely on caches, and if so, how this should be done. In message passing, they help us determine what data to replicate and how to manage the replication. Of course, it is not only the working sets of individual applications that matter for sizing the memory hierarchy but those of the entire workloads and the operating system that run on the machine. For hardware caches, the size of cache needed to hold a working set depends on its organization (associativity and block size) as well.

Exploiting Spatial Locality

A level of the extended memory hierarchy exchanges data with the next level at a certain *granularity of data transfer*. This granularity may be fixed (e.g., a cache block or a page of main memory) or flexible (e.g., explicit user-controlled messages or user-defined objects). It usually becomes larger as we go farther away from the processor since the latency and fixed start-up cost of each transfer become greater and should be amortized over a larger amount of data. To exploit a large granularity of communication or data transfer, we should organize our code and data structures to exploit spatial locality.³ Not doing so can lead to artifactual communication if the transfer is to or from a remote node and is implicit (at some fixed granularity) as in a shared address space. Even if the transfer is explicit and of user-determined size, poor spatial locality can lead to more costly communication, since either smaller messages may have to be sent or the data may have to be made contiguous before it is sent. As in uniprocessors, poor spatial locality can also lead to a high frequency of TLB misses.

3. The principle of spatial locality states that if a given memory location is referenced now, then it is likely that memory locations close to it in the address space will be referenced in the near future. It should be clear that what is called spatial locality at the granularity of individual words can also be viewed as temporal locality at the granularity of cache blocks or larger units; that is, if a cache block is accessed now, then it (and the data on it) is likely to be accessed in the near future.

In a shared address space, artifactual communication can also result from mismatches of spatial locality with two other important granularities. One is the *granularity of allocation*, which is the granularity at which data is allocated in the local memory or replication store (e.g., a page in main memory). This determines the granularity at which the data can be distributed among physical main memories; that is, when data is allocated through the operating system at page granularity, we cannot allocate a part of a page in one node's memory and another part of the page in another node's memory. Suppose two words that are mostly accessed by two different processors fall on the same page. The page might be allocated in only one processor's local memory, in which case capacity or conflict cache misses to its word by the other processor will generate communication. The other important granularity is the *granularity of coherence*, in which case unrelated words that happen to fall on the same unit of coherence in a coherent shared address space can also cause artifactual communication. This problem, called *false sharing*, is discussed further in Chapter 5.

The techniques used for all these aspects of spatial locality in a shared address space are similar to those used on a uniprocessor, with one new aspect: we should try to keep the data accessed by a given processor close together (contiguous) in the address space and data accessed by different processors apart. Spatial locality issues in a shared address space are best examined in the context of particular architectural styles, and we do so in Chapters 5 and 8. Here, for illustration, we look at one example: how data may be restructured to interact better with the granularity of allocation in the equation solver kernel.

EXAMPLE 3.2 Consider a shared address space system in which main memory is physically distributed among the nodes and in which the granularity of allocation in main memory is a page (4 KB, say). Assume that a given page is allocated in only one node's main memory. Now consider the grid used in the equation solver kernel. What is the problem created by the granularity of allocation, and how might it be addressed?

Answer The natural data structure with which to represent a two-dimensional grid in a shared address space, as in a sequential program, is a two-dimensional array. In a typical programming language, a two-dimensional array data structure is allocated in either a row-major or column-major order.⁴ The gray arrows in Figure 3.8(a) show the contiguity of virtual addresses in a row-major allocation, which is the one we assume. While a two-dimensional shared array has the programming advantage of being the same data structure used in a sequential program, it interacts poorly with the granularity of allocation on a machine with physically distributed memory.

Consider the partition of processor P_5 in Figure 3.8(a). An important working set for the processor is its entire partition, which it streams through in every sweep and reuses across sweeps. If its partition does not fit in the processor's cache hierarchy,

4. Consider the array as being a two-dimensional matrix, with the first dimension specifying the row number in the matrix and the second dimension the column number. Row-major allocation means that all elements in the first row are contiguous in the virtual address space, followed by all the elements in the second row, and so on. The C programming language, which we assume here, is a row-major language. Fortran, for example, is column-major.

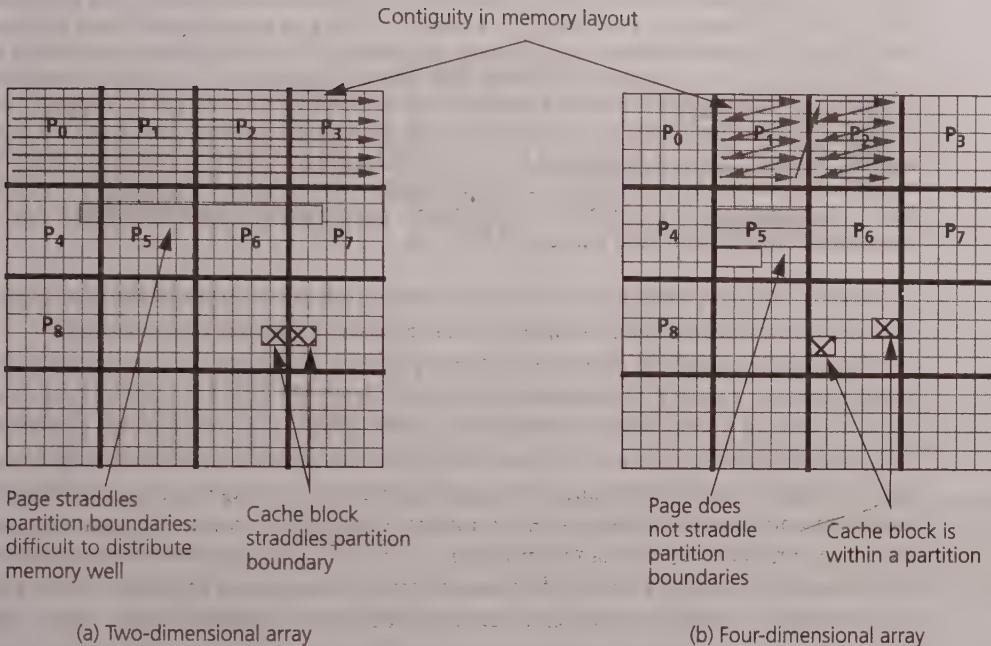


FIGURE 3.8 Two-dimensional and four-dimensional arrays used to represent a two-dimensional grid in a shared address space

we would like it to be allocated in local memory so that the misses can be satisfied locally. The problem is that consecutive subrows of this partition are not contiguous with one another in the address space but are separated by the length of an entire row of the grid (which contains subrows of other partitions). This makes it impossible to distribute data appropriately across main memories if a subrow of a partition is either smaller than a page or not a multiple of the page size or not well aligned to page boundaries. Subrows from two (or more) adjacent partitions will fall on the same page, which at best will be allocated in the local memory of one of those processors. If a processor's partition does not fit in its cache or if it incurs conflict misses, it may have to communicate every time it accesses a grid element in its own partition that happens to be allocated nonlocally.

The solution in this case is to use a higher-dimensional array to represent the two-dimensional grid. The most common example is a four-dimensional array, in which case the processes are arranged conceptually in a two-dimensional grid of partitions, as seen in Figure 3.8(b). The first two indices specify the partition or process being referred to, and the last two represent the subrow and subcolumn numbers within that partition. For example, if the size of the entire grid is $1,024 \times 1,024$ elements, and there are 16 processes, then each partition will be a subgrid of size

$$\frac{1,024}{\sqrt{16}} \times \frac{1,024}{\sqrt{16}}$$

or 256×256 elements. In the four-dimensional array representation of the grid, the array will be of size $4 \times 4 \times 256 \times 256$ elements. The key property of these higher-dimensional representations is that each process's 256×256 element partition is now contiguous in the address space (see the contiguity in the virtual memory layout in Figure 3.8[b]). The data distribution problem can now occur only at the endpoints of entire partitions, rather than of each subrow, and does not occur at all if the data structure is aligned to a page boundary. However, it is substantially more complicated to write code using the higher-dimensional arrays, particularly for array indexing of neighboring processes' partitions in the case of the near-neighbor computation (see Exercise 3.16). ■

More complex applications and data structures illustrate more significant trade-offs in data structure design for spatial locality, as we discuss in later chapters.

The spatial locality in processes' access patterns and how they scale with the problem size or number of processors affects the desirable sizes for various granularities in a shared address space architecture—specifically, the granularities of allocation, transfer, and coherence. It also affects the importance of providing support tailored toward small versus large messages in message-passing systems. Amortizing hardware and transfer costs pushes us toward large granularities, but granularities too large can cause performance problems, many of them specific to multiprocessors. Finally, the spatial locality of access affects the occurrence of conflict misses in a cache. Since conflict misses can generate artifactual communication when the backing store is nonlocal, multiprocessors push us toward higher associativity for caches. There are many cost, performance, and programmability trade-offs concerning support for data locality in multiprocessors, and our choices are best guided by the behavior of applications.

Finally, interesting trade-offs often emerge among algorithmic partitioning goals, implementation issues, and architectural interactions that generate artifactual communication, suggesting that careful examination of trade-offs is needed to obtain the best performance on a given architecture. Let us illustrate using the equation solver kernel in Example 3.3.

EXAMPLE 3.3 Given the performance issues discussed so far, should we choose to partition the equation solver kernel into squarelike subgrids (blocks) or into contiguous strips of rows?

Answer If we only consider inherent communication, we already know that a block domain decomposition is better than partitioning into contiguous strips of rows (see Figure 3.5). However, a strip decomposition has the advantage that it keeps a partition wholly contiguous in the address space even with the simpler, two-dimensional array representation. Hence, it does not suffer problems related to the interactions of spatial locality with machine granularities, such as the granularity of allocation mentioned previously. This particular interaction in the block case can of course be solved by using a higher-dimensional array representation. However, a more difficult interaction to solve is with the granularity of communication. In a subblock assignment, consider a neighbor element from another partition at a column-oriented partition boundary (see Figure 3.9). If the granularity of communication is large, then when a process references this element from its neighbor's partition, it will

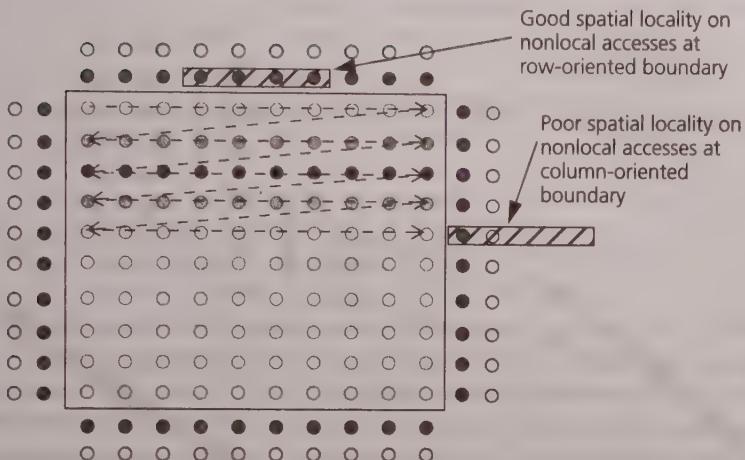


FIGURE 3.9 Spatial locality in accesses to nonlocal data in the equation solver kernel. Only one process's partition is shown, together with the area around its borders. The shaded points are the nonlocal points that the processor owning the partition accesses. The hatched rectangles are cache blocks, showing good spatial locality along the row boundary but poor locality along the column.

fetch not only that element but also a number of other elements that are on the same unit of communication. These other elements are not neighbors of the fetching process's partition regardless of whether a two-dimensional or four-dimensional representation is used, so they are useless and waste communication bandwidth. With a partitioning into strips of rows, there are no column-oriented partition boundaries; a referenced nonlocal element still causes other elements from its row to be fetched, but now these elements are indeed neighbors of the fetching process's partition. They are therefore useful, and in fact the large granularity of communication results in a valuable prefetching effect. Overall, there are many combinations of application and machine parameters for which the performance losses in block partitioning owing to artifactual communication will dominate the performance benefits from reduced inherent communication. We might imagine that strip partitioning should most often perform better when a two-dimensional array is used in the block case, but it may also do so in some cases when a four-dimensional array is used (there is no motivation to use a four-dimensional array with a strip partitioning). Thus, artifactual communication may cause us to go back and revise our partitioning method from block to strip. Figure 3.10(a) illustrates this effect for the Ocean application on the Origin2000 machine. The effect is much larger on systems that have larger granularities of communication and more expensive communication, for example, systems that support the shared address space programming model in software. Figure 3.10(b) uses the equation solver kernel with a larger grid size to illustrate the impact of data placement. Note that a strip decomposition into columns rather than rows will yield the worst of both worlds when data is laid out in memory in row-major order. ■

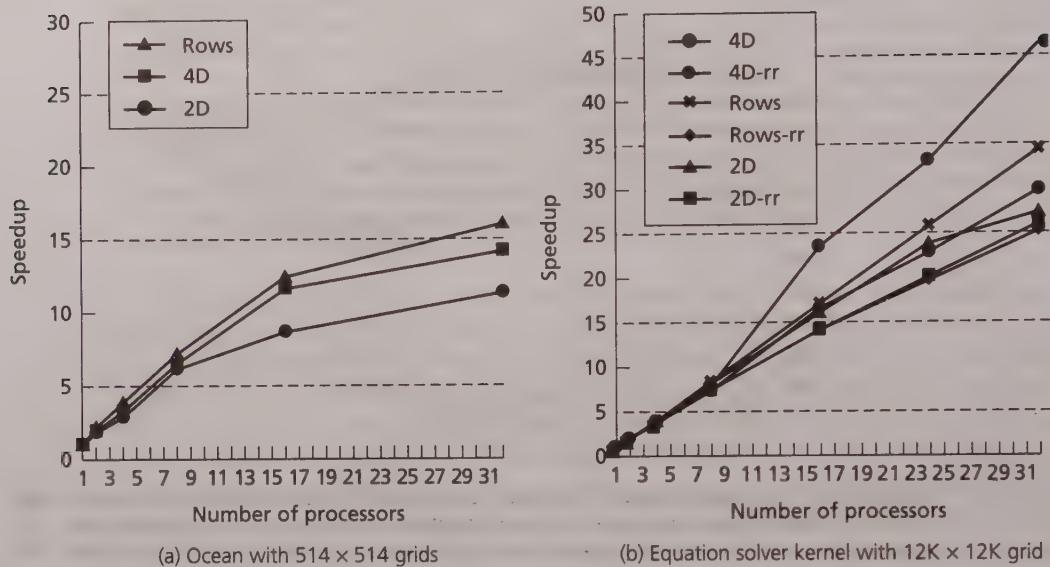


FIGURE 3.10 The impact of data structuring and spatial locality on performance. All measurements are on the SGI Origin2000. “2D” and “4D” imply two- and four-dimensional data structures, respectively, with block (squarelike) assignment. “Rows” uses the two-dimensional array with strip assignment into chunks of rows. In (b), the postfix “rr” means that pages of data are distributed round-robin among physical memories. Without “rr,” it means that pages are placed in the local memory of the processor to which their data is assigned, as far as possible. We see from (a) that the strip assignment outperforms the 2D block assignment because of spatial locality interactions with long cache blocks (128 bytes on the Origin2000) and is even a little better than the 4D array block assignment due to poor spatial locality in the latter in accessing border elements at column-oriented partition boundaries. The graph in (b) shows that, despite the very aggressive communication architecture, in all partitioning schemes proper data distribution in main memory is important to performance, though least successful for the block partitions with 2D arrays. In the best case, we see superlinear speedups once enough processors are used that the size of a processor’s partition of the grid (its important working set) fits into its cache. The differences are much larger on machines with less aggressive communication architectures and smaller replication stores.

3.3.2 Structuring Communication to Reduce Cost

Whether communication is inherent or artificial, how much the communication contributes to execution time is determined by how it is organized or structured into messages. A small communication-to-computation ratio may have a much greater impact on execution time than a large ratio if the structure of the latter interacts much better with the system. This is an important issue in obtaining good performance from a real machine and is the last major performance issue we examine. Let us begin by examining more closely what the structure of communication means.

In Chapter 1, we introduced a model for the cost of communication as seen by a processor, given a frequency of program-initiated communication operations or

messages (explicit messages or messages initiated implicitly by read and write operations). Combining Equations 1.5 and 1.6, that model for the cost C is

$$C = \text{Frequency} \times \left(\text{Overhead} + \text{Delay} + \frac{\text{Length}}{\text{Bandwidth}} + \text{Contention} - \text{Overlap} \right)$$

or

$$C = f \times \left(o + l + \frac{n_c/m}{B} + t_c - \text{Overlap} \right) \quad (3.4)$$

where f is the frequency of communication messages in the program; o is the combined overhead of handling initiation and reception of a message on the sending and receiving processors, assuming no contention with other activities; l is the nonoverhead delay for the first bit of the message to reach the destination processor or memory (assuming no contention), which includes delay through the assists and network interfaces as well as the delay in the network fabric itself; n_c is the total amount of data communicated by the program; m is the number of messages (so n_c/m is the average length of a message); B is the point-to-point bandwidth of communication afforded for the transfer by the communication path, excluding the processor overhead (i.e., the rate at which the rest of the message data arrives at the destination after the first bit, assuming that the entire path through the network from source to destination acts as a single pipeline and there is no contention); t_c is the time induced by contention for resources with other activities; and Overlap is the amount of the communication cost that can be overlapped with computation or other communication (i.e., that is not in the critical path of a processor's execution). The bandwidth B is the inverse of the overall occupancy discussed in Chapter 1. It may be limited by the network links, the network interface, or the communication assist.

This expression for communication cost can be substituted into Equation 3.1 to yield our final expression for speedup. The portion of the cost expression inside the parentheses is our cost model for a single one-way message. If messages are round-trip, we must make the appropriate adjustments. The cost of a message, ignoring overlap, is also called its *latency*. In addition to reducing communication volume (n_c), our goals in structuring communication may include (1) reducing communication overhead ($m \times o$), (2) reducing delay ($m \times l$), (3) reducing contention ($m \times t_c$), and (4) overlapping communication with computation or other communication to hide its latency. Let us discuss programming techniques for addressing each of these issues.

Reducing Overhead

Since the overhead o associated with initiating or processing a message is usually fixed by hardware or system software, the way to reduce the cost due to communication overhead is to make messages fewer in number and hence larger—that is, to

reduce the message frequency.⁵ Explicitly initiated communication allows the programmer greater flexibility in specifying the sizes of messages (recall the send primitive described in Section 2.3.6). On the other hand, implicit communication through read and write operations does not afford the program direct control, and the system must take responsibility for coalescing the reads and writes into larger messages if necessary.

Making messages larger is easy in applications that have regular data access and communication patterns. For example, in the message-passing equation solver partitioned into rows, we send an entire row of data in a single message. But it can be difficult in applications that have irregular and unpredictable communication patterns, such as Barnes-Hut or Raytrace. As we shall see in Section 3.6, it may require changes to the parallel algorithm and extra work to determine which data to coalesce, resulting in a trade-off between the cost of this computation and the savings in overhead. Some computation may be needed to determine what data should be sent and to which process, and the data may have to be gathered and packed into a message at the sender and unpacked and scattered into appropriate memory locations at the receiver.

Reducing Delay

Delay through the assist and network interface can be reduced by optimizing those hardware components. There is not much a programmer can do about this delay. Consider the network transit delay—that is, the delay through the network fabric itself. In the absence of contention and assuming messages are pipelined through the network, the transit delay l of a bit through the network itself can be expressed as $h \times t_h$, where h is the number of hops between adjacent network nodes or switches that the message traverses, and t_h is the delay or latency for a single bit of data to traverse a single network hop, including the link and the router or switch. Like message overhead, t_h is determined by the system, and the program must focus on reducing the f and h components of the $f \times h \times t_h$ delay cost. (In store-and-forward rather than pipelined networks, t_h would be the time for the entire message to traverse a hop, not just a single bit.)

The number of hops h can be reduced by mapping processes to processors so that the topology of interprocess communication in the application exploits locality in the physical topology of the network. How well this can be done in general depends on the application and on the structure and richness of the network; for example, the nearest-neighbor equation solver kernel (and the Ocean application) would map very well onto a mesh-connected multiprocessor but not onto a unidirectional ring. Our other case study applications are more irregular in their communication patterns. (We examine different topologies used in real machines and discuss their trade-offs in Chapter 10.)

5. Some explicit message-passing systems provide different types of messages with different costs and functionalities that a program can choose from.

Research in mapping parallel algorithms to network topologies has been quite extensive since it was thought that as the number of processors p became large, poor mappings would cause the delay due to the $h \times t_h$ term to dominate the cost of messages. How important topology actually is in practice depends on several factors: how large the t_h term is relative to the overhead o of getting a message into and out of the network; the number of processing nodes on the machine, which determines the maximum number of hops h for a given topology; and whether the machine is used to run a single application at a time in “batch” mode or is multiprogrammed among applications. It turns out that network topology is not considered so important on modern machines as it once was because of the characteristics of the machines along all these three axes: overhead dominates hop latency (especially in machines that do not provide hardware support for a shared address space), the number of nodes is usually not extremely large, and the machines are often used as general-purpose, multiprogrammed servers. Topology-oriented program design might not be very useful in multiprogrammed systems since the operating system controls resource allocation dynamically and might transparently change the mapping of processes to processors at run time. For these reasons, the mapping step of parallelization receives considerably less attention than decomposition, assignment, and orchestration. However, this may change again as technology and machine architecture evolve.

Reducing Contention

The communication architectures of multiprocessors consist of many resources, including network links and switches, communication assists, memory systems, and network interfaces. All of these resources have a nonzero *occupancy*, or time for which they are occupied servicing a given transaction. Another way of saying this is that they have finite bandwidth (or rate, which is the reciprocal of occupancy) for servicing transactions. If several messages contend for a resource, some of them will have to wait while others are serviced, thus increasing message latency and reducing the bandwidth available to any single message. Resource occupancy contributes to message cost even in the absence of contention since the time taken to pass through a resource is part of the delay (or overhead, as the case may be), but it can also cause contention. The occupancy of a resource may even be greater than the delay through it.

Contention is a particularly insidious performance problem, for several reasons. First, it is easy to overlook when writing a parallel program, particularly if it is caused by artifactual communication. Second, its effect on performance can be dramatic. If p processors simultaneously contend for a resource of occupancy x , the first to obtain the resource incurs a latency of x because of that resource, whereas the last incurs a latency of at least $p \times x$. In addition to the large stall time, the differences in stall time across processors can also lead to large load imbalances and hence synchronization wait times. Thus, the contention caused by the occupancy of a resource can be much more dangerous than just the delay it contributes in uncontended cases. Third, contention for one resource can hold up other resources, thus stalling

transactions that don't even need the resource that is the source of the contention. This is similar to the way contention for a single-lane exit off a multilane highway causes congestion on the entire stretch of highway. The resulting congestion also affects cars that don't need that exit but want to keep going on the highway since they may be stuck behind cars that do need that exit. The backup of cars covers up other unrelated resources (previous exits), making them inaccessible and ultimately clogging up the highway. Bad cases of contention can quickly saturate the entire communication architecture. The final reason that contention is so troublesome is related to the third: the cause of contention may be particularly difficult to identify since the effects might be felt at very different points in the program than the original cause (all the more so if communication is implicit).

Contention in a network can also be viewed as being of two types: at the links or switches within the network, called *network contention*, and at the endpoints or processing nodes, called *endpoint contention*. Network contention, like network delay, can be reduced by mapping processes and scheduling the communication appropriately in the network topology. Endpoint contention occurs when many processors need to communicate with the same processing node at the same time (or when communication transactions interfere with local memory references). When this contention becomes severe, we call that processing node or resource a *hot spot*. Let us examine a simple example of how a hot spot may be formed and how it might be alleviated in software.

Recall the case of processes that want to accumulate their partial sums into a global sum, as in our equation solver kernel. The resulting contention for the global sum can be reduced by using tree-structured communication rather than having all processes send their updates to the owning node directly. Figure 3.11 shows the structure of such many-to-one communication using a binary fan-in tree. The nodes of this tree (often called a software combining tree) are the participating processes. A leaf process sends its update to its parent, which combines its children's updates with its own and sends the combined update to its parent, and so on until the updates reach the root (the process that holds the global sum) in $\log_2 p$ steps. A similar fan-out tree can be used to send data from one to many processes. Tree-based approaches are used to design scalable synchronization primitives, such as barriers that often experience a lot of contention, as well as library routines for other communication patterns.

In general, two programming principles for alleviating contention are to avoid having too many processes communicating with the same process and to stagger messages to the same destination in time so as not to overwhelm the destination or the resources along the way. Contention is often caused when communication is bursty (i.e., the program spends some time not communicating and then suddenly goes through a burst of communication), and temporal staggering reduces burstiness. However, this must be traded off with the advantages of making messages large, which unfortunately tends to increase burstiness.

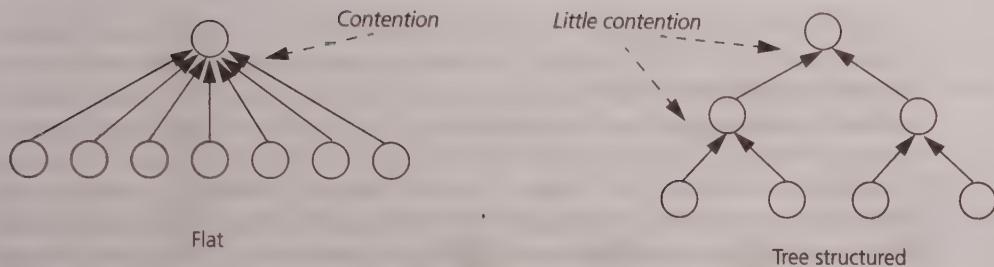


FIGURE 3.11 Two ways of structuring many-to-one communication: flat and tree structured in a binary fan-in tree. Note that the destination processor may receive up to $p - 1$ messages at a time in the flat case, whereas no processor is the destination of more than two messages in the binary tree.

Overlapping Communication with Computation or Other Communication

Despite efforts to reduce overhead and delay, the technology trends discussed in Chapter 1 suggest that the end-to-end communication latency is likely to remain very large in processor cycles. Already, it is in the hundreds of processor cycles, even on machines that provide full hardware support for a shared address space and use high-speed networks, and is at least an order of magnitude higher on message-passing machines due to the higher overhead term α caused by software management. If the processor were to remain idle (stalled) while incurring this latency for every word of data communicated, only programs with an extremely low ratio of communication to computation would yield effective parallel performance. Programs that communicate a lot must therefore find ways to hide the latency of communication from the process's critical path by overlapping it with computation or other communication as much as possible, and systems must provide the necessary support.

Techniques to hide communication latency come in different, often complementary flavors, and we shall examine them in Chapter 11. One approach is simply to make messages larger, thus incurring the latency of the first word but hiding that of subsequent words through pipelined transfer of the large message. Another approach, which we can call *precommunication*, is to initiate the communication well before the data is actually needed, so that by the time the data is needed it is likely to have already arrived. A third technique is to initiate the communication where it naturally belongs in the program but to hide its cost by finding something else for the processor to do—some computation or other communication that occurs later in the same process—while the communication is in progress. A fourth, called *multithreading*, is to switch to a different thread or process when a communication event is encountered. While the specific techniques and mechanisms depend on the communication abstraction and the approach taken, they all fundamentally require the program to have extra concurrency (also called *slackness*) beyond the number of processors used so that independent work (computation or communication) can be found to overlap with the communication latency.

Much of the focus in parallel architecture has in fact been on reducing communication cost as seen by the processor: by reducing communication overhead and delay, by increasing bandwidth and reducing occupancy, and by providing mechanisms to alleviate contention and overlap communication with computation or other communication. Many of the later chapters therefore devote a lot of attention to covering these issues—including the design of node-to-network interfaces, communication assists, and protocols that minimize both software and hardware overhead (Chapters 5 through 9); the design of network topologies, primitive operations, and routing strategies that are well suited to the communication patterns of applications (Chapter 10); and the design of mechanisms to hide communication cost from the processor (Chapter 11). Aggressive architectural methods are usually expensive, so it is important that they can be used effectively by real programs and that their performance benefits justify their costs.

3.4 PERFORMANCE FACTORS FROM THE PROCESSOR'S PERSPECTIVE

To understand the impact of the different performance factors in a parallel program on a parallel architecture, it is useful to look from an individual processor's viewpoint at the different components of time spent executing the program—that is, how much time the processor spends in different activities as it executes instructions, accesses data in the extended memory hierarchy, and coordinates its activities with other processors. These different components of time can be related quite directly to the software performance issues studied in this chapter, helping us relate software techniques to hardware performance. This view also helps us understand what a parallel execution looks like as a workload presented to the architecture and will be useful when we discuss workload-driven architectural evaluation in the next chapter.

In Equation 3.3, we described the time spent executing a sequential program on a uniprocessor as the sum of the time spent actually executing instructions (*busy*) and the time stalled on the memory system (*data access*), where the latter is a “nonideal” factor that reduces performance. Figure 3.12(a) shows a profile of a hypothetical sequential program. In this case, about 80% of the execution time is spent performing instructions, which can be reduced only by improving either the algorithm or the processor. The other 20% is spent stalled on the memory system, which can be improved by improving data locality or the memory system.

In multiprocessors, we can take a similar view, though there are more such nonideal factors. This view cuts across programming models: for example, being stalled waiting for a receive to complete is really very much like being stalled waiting for a remote read to complete or a synchronization event to occur. If the same program is parallelized and run on a four-processor machine, the execution time profile of the four processors might look like that in Figure 3.12(b). The figure assumes a global synchronization point at the end of the program so that all processes terminate at the same time. Note that the parallel execution time (55 s) is greater than one-fourth of the sequential execution time (100 s); that is, we have obtained a speedup of only

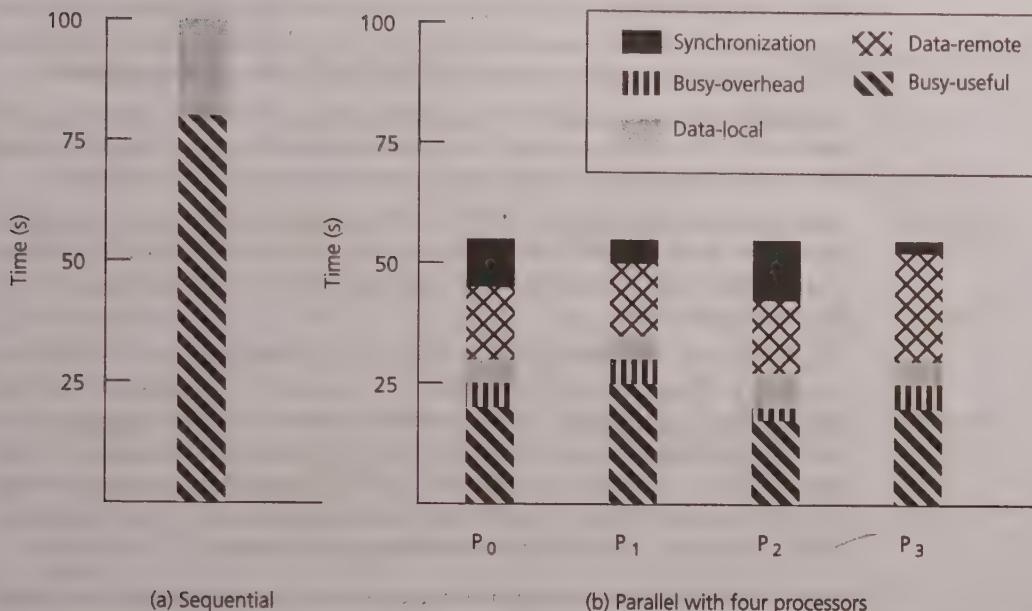


FIGURE 3.12 Components of execution time from the perspective of an individual processor

100/55, or 1.8, instead of the fourfold speedup we may have hoped for. Why this is the case and what specific software or programming factors contribute to it can be determined by examining the components of parallel execution time from the perspective of an individual processor. On our generic parallel architecture with distributed memory, there are five components of parallel execution time:

1. *Busy-useful*: the time that the processor spends executing instructions that would have been executed in the sequential program as well. Assuming a deterministic parallel program⁶ that is derived directly from the sequential algorithm, the sum of the busy-useful times for all processors is equal to the busy-useful time for the sequential execution.
6. A parallel algorithm is deterministic if the result it yields for a given input data set is always the same independent of the number of processes used or the relative timings of events. More generally, we may consider whether all the intermediate calculations in the algorithm are deterministic. A nondeterministic algorithm is one in which the result and the work done by the algorithm to arrive at the result depend on the number of processes and relative event timing. An example is a parallel search through a graph, which stops as soon as any path taken through the graph finds a solution. Nondeterministic algorithms complicate our simple model of where time goes since the parallel program may do less useful work than the sequential program to arrive at the answer. Such situations can lead to *superlinear speedup*—that is, speedup greater than the factor by which the number of processors is increased. However, not all forms of nondeterminism have such beneficial results. Recall that the red-black equation solver described in Chapter 2 is deterministic while the asynchronous one is not.

2. *Busy-overhead*: the time that the processor spends executing instructions that are not needed in the sequential program but only in the parallel program. This corresponds directly to the extra work done in the parallel program.
3. *Data-local*: the time the processor is stalled waiting for a data reference to be satisfied by the memory system on its own processing node; that is, waiting for a reference that does not generate communication with other nodes.
4. *Data-remote*: the time the processor is stalled waiting for data to be communicated to or from another (remote) processing node, whether due to inherent or artifactual communication. This represents the cost of communication as seen by the processor.
5. *Synchronization*: the time spent waiting for another process to signal the occurrence of an event that will allow it to proceed. This includes the load imbalance and serialization in the program as well as the time spent actually executing synchronization operations and accessing synchronization variables. While it is waiting, the processor could be repeatedly polling a variable until that variable changes value—thus executing instructions—or it could be stalled, depending on how synchronization is implemented.⁷

The synchronization, busy-overhead, and data-remote components are not found in a sequential program running on a uniprocessor system and are overheads introduced by parallelism. While inherent communication is mostly included in the data-remote component, some (usually very small) part of it might show up as data-local time as well. For example, data that is assigned to the local memory of a processor P might be updated by another processor Q but asynchronously returned to P's memory (due to replacement from Q, say) before P references it. P may not see the communication cost in this case. Finally, the data-local component is interesting because it is a performance overhead in both the sequential and parallel cases. While the other overhead components tend to increase with the number of processors for a fixed problem or input data set, this component may decrease: a given processor is responsible for only a portion of the overall calculation, so it may only access a fraction of the data that the sequential program does and thus obtain better local cache and memory behavior. In fact, if the data-local overhead reduces enough, it can give rise to superlinear speedups even for deterministic parallel programs (superlinear speedup means speedup greater than the number of processors used). Figure 3.13 summarizes the correspondence between parallelization issues, the steps in which they are largely addressed, and processor-centric components of execution time.

7. Synchronization introduces components of time that overlap with other categories. For example, the time to satisfy the processor's first access to the synchronization variable for the current synchronization event, or the time spent actually communicating the occurrence of the synchronization event, may be included either in synchronization time or in the relevant data access category. Here it is included in the latter. In addition, if a processor executes instructions to poll a synchronization variable while waiting for an event to occur, that time may be defined as busy-overhead or as synchronization. This text includes it in synchronization time since it is essentially load imbalance.

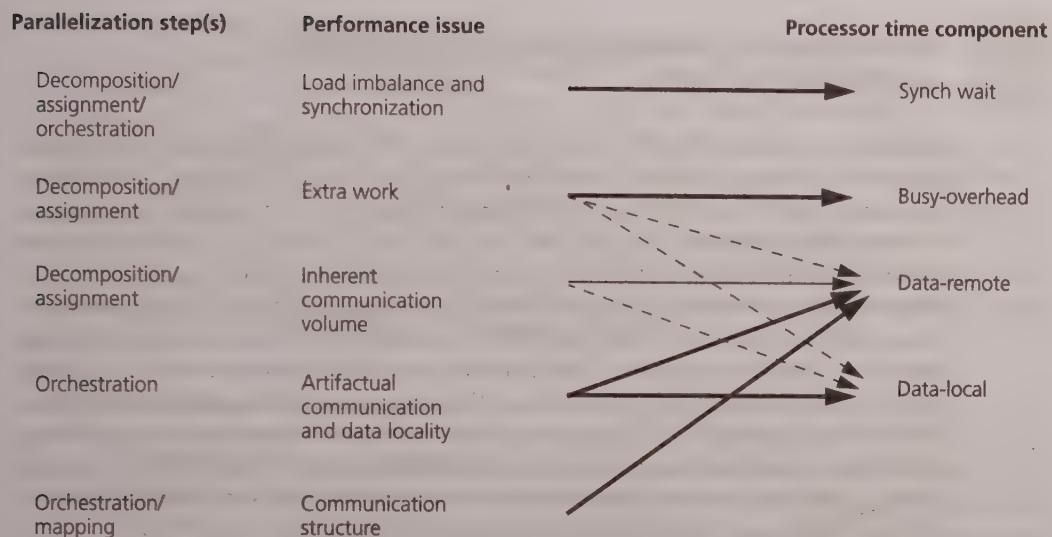


FIGURE 3.13 Mapping between parallelization issues and processor-centric components of execution time. Bold lines depict direct relationships, and dotted lines depict significant side-effect contributions. On the left is the parallelization step in which the issues are mostly addressed.

Using these components, we may further refine our model of speedup for a fixed problem as shown in Equation 3.5, once again assuming a global synchronization at the end of the execution. (Otherwise, we would take the maximum over processes in the denominator instead of taking the time profile of any single process.)

$$\text{Speedup}_{\text{problem}}(p) = \frac{\text{Busy}(1) + \text{Data}_{\text{local}}(1)}{\text{Busy}_{\text{useful}}(p) + \text{Data}_{\text{local}}(p) + \text{Synch}(p) + \text{Data}_{\text{remote}}(p) + \text{Busy}_{\text{overhead}}(p)} \quad (3.5)$$

Our goal in addressing the performance issues has been to keep the terms in the denominator low and thus minimize the parallel execution time (see Figure 3.13). As we have seen, both the programmer and the architecture have their roles to play. The architecture can do little to help if the program is poorly load balanced or if an inordinate amount of extra work exists. However, the architecture can reduce the incentive for creating such ill-behaved parallel programs by making communication and synchronization more efficient. The architecture can also reduce the artifactual communication incurred, provide convenient naming so that flexible assignment mechanisms can be easily employed, and make it possible to hide the cost of communication by overlapping it with useful work.

3.5 THE PARALLEL APPLICATION CASE STUDIES: AN IN-DEPTH LOOK

Having discussed the major performance issues for parallel programs in a general context and having applied them to the simple equation solver kernel, we are ready to examine how to achieve good parallel performance on more realistic applications on real multiprocessors. In particular, we now return to the four application case studies that motivated us to study parallel software in Chapter 2, apply the four steps of the parallelization process to each case study, and at each step address the major performance issues that arise there. In the process, we can understand and respond to the trade-offs among the different performance issues as well as between performance and ease of programming. Examining the components of execution time on a real machine will also help us see the types of workload characteristics that different applications present to a parallel architecture. Understanding the relationship between parallel applications, software techniques, and workload characteristics will be very important as we proceed through the rest of the book.

Parallel applications come in various shapes and sizes with very different characteristics and trade-offs among performance issues. Our four case studies provide an interesting though necessarily restricted cross section through the application space. In examining how to parallelize and, particularly, orchestrate the applications for good performance, we shall focus for concreteness on a specific architectural style: a cache-coherent shared address space multiprocessor with main memory physically distributed among the processing nodes.

The discussion of each application is divided into four subsections. The first describes in more detail the sequential algorithms and the major data structures used. The second describes the partitioning of the application (i.e., the decomposition of the computation and its assignment to processes), addressing the algorithmic performance issues of load balance, communication volume, and the overhead of computing the assignment. The third subsection is devoted to orchestration: it describes the spatial and temporal locality in the program as well as the synchronization used and the amount of work done between synchronization points. The fourth subsection discusses mapping to a network topology. Finally, for illustration we present the components of execution time as obtained for a real execution (using a particular problem size) on a specific machine of the chosen style: a 32-processor Silicon Graphics Origin2000. The busy-useful and busy-overhead components cannot be separated from each other in measurements on this machine, and neither can the data-local and data-remote components, so execution time is divided into three components: busy, data wait, and synchronization. While the level of detail at which we treat the case studies may appear high in some places, these details will be important in explaining the experimental results we shall obtain in later chapters using these applications.

3.5.1 Ocean

Ocean, which simulates currents in an ocean basin, resembles many important applications in computational fluid dynamics. Several of its properties are also representative of a wide range of applications, both scientific and commercial, that stream through large data structures and perform little computation at each data point. At each horizontal cross section through the ocean basin, several different variables are modeled, including the current, temperature, pressure, and friction. Each variable is discretized and represented by a regular, uniform two-dimensional grid of size $(n + 2)$ -by- $(n + 2)$ points ($n + 2$ is used instead of n so that the number of internal, nonborder points that are actually updated in the equation solver is n -by- n). In all, about 25 different grid data structures are used by the application.

The Sequential Algorithm

After the currents at each cross section are initialized, the outermost loop of the application proceeds over a large, user-defined number of time-steps. Every time-step first sets up and then solves partial differential equations on the grids. A time-step consists of 33 different grid computations, each involving one or a small number of grids (variables). Typical grid computations include adding together scalar multiples of a few grids and storing the result in another grid (e.g., $A = \alpha_1 B + \alpha_2 C - \alpha_3 D$), performing a single nearest-neighbor averaging sweep over a grid and storing the result in another grid, and solving a system of partial differential equations on a grid using an iterative method.

The iterative equation solver used is the multigrid method. This is a complex but efficient variant of the simple equation solver kernel we have discussed so far. In the simple solver, each sweep traverses the entire n -by- n grid (ignoring the border columns and rows). A multigrid solver, on the other hand, performs sweeps over a hierarchy of grids. The original n -by- n grid is the finest-resolution grid in the hierarchy; the grid at each coarser level removes every alternate grid point in each dimension, resulting in grids of size $n/2$ -by- $n/2$, $n/4$ -by- $n/4$, and so on. The first sweep of the solver traverses the finest grid, and successive sweeps are performed on coarser or finer grids depending on the error computed in the previous sweep, terminating when the system converges within a user-defined tolerance on the finest grid. To keep the computation deterministic and make it more efficient, a red-black ordering is used (see Section 2.3.2).

Decomposition and Assignment

Ocean affords concurrency at two levels within a time-step: across grid computations (function parallelism) and within a single grid computation (data parallelism). Little concurrency is available across successive time-steps. Concurrency across grid computations can be discovered by writing down which grids each computation reads and writes and analyzing the data dependences among them at this level. The

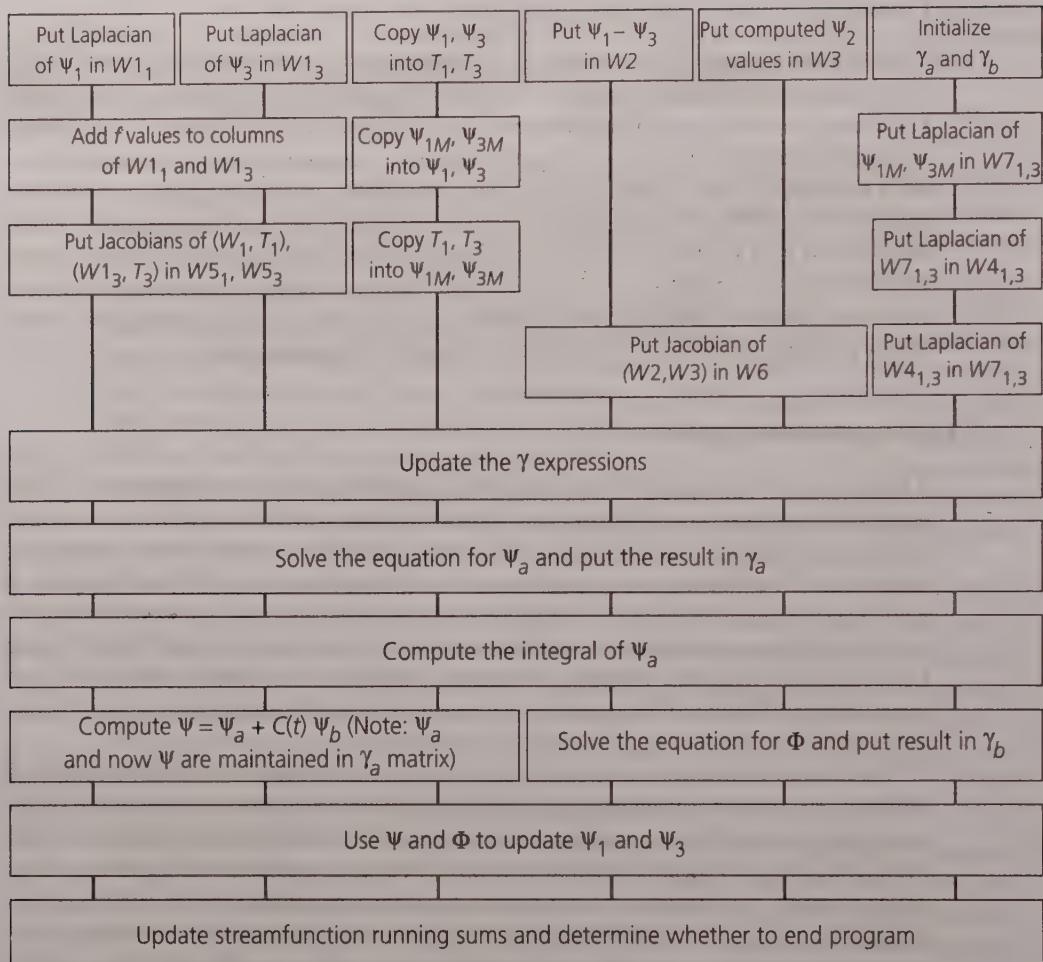


FIGURE 3.14 Ocean: The phases in a time-step and the dependences among grid computations. Each box is a grid computation (or pair of similar computations). Computations connected by vertical lines are dependent while others, such as those in the same row, are independent. The parallel program treats each horizontal row as a phase and synchronizes between phases.

resulting dependence structure and concurrency are depicted in Figure 3.14. Clearly, there is not enough concurrency across grid computations (i.e., not enough vertical sections) to occupy more than a few processors. We must therefore exploit the data parallelism within a grid computation as well, and we need to decide what combination of function and data parallelism is best.

In this case study, we choose to have all processes collaborate on each grid computation rather than to divide the processes among the available concurrent grid computations and use both levels of parallelism. Combined data and function paral-

lelism would increase the size of each process's partition of a grid and hence reduce the communication-to-computation ratio. However, the work associated with different grid computations is quite varied and also depends on problem size in different ways, which complicates load balancing. Second, since several different computations in a time-step access the same grid, for communication and data locality reasons we would not like the same grid to be partitioned in different ways among processes in different computations. Third, all the grid computations are fully data parallel, and all grid points in a given computation do roughly the same amount of work, so we can statically assign grid points to processes. Nonetheless, knowing which grid computations are independent is useful because it allows processes to avoid synchronizing between them (see Figure 3.14).

The issues regarding inherent communication are very similar to those in the simple equation solver, so we use a block-structured (squarelike) domain decomposition of each grid. There is one complication—a trade-off between data locality and load balance related to the points at the border of the grid in some grid computations. The internal n -by- n points do similar work and are divided equally among all processes. Complete load balancing demands that border points, which often do less work, also be divided equally among processors. However, communication and data locality suggest that border points should be assigned to the processes that own the nearest internal points, which assign no border elements to several of the processes. We follow the latter strategy, incurring a slight load imbalance.

Finally, let us examine the multigrid equation solver. The grids at all levels of the multigrid hierarchy are partitioned in the same block-structured domain decomposition. However, the number of grid points per process decreases as we go to coarser levels of the hierarchy, so at the highest levels, some processes may become idle. Fortunately, relatively little (if any) time is spent at these load-imbalanced levels. The ratio of communication to computation also increases at higher levels since there are fewer points per process. This illustrates the importance of measuring speedups relative to the best sequential algorithm (here multigrid): a classical, nonhierarchical parallel iterative solver on the original (finest) grid would likely yield better *self-relative* speedups (relative to a single processor performing the same computation) than the parallel multigrid solver, but the multigrid solver is far more efficient sequentially and overall. In general, less efficient sequential algorithms often yield better self-relative speedups, but these are not useful measures for an end user.

Orchestration

Here we are mostly concerned with artifactual communication, data locality, and synchronization. Let us consider issues related to spatial locality first, then temporal locality, and finally synchronization.

Spatial Locality Within a grid computation, the issues related to spatial locality are similar to those of the simple equation solver kernel in Section 3.3.1. A four-dimensional array data structure is therefore used to represent each grid. This

results in very good spatial locality, particularly on local data. Accesses to nonlocal data (the elements at the boundaries of neighboring partitions) yield good spatial locality along row-oriented partition boundaries and poor locality (hence fragmentation or waste in communication) along column-oriented boundaries. One major difference between the simple solver and the complete Ocean application is that Ocean involves 33 different grid computations in every time-step, each involving one or more out of 25 different grids, so we experience many cache conflict misses across grids. These conflict misses are reduced by ensuring that the allocated dimensions of the arrays are not powers of two (even if the program uses power-of-two grids), but it is difficult to lay out different grids relative to one another to minimize conflict misses. A second difference has to do with the multigrid solver. Since a process's partition has fewer grid points at higher levels of the grid hierarchy, spatial locality is reduced and it is more difficult to distribute data appropriately among main memories at page granularity, despite the use of four-dimensional arrays.

Working Sets and Temporal Locality Ocean has a complicated working set hierarchy, with six working sets. The first two are due to the use of near-neighbor computations within a grid and are similar to those for the simple equation solver kernel. The first working set is captured when the cache is large enough to hold a few grid points so that a point that is accessed as the right neighbor for the previous point is reused to compute itself and to serve as the left neighbor for the next point. The second working set comprises three subrows of a process's partition. When the process returns from one subrow to the beginning of the next in a near-neighbor computation, it can reuse the elements of the previous subrow.

The rest of the working sets are not well defined as single working sets and do not produce sharp knees in the working set curve. The third working set constitutes a process's entire partition of a grid used in the multigrid solver. This could be the partition at any level of the multigrid hierarchy at which the process tends to iterate, so it is not really a single working set. The fourth working set consists of the sum of a process's subgrids at several successive levels of the grid hierarchy within which it tends to iterate (in the extreme, this becomes all levels of the grid hierarchy). The fifth working set allows reuse on a grid across grid computations or even phases; thus, it is large enough to hold a process's partition of several grids. The last working set holds all the data that a process is assigned in every grid so that all the data can be reused across time-steps.

The working sets that are most important to performance are the first three or four, depending on how the multigrid solver behaves. The largest among these grows linearly with the size of the data set per process. This growth rate is common in applications that repeatedly stream through their data sets, so with large data sets, some important working sets do not fit in the local caches. Fortunately, large data sets in these streaming applications make it easy to distribute data in memory at page granularity, so the working sets for a process consist mostly of local rather than communicated data. The little reuse that nonlocal data affords is captured by the first two working sets.

Synchronization Ocean uses two types of synchronization. First, global barriers are used to synchronize all processes between computational phases (see Figure 3.14) as well as between sweeps of the multigrid equation solver. Between several of the phases, we could replace the barriers with finer-grained point-to-point synchronization at the level of grid points to obtain some overlap across phases; however, in this case the overlap is likely to be too small to justify the programming complexity and the overhead of many more synchronization operations. The second form of synchronization is the use of locks to provide mutual exclusion for global reductions, for example, to determine convergence in the solver. The work between synchronization points is large, typically proportional to the size of a process's partition of a grid.

Mapping

Given the near-neighbor communication pattern, we would like to map processes to processors such that processes whose partitions are adjacent to each other in the grid run on processors that are near each other in the network topology. Our subgrid partitioning of two-dimensional grids clearly maps very well to a two-dimensional mesh network. However, as in all our programs, mapping of processes to processors is not enforced by the program but is left to the system.

Summary

Ocean is a good representative of many applications that stream through regular arrays. The computation-to-communication ratio is proportional to n/\sqrt{p} for a problem with n -by- n grids and p processors, load balance is good except when n is not large relative to p , and the parallel efficiency for a given number of processors increases with the grid size. Since a processor streams through its portion of the grid in each grid computation, since only a few instructions are executed per access to grid data during each sweep, and since significant potential exists for conflict misses across grids, data distribution in main memory can be very important on machines with physically distributed memory.

Figure 3.15 shows the breakdown of execution time into busy, waiting at synchronization points, and waiting for data accesses to complete for a particular execution of Ocean with $1,030 \times 1,030$ grids using 2D and 4D arrays on a 32-processor SGI Origin2000 machine. This machine has very large per-processor second-level caches (4 MB), so with four-dimensional array representations each processor's partition tends to fit comfortably in its cache. The problem size is large enough relative to the number of processors that the inherent communication-to-computation ratio is quite low. The major bottleneck is the time spent waiting at barriers. Smaller problems would stress communication more, whereas larger problems and proper data distribution would put more stress on the local memory system. With two-dimensional arrays, the story is clearly different. Conflict misses are frequent, and with data being difficult to distribute appropriately in main memory many of these misses are not satisfied locally, leading to long latencies, contention, and high data wait time.

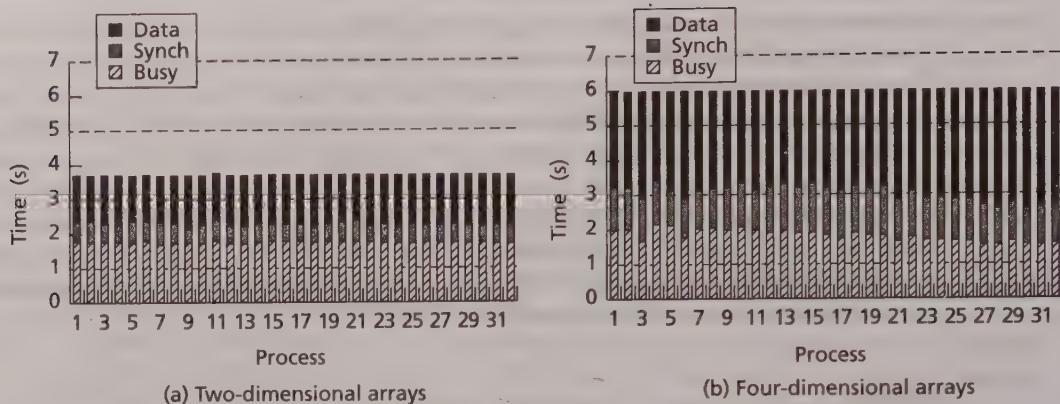


FIGURE 3.15 Execution time breakdown for Ocean on a 32-processor Origin2000. The size of each grid is $1,030 \times 1,030$, and the convergence tolerance is 10^{-3} . The use of four-dimensional arrays to represent the two-dimensional grids in (b) clearly reduces the time spent stalled on the memory system (including communication). This data wait time is very small because a processor's partition of the grids it uses at a given time fit very comfortably in the large 4-MB second-level caches in this machine. With smaller caches or much bigger grids, the time spent stalled waiting for (local) data would have been much larger.⁸

3.5.2 Barnes-Hut

The galaxy simulation has far more irregular and dynamically changing behavior than Ocean. Recall that it solves an n -body problem, in which the major computational challenge is to compute the influences that n bodies in a system exert on one another. The algorithm it uses for computing forces on the stars, the Barnes-Hut method, is an efficient hierarchical method for solving the n -body problem in $O(n \log n)$ time.

The Sequential Algorithm

The galaxy simulation proceeds over hundreds of time-steps, each step computing the net force on every body and thereby updating that body's position and other attributes. Recall the insight that the force calculation in the Barnes-Hut method is based on: if the magnitude of interaction between bodies falls off rapidly with distance (as it does in gravitation), so the effect of a large group of bodies may be approximated by a single equivalent body if that group of bodies is far enough away from the point at which the effect is being evaluated. The hierarchical application of this insight implies that the farther away the bodies, the larger the group that can be approximated by a single body.

8. In this and subsequent execution time breakdowns, there is no artificial final barrier to cause all processes to wait until the last is finished, as in Figure 3.12.

To facilitate a hierarchical approach, the Barnes-Hut algorithm represents the three-dimensional space containing the galaxies as a tree, as follows. The root of the tree represents a space cell containing all bodies in the system. The tree is built by adding bodies into the initially empty root cell and subdividing a cell into its eight equally sized children as soon as it contains more than a fixed number of bodies (here ten). The result is an oct-tree whose internal nodes are space cells and whose leaves contain the individual bodies.⁹ Empty cells resulting from a cell subdivision are ignored. The tree (and the Barnes-Hut algorithm) is therefore adaptive in that it extends to more levels in regions that have high body densities. While we use a three-dimensional problem, Figure 3.16 shows a small two-dimensional example domain and the corresponding quadtree for simplicity. The positions of the bodies change across time-steps, so the tree is rebuilt every time-step. This results in the overall computational structure shown in Figure 3.17, with most of the time being spent in the force calculation phase.

The tree is traversed once per body to compute the net force acting on that body. The force calculation algorithm for a body starts at the root of the tree and conducts the following test recursively for every cell it visits. If the center of mass of the cell is far enough away from the body, the entire subtree under that cell is approximated by a single body at the center of mass of the cell, and the force this center of mass exerts on the body is computed. The rest of that subtree is not traversed. If, however, the center of mass is not far enough away, the cell must be “opened” and each of its sub-cells visited. A cell is determined to be far enough away if the following condition is satisfied:

$$\frac{l}{d} < \theta \quad (3.6)$$

where l is the length of a side of the cell, d is the distance of the body from the center of mass of the cell, and θ is a user-defined accuracy parameter (θ is usually between 0.5 and 1.2). In this way, a body traverses deeper into those parts of the tree representing space that is physically close to it and groups distant bodies at a hierarchy of length scales. Since the expected depth of the tree is $O(\log n)$ and the number of bodies for which the tree is traversed is n , the expected complexity of the algorithm is $O(n \log n)$. Actually it is

$$O\left(\frac{1}{\theta^2} \times n \log n\right)$$

since θ determines the number of tree cells touched at each level in a traversal (smaller θ implies greater accuracy and more tree cells touched). Bodies in denser parts of the space traverse deeper down the tree to compute the forces on themselves, so the work associated with bodies is not uniform.

9. An oct-tree is a tree in which every node has a maximum of eight children. In two dimensions, a quadtree would be used, in which the maximum number of children is four.

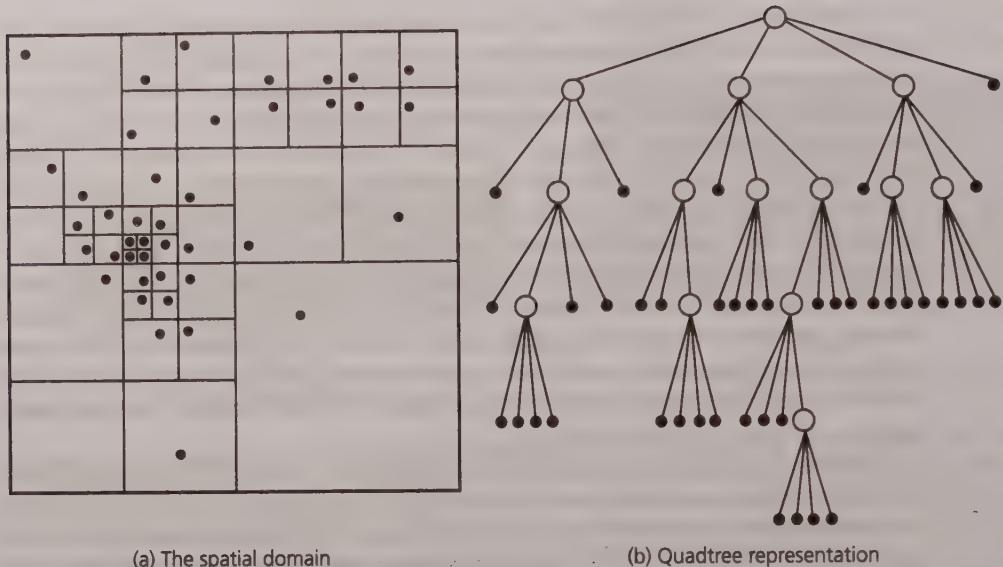


FIGURE 3.16 Barnes-Hut: A two-dimensional particle distribution and the corresponding quadtree

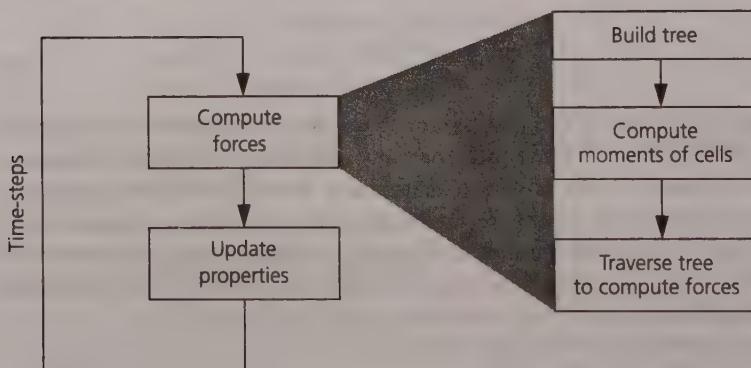


FIGURE 3.17 Flow of computation in the Barnes-Hut application. The force computation phase of an n -body problem expands into three phases (shown on the right) in the Barnes-Hut method.

Conceptually, the main data structure in the application is the Barnes-Hut tree. The tree is implemented in both the sequential and parallel programs with two arrays: an array of bodies and an array of tree cells. Each body and cell is represented as a structure or record. The fields for a body include its three-dimensional position, velocity, acceleration, and mass. A cell structure also has pointers to its children in the tree, and a three-dimensional center of mass. There is also a separate array of

pointers to bodies and one of pointers to cells. Every process owns a contiguous chunk of pointers in these arrays, not necessarily of equal size, which in every time-step are set to point to the bodies and cells that are assigned to it in that time-step. Since the structure and partitioning of the tree changes across time-steps as the galaxy evolves, the actual bodies and cells assigned to a process are not contiguous in the body and cell arrays.

Decomposition and Assignment

Each of the phases within a time-step is executed in parallel, with global barrier synchronization between phases. The natural unit of decomposition (task) in all phases is a body, except in computing the cell centers of mass, where it is a cell.

Unlike Ocean, which has a regular and predictable structure of both computation and communication, the Barnes-Hut application presents many challenges for effective assignment. First, the nonuniformity of the galaxy implies that both the amount of work per body and the communication patterns among bodies are nonuniform, so a good assignment cannot be discovered by inspection. Second, the distribution of bodies changes across time-steps, which means that static assignment is not likely to work well. Third, since the information needs in force calculation fall off with distance equally in all directions, reducing interprocess communication demands that partitions be spatially contiguous and not biased in size toward any one direction. Fourth, the different phases in a time-step have different distributions of work among the bodies/cells, and hence different preferred partitions. For example, the work in the update phase is uniform across all bodies, whereas that in the force calculation phase clearly is not. Another challenge for good performance is that the communication needed among processes is naturally fine grained and irregular.

We focus our partitioning efforts on the force calculation phase since it is by far the most time-consuming. The partitioning is not modified for other phases in accordance with their needs since the cost of doing so, both in repartitioning and in loss of locality, outweighs the potential benefits and since similar partitions are likely to work well for tree building and moment calculation phases (although not for the update phase).

We can use profiling-based semistatic partitioning in this application, taking advantage of the fact that although the spatial distribution of bodies at the end of the simulation may be radically different from that at the beginning, it evolves slowly with time and changes little between two successive time-steps. As we perform the force calculation phase in a time-step, we record the work done by every particle in that time-step (i.e., we count the number of interactions it computes with other bodies or cells). We then use this work count as a measure of the work associated with that particle in the next time-step. Work counting is cheap since it only involves incrementing a local counter when an (expensive) interaction is performed. Now we need to combine this load-balancing method with assignment techniques that also achieve the communication goal: keeping partitions contiguous in space and not biased in size toward any one direction. We briefly discuss two techniques:

the first because it is applicable to many irregular problems and the second because it is better suited to this application and is what our program uses.

The first technique, called orthogonal recursive bisection (ORB), preserves physical locality by partitioning the domain space directly. The space is recursively subdivided into two rectangular subspaces with equal work, using the preceding load-balancing measure, until one subspace per process remains (see Figure 3.18[a]). Initially, all processes are associated with the entire domain space. Every time a space is divided, half the processes associated with it are assigned to each of the subspaces that result. The Cartesian direction in which division takes place is usually alternated with successive divisions, and a parallel median finder is used to determine where to split the current subspace. A separate binary tree of depth $\log p$ is used to keep track of the divisions and to implement ORB. (Details of using ORB for this application can be found in [Salmon 1990].)

The second technique, called costzones, recognizes that the Barnes-Hut algorithm already has a representation of the spatial distribution of bodies encoded in its tree data structure. Thus, we can partition this existing data structure itself and thereby achieve the goal of partitioning space (see Figure 3.18[b]). Every internal cell stores the total cost associated with all the bodies it contains. The total work or cost in the system is divided among processes so that every process has a contiguous, equal range or zone of work (for example, a total work of 1,000 units would be split among 10 processes so that zone 1–100 units is assigned to the first process, zone 101–200 to the second, and so on). Which costzone a body in the tree belongs to can be determined by the total cost of an in-order traversal of the tree up to that body. Processes traverse the tree in parallel, picking up the bodies that belong in their costzone. (Details can be found in [Singh et al. 1995].) The costzones method is much easier to implement than ORB. While the two result in partitions with similar load balance and inherent communication properties, the costzones method yields better overall performance in a shared address space. This is mostly because the time spent in the partitioning phase itself (i.e., computing the partitions) is much smaller, which illustrates the impact of extra work.

Orchestration

Orchestration issues in Barnes-Hut reveal many differences from Ocean, illustrating that even applications in scientific computing can have widely different behavioral characteristics of architectural interest.

Spatial Locality While the shared address space makes it easy for a process to access the parts of the shared tree that it needs in all the computational phases, distributing data to keep a process's assigned bodies and cells in its local main memory is not as easy as in Ocean. First, data would have to be redistributed dynamically as assignments change across time-steps, which can be expensive. Second, the logical granularity of data (a particle/cell) is much smaller than the physical granularity of allocation in memory (a page), and the fact that bodies/cells assigned to the same

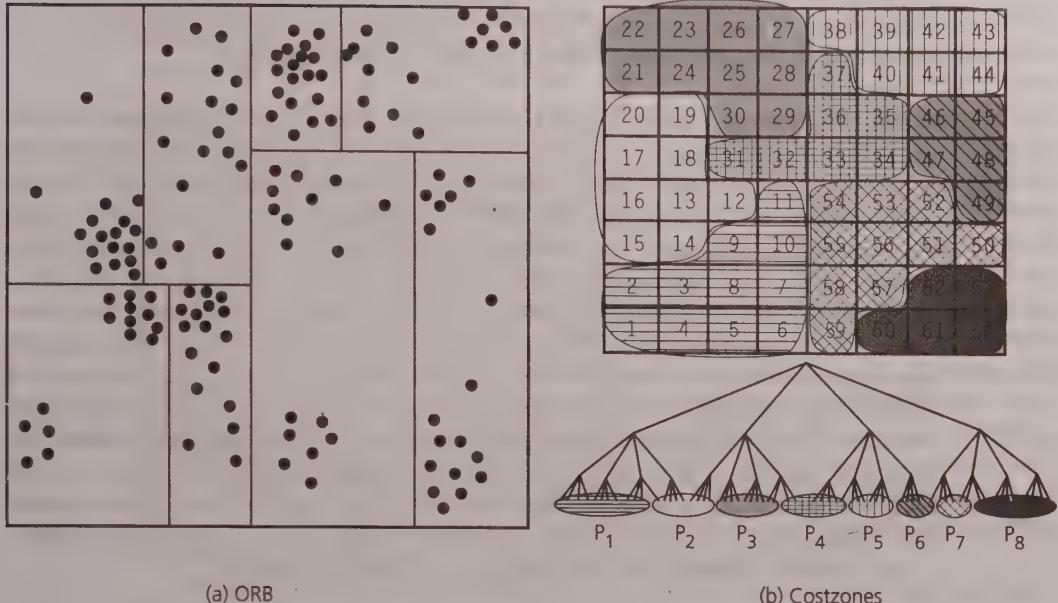


FIGURE 3.18 Partitioning schemes for Barnes-Hut: ORB and costzones. ORB partitions space directly by recursive bisection, while costzones partition the tree. (b) shows both the partitioning of the tree and how the resulting space is partitioned by costzones. ORB leads to more regular (rectangular) partitions than costzones, but their communication and load balance properties are quite similar.

process are contiguous in physical space does not mean that they are spatially contiguous in the body/cell arrays. Fixing these problems requires overhauling the data structures that store bodies and cells: using separate arrays or lists per process that are modified across time-steps as assignments change, and hence different data structures than those used in the sequential program. Fortunately, there is enough temporal locality in the application that data distribution is not so important in a shared address space (again unlike Ocean). In addition, the vast majority of the cache misses are to bodies and cells that are assigned to other processors anyway, so data distribution itself wouldn't help make the misses local. We therefore simply distribute pages of shared data in a round-robin interleaved manner among nodes, without attention to which node gets which pages.

While in Ocean large cache blocks improve local access performance, limited only by partition size, here multiword cache blocks help exploit spatial locality only to the extent that reading a particle's displacement or moment data involves reading several double-precision words of data. Very large transfer granularities might cause more fragmentation than useful prefetching to occur for the same reason that data distribution at page granularity is difficult: unlike in Ocean, locality of bodies/cells in the arrays does not match that in physical space (on which assignment is based),

so fetching data from more than one body/cell in the array upon a miss may be harmful rather than beneficial. Spatial locality depends on the size of a body or cell structure and does not improve much with the number of bodies.

Working Sets and Temporal Locality The first working set in this program contains the data used to compute forces between a single body-body or body-cell pair. The interaction with the next body or cell in the traversal will reuse this data. The second working set is the most important to performance. It consists of the data encountered in the entire tree traversal to compute the force on a single body. Because of the way partitioning is done, the next body on which forces are calculated will be close to this body in space, so the tree traversal to compute the forces on that body will reuse most of this data. As we go from body to body, the composition of this working set changes slowly, but the amount of reuse is tremendous, and the resulting working set is small even though overall a process accesses a very large amount of data in irregular ways. Much of the data in this working set is from other processes' partitions, and most of this data is allocated nonlocally. Thus, it is the temporal locality exploited on shared data (both local and nonlocal) that is critical to the performance of the application, unlike in Ocean where it is data distribution.

By the same reasoning that the complexity of the algorithm is

$$O\left(\frac{1}{\theta^2} \times n \log n\right)$$

the expected size of this working set is proportional to

$$O\left(\frac{1}{\theta^2} \times \log n\right)$$

even though the overall memory requirement of the application is close to linear in n : each particle accesses about this much data from the tree to compute the force on it. The constant of proportionality is small, being the amount of data accessed from each body or cell visited during force computation. Since this working set grows slowly and fits comfortably in modern second-level caches, we do not need to replicate data in main memory. In Ocean, some important working sets grow linearly with the data set size, and we do not always expect them to fit in the cache; however, proper data distribution is easy and keeps most cache misses local, so even in Ocean we do not need replication in main memory.

Synchronization Barriers are used to maintain dependences among bodies and cells across some of the computational phases, such as between building the tree and using it to compute forces. The unpredictable nature of the dependences makes it difficult to replace the barriers by point-to-point synchronization at the granularity of bodies or cells, at least with the programming primitives we assume. The small number of barriers used in a time-step is independent of problem size or number of processors, depending only on the number of phases.

Synchronization is unnecessary within the force computation phase itself. While communication and sharing patterns in the application are irregular, they are phase structured. That is, although a process reads body and cell data from many other processes in the force calculation phase, the fields of a body structure that are written in this phase (the accelerations and velocities) are not the same as those that are read in it (the displacements and masses). The displacements are written only at the end of the update phase, and masses are not modified after initialization. However, in other phases, the program uses both mutual exclusion with locks and point-to-point event synchronization with flags in more interesting ways than Ocean. In the tree-building phase, a process that is ready to add a body to a cell must first obtain mutually exclusive access to the cell since other processes may want to read or modify the cell at the same time. This is implemented with a lock per cell. The phase that calculates cell centers of mass is essentially an upward pass through the tree from the leaves to the root, computing the moments of cells from those of their children. Point-to-point event synchronization is implemented using flags to ensure that a parent does not read the moment of its child until that child has itself been updated by all its children. This is an example of multiple-producer, single-consumer group synchronization. There is no synchronization within the update phase.

The work between synchronization points is large, particularly in the force computation and update phases, where it is

$$O\left(\frac{n \log n}{p}\right)$$

and $O(n/p)$, respectively. The need for locking cells in the tree-building and center-of-mass phases causes the work between synchronization points in those phases to be substantially smaller.

Mapping

The irregular nature makes this application more difficult to map perfectly for network locality in common networks such as meshes. The ORB partitioning scheme maps very naturally to a hypercube topology (discussed in Chapter 10) but not so well to a mesh or other less richly interconnected network. This property does not hold for costzones partitioning, which naturally maps to a one-dimensional array of processors but does not easily guarantee to keep communication local in most network topologies.

Summary

The Barnes-Hut application exhibits irregular, fine-grained, time-varying communication and data access patterns that are becoming increasingly prevalent even in scientific computing as we try to model more complex natural phenomena. Successful partitioning techniques for this application are not obvious by inspection of the code

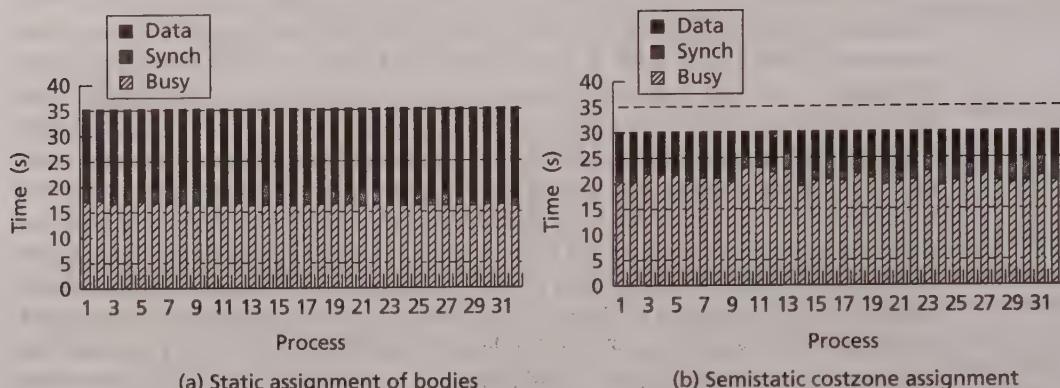


FIGURE 3.19 Execution time breakdown for Barnes-Hut with 512-K bodies on the Origin2000.

The particular static assignment of bodies used is quite randomized, so given the large number of bodies relative to processors, the workload evens out due to the law of large numbers. The bigger problem with the static assignment is that because it is effectively randomized, the particles assigned to a processor are not close together in space so the communication-to-computation ratio is much larger than in the semistatic scheme. This is why data wait time is much smaller in the semistatic scheme. If we had assigned contiguous areas of space to processes statically, data wait time would be small, but load imbalance and hence synchronization wait time would be large. Even with the current static assignment, there is no guarantee that the assignment will remain load balanced as the galaxy evolves over time.

and require the use of insights from the application domain. These insights allow us to avoid using fully dynamic assignment methods, such as task queues and stealing.

Figure 3.19 shows the breakdown of execution time for this application on the 32-processor SGI Origin2000 machine. Load balance is quite good even with a static partitioning of the array of body pointers to processors precisely because there is little relationship between the locations of the bodies in the array and in physical space. However, the data access cost for a static partition is high due to a considerable amount of inherent and artifactual communication caused by the lack of contiguity in physical space. Semistatic costzone partitioning reduces this data access overhead substantially without compromising load balance.

3.5.3 Raytrace

Recall that in ray tracing rays are shot through the pixels in an image plane into a three-dimensional scene and the paths of the rays are traced as they bounce around to compute a color and opacity for the corresponding pixels. The algorithm uses a hierarchical representation of space called a Hierarchical Uniform Grid (HUG), which is similar in structure to the oct-tree used by the Barnes-Hut application. The root of the tree represents the entire space enclosing the scene, and each leaf holds a linked list of the object primitives that fall into that leaf (the maximum number of

primitives per leaf is defined by the user, as are some other aspects of the tree structure). The hierarchical grid or tree makes it efficient to skip empty regions of space when tracing a ray and quickly find the next interesting cell.

The Sequential Algorithm

For a given viewpoint, the sequential algorithm fires one ray into the scene through every pixel in the image plane. These initial rays are called primary rays. At the first object that a ray encounters (found by traversing the hierarchical uniform grid), it is first reflected toward every light source to determine whether it is in shadow from that light source. If it isn't, the contribution of the light source to its color and brightness is computed. The ray is also reflected from and refracted through the object as appropriate. Each reflection and refraction spawns a new ray, which undergoes the same procedure recursively for every object that it encounters. Thus, each primary ray generates a tree of rays. Rays are terminated when they leave the volume enclosing the scene or according to some user-defined criterion (such as the maximum number of levels allowed in a ray tree). Ray tracing, and computer graphics in general, affords several trade-offs between execution time and image quality, and many algorithmic optimizations have been developed to improve performance without significantly compromising image quality.

Decomposition and Assignment

There are two natural approaches to exploiting parallelism in ray tracing. One is to divide the space and, hence, the objects in the scene among processes and have a process compute the ray interactions that occur within its space. The unit of decomposition here is a subspace. When a ray leaves a process's subspace, it will be handled by the next process whose subspace it enters. This is called a *scene-oriented* approach. The alternative *ray-oriented* approach is to divide pixels in the image plane among processes. A process is responsible for the rays that are fired through its assigned pixels, and it follows a ray in its entire path through the scene, computing the interactions of the entire ray tree that the ray generates. The unit of decomposition here is a primary ray. The decomposition unit can be made finer by allowing different processes to process rays generated by the same primary ray (i.e., from the same ray tree) if necessary. The scene-oriented approach preserves more locality in the scene data since a process only touches the scene data in its subspace and the rays that enter that subspace. However, the ray-oriented approach is much easier to program—particularly starting from a sequential program that loops over rays—and to implement with low overhead in a shared address space since rays can be processed independently without synchronization and the scene data is read-only. It is also easily used in a message-passing model with explicit replication of nonlocal scene data. This program therefore uses a ray-oriented approach. The degree of concurrency for an n -by- n plane of pixels is $O(n^2)$ and is usually ample.

Unfortunately, a static block partitioning of the image plane would not be load balanced. Rays from different parts of the image plane might encounter very different numbers of reflections and hence very different amounts of work. The distribution of work is highly unpredictable, so we use a distributed task-queuing system (one queue per processor) with task stealing for load balancing.

To determine how to initially assign rays or pixels to processes, consider communication. Since the scene data is read-only, it causes no inherent communication. If we replicated the entire scene on every node, there would be no communication at all except due to task stealing. However, this approach does not allow us to render a scene larger than what fits in a single node's memory, so the data set size cannot scale with the number of processors used. Other than task stealing, communication is generated because only $1/p$ of the scene is allocated locally on a node while a process accesses the scene widely and unpredictably. To reduce this artifactual communication, we would like processes to reuse scene data as much as possible rather than to access the entire scene randomly. For this, we can exploit spatial coherence in ray tracing: because of the way light is reflected and refracted, rays that pass through adjacent pixels from the same viewpoint are likely to traverse similar parts of the scene and to be reflected in similar ways. This suggests that we should use domain decomposition on the image plane to initially assign pixels to task queues. Since the adjacency or spatial coherence of rays works in all directions in the image plane, a block-oriented domain decomposition works well. This also reduces the communication of image pixels themselves.

Given p processors, the image plane is partitioned into p rectangular blocks of size as close to equal as possible. Every image block or partition is further subdivided into fixed-size square image tiles, which are the units of task granularity and stealing (see Figure 3.20 for a four-process example). These tile tasks are initially inserted into the task queue of the processor to which that block is assigned. A processor ray traces the tiles in its block in scan-line order. When it has finished with its block, it steals tile tasks from other processors that are still busy. The choice of tile size is a compromise between preserving locality through spatial coherence and reducing the number of accesses to other processors' queues, both of which reduce communication, and keeping the task size small enough to ensure good load balance. We could also initially assign tiles to processes in an interleaved manner in both dimensions (called a *scatter decomposition*) to improve load balance in the initial assignment and reduce task stealing at some cost in spatial coherence.

Orchestration

Given the preceding decomposition and assignment, let us examine spatial locality, temporal locality, and synchronization.

Spatial Locality Most of the shared data accesses are to the scene data. However, because of changing viewpoints and the fact that rays bounce about unpredictably, it is impossible to divide the scene into parts that are each accessed only (or even dominantly) by a single process. In addition, the scene data structures are naturally small

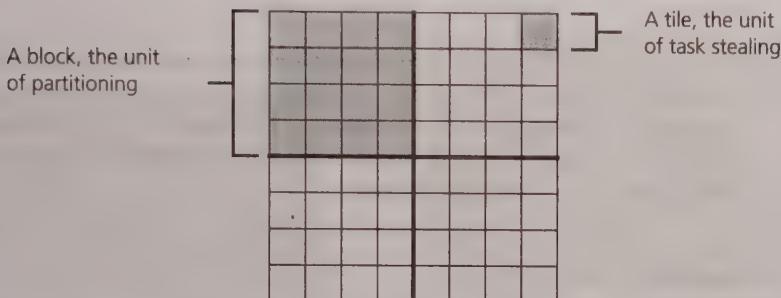


FIGURE 3.20 Image plane partitioning in Raytrace for four processors. Each tile contains several pixels. A contiguous block of tiles is assigned to every process. When a process has finished processing its assigned block, it steals available tiles from other processes.

and linked together with pointers, so it is very difficult to distribute them among memories at the granularity of pages. We therefore resort to using a round-robin layout of the pages that hold scene data to reduce hot spots and contention. Image data is small, and we try to allocate the few pages it falls on in different memories as with scene data. The block assignment described previously preserves spatial locality at cache block granularity in the image plane quite well, though it can lead to some loss of locality at tile boundaries, particularly with task stealing. A strip decomposition in rows of the image plane would be better from the viewpoint of spatial locality but would not exploit spatial coherence in the scene so well. As in Ocean, the best choices may be architecture dependent, and the assignment can be easily parameterized. Spatial locality on scene data is not very high and does not improve with larger scenes.

Temporal Locality Because of the read-only nature of the scene data, if there were unlimited capacity for replication, then only the first reference to nonlocally allocated data would cause communication. With finite replication capacity, on the other hand, data may be replaced and may have to be recommunicated. The domain decomposition and spatial coherence methods described earlier enhance temporal locality on scene data and reduce the sizes of the working sets. However, since the access patterns are so unpredictable due to the bouncing of rays, working sets are relatively large and ill defined. Note that most of the scene data accessed and hence the working sets are likely to be nonlocal. Nonetheless, this shared address space program does not replicate data in main memory: the working sets are not sharp, caches on machines are becoming larger, and replication in main memory has a cost, so it is unclear that the benefits outweigh the overhead.

Synchronization and Granularity Only a single barrier is used after an entire scene is rendered and before it is displayed. Locks are used to protect task queues and for some global variables that track statistics for the program. The work between synchronization points is the work associated with a tile of rays, which is usually quite large.

Mapping

Since Raytrace has very unpredictable access and communication patterns to its scene data, there is little scope for optimizing artificial communication through mapping. The initial assignment partitions the image into a two-dimensional grid of blocks, making it natural to map to a two-dimensional mesh network, but the effect of mapping is not likely to be large.

Summary

This application tends to have large working sets and relatively poor spatial locality but a low communication-to-computation ratio as long as there is ample scene replication capacity. Figure 3.21 shows the breakdown of execution time on the Origin2000 machine for a standard data set consisting of a number of balls arranged in a bunch, illustrating the importance of task stealing in reducing load imbalance and hence wait time at barrier synchronization. The extra communication and synchronization incurred as a result of task stealing is well worthwhile.

3.5.4 Data Mining

A key difference in the data mining application from the previous ones is that the data being accessed and manipulated typically resides on disk rather than in memory. It is very important to reduce the number of disk accesses since their cost is very high and to reduce the contention for a disk controller by different processors. The techniques for reducing disk access cost are essentially the same as those for reducing communication and memory access cost.

Recall the basic insight used in association mining: if an itemset of size k is large (i.e., it occurs in more than a threshold fraction of the transactions), then all subsets of that itemset must also be large. For illustration, consider a database consisting of five items—A, B, C, D, and E—of which one or more may be present in a particular transaction. The items within a transaction are lexicographically sorted. Consider L_2 , the list of large itemsets of size two. This list might be {AB, AC, AD, BC, BD, CD, DE}. The itemsets within L_2 are also lexicographically sorted. Given this L_2 , the list of itemsets that are candidates for membership in L_3 are obtained by performing a join operation on the itemsets in L_2 —that is, taking pairs of itemsets in L_2 that share a common first item (say, AB and AC) and combining them into a lexicographically sorted itemset of size three (here ABC). The resulting candidate list C_3 in this case is {ABC, ABD, ACD, BCD}. Of these itemsets in C_3 , some may actually occur with enough frequency to be placed in L_3 , and so on. In general, the join operation to obtain C_k from L_{k-1} finds pairs of itemsets in L_{k-1} whose first $k - 2$ items are the same and combines them to create a new itemset for C_k . Itemsets of size $k - 1$ that have common $(k - 2)$ -sized prefixes are said to form an equivalence class (e.g., {AB, AC, AD}, {BC, BD}, {CD}, and {DE} in this example for $k = 3$). Only itemsets in the

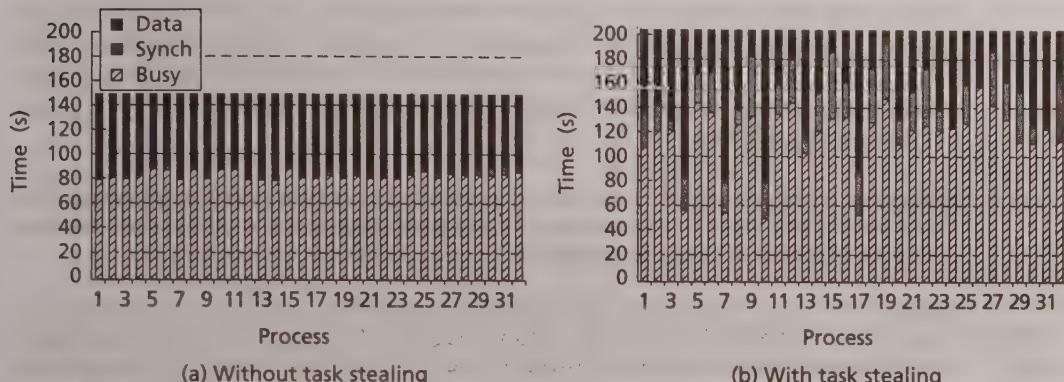


FIGURE 3.21 Execution time breakdown for Raytrace with the balls data set on the Origin2000. Task stealing is clearly very important for balancing the workload (and hence reducing synchronization wait time at barriers) in this highly unpredictable application.

same $k - 2$ equivalence class need to be considered together to form C_k from L_{k-1} , which greatly reduces the number of pairwise itemset comparisons we need to do to determine C_k .

The Sequential Algorithm

A simple sequential method for association mining is to first traverse the data set and record the frequencies of all itemsets of size one, thus determining L_1 . From L_1 , we can construct the candidate list C_2 and then traverse the data set again to find which entries of C_2 occur frequently enough to be placed in L_2 . From L_2 , we can construct C_3 and then traverse the data set to determine L_3 , and so on until we have found L_k . Although this method is simple, it requires reading all transactions in the database from disk k times, which is expensive.

An alternative sequential algorithm seeks to reduce the amount of work done to compute candidate lists C_k from lists of large itemsets L_{k-1} and especially to reduce the number of times data must be read from disk to determine the frequencies of the itemsets in C_k . We have seen that equivalence classes can be used to achieve the first goal. In fact, they can be used to construct a method that achieves both goals together. The idea is to transform the way in which data is stored in the database. Instead of storing transactions in the form $\{T_x, A, B, D, \dots\}$ —where T_x is the transaction identifier and A, B, D are items in the transaction—we can keep in the database records of the form $\{IS_x, T_1, T_2, T_3, \dots\}$, where IS_x is an itemset and T_1, T_2 , and so on are transactions that contain that itemset. That is, a database record is maintained per itemset rather than per transaction. If the large itemsets of size $k - 1$ (i.e., the elements of L_{k-1}) that are in the same $k - 2$ equivalence class are identified, then

computing the candidate list C_k requires only examining all pairs of these itemsets. Since each itemset has its list of transactions attached, the size of each resulting itemset in C_k can be computed at the same time as constructing the C_k itemset itself from a pair of L_{k-1} itemsets, by simply computing the intersection of the transactions in that pair's lists.

EXAMPLE 3.4 Suppose $\{AB, 1, 3, 5, 8, 9\}$ and $\{AC, 2, 3, 4, 8, 10\}$ are large itemsets of size two in the same one-equivalence class (they each start with A). How will the data be accessed in disk and memory?

Answer The list of transactions that contain itemset ABC is $\{3, 8\}$, so the occurrence count of itemset ABC is two. This means that once the database is transposed and the one-equivalence classes identified, the rest of the computation for a single one-equivalence class can be done to completion (i.e., all large itemsets of size k found) before considering any data from other one-equivalence classes. If a one-equivalence class fits in main memory, then after the transposition of the database a given data item needs to be read from disk only once, greatly reducing the number of expensive I/O accesses. A form of blocking for temporal locality has been achieved. ■

Decomposition and Assignment

The two sequential methods also differ in their parallelization, with the latter method having advantages in this respect as well. To parallelize the first method, we could first divide the database among processors. At each step, a processor traverses only its local portion of the database to determine partial occurrence counts for the candidate itemsets, incurring no communication or nonlocal disk accesses in this phase. The partial counts are then merged into global counts to determine which of the candidates are large. Thus, in parallel this method requires not only multiple passes over the database but also interprocessor communication and synchronization at the end of every pass.

In the second method, the equivalence classes that helped the sequential method reduce disk accesses are very useful for parallelization as well. Since the computation on each one-equivalence class is independent of the computation on any other, we can simply divide the one-equivalence classes among processes that can thereafter proceed independently for the rest of the program without communication or synchronization. The itemset lists (in the transformed format) corresponding to an equivalence class can be stored on the local disk of the process to which the equivalence class is assigned, so no need for remote disk access remains after this point. As in the sequential algorithm, each process can complete the work on one of its assigned equivalence classes before proceeding to the next one, so each itemset record from the local disk should also be read only once as part of its equivalence class.

The challenge is ensuring a load-balanced assignment of equivalence classes to processes. A simple metric for load balance is to assign equivalence classes based on

the number of initial entries in them. However, as the computation unfolds to compute itemsets of size k , the amount of work is determined more closely by the number of large itemsets that are generated at each step. Heuristic measures that estimate this or some other more appropriate work metric can be used as well. Otherwise, we may have to resort to dynamic tasking and task stealing, which can compromise much of the simplicity of this method (namely, that once processes are assigned their initial equivalence classes, they no longer have to communicate, synchronize, or perform remote disk access).

The first step in this approach, of course, is to compute the one-equivalence classes and the large itemsets of size two in them as a starting point for the parallel assignment. To compute these itemsets, we are better off using the original transaction-oriented form of the database rather than the transformed version, so we do not transform the database yet (see Exercise 3.18). Every process sweeps over the transactions in its local portion of the database and, for each pair of items in a transaction, increments a local counter for that item pair (the local counts can be maintained as a two-dimensional upper-triangular matrix, with the indices being items). The local counts are then merged, involving interprocess communication, and the large itemsets of size two are determined from the resulting global counts. These itemsets are then partitioned into one-equivalence classes, which are assigned to processes as described earlier.

The next step is to transform the database from the original $\{T_x, A, B, D, \dots\}$ organization by transaction to the $\{IS_x, T_1, T_2, T_3, \dots\}$ organization by itemset, where the IS_x are initially the size two itemsets. This can be done in two steps—a local step and a communication step. In the local step, a process constructs the partial transaction lists for large itemsets of size two from its local portion of the database. In the communication step, a process (at least conceptually) “sends” the lists for those size two itemsets whose one-equivalence classes are not assigned to it to the process to which they are assigned and “receives” from other processes the lists for the equivalence classes that are assigned to it. The incoming partial lists are merged into the local lists, preserving a lexicographically sorted order, after which the process holds the transformed database for its assigned equivalence classes. It can now compute the itemsets of size k step by step for each of its equivalence classes, without any communication, synchronization, or remote disk access (if there is no task stealing). At the end of the calculation, the results for the large itemsets of size k are available from the different processes. The communication step of the transformation phase is usually the most expensive step in the algorithm and is quite like transposing a matrix, except that the sizes of the communications among different pairs of processes are different.

Orchestration

Given this decomposition and assignment, let us examine spatial locality, temporal locality, and synchronization.

Spatial Locality The organization of the computation and the lexicographic sorting of the itemsets and transactions causes most of the traversals through the data to be simple front-to-back sweeps that exhibit very good predictability and spatial locality. This is particularly important in reading from disk, since it is important to amortize the high start-up costs of a disk read over a large amount of useful data.

Temporal Locality As discussed earlier, proceeding over one equivalence class at a time is much like blocking, although how successful it is depends on whether the data for that equivalence class fits in main memory. As the computation for an equivalence class proceeds, the number of large itemsets becomes smaller, so reuse in main memory is more likely to be exploited. Note that here it is more likely that we are exploiting temporal locality in main memory rather than in the cache, although the techniques and goals are similar at any level of the extended memory hierarchy.

Synchronization The major forms of synchronization are the reductions of partial occurrence counts into global counts in the first step of the algorithm (computing the large size two itemsets) and a barrier after this to begin the transformation phase. The reduction is required only for itemsets of size two since thereafter every process continues independently to compute the large itemsets of size k in its assigned equivalence classes. Further synchronization may be needed if dynamic task management is used for load balancing.

Mapping

The communication to transform the database is all-to-all: a process may “send” different itemsets of size two and their partial transaction lists to all other processes and may “receive” or read such lists from them all. It is difficult to map all-to-all communication in a contention-free manner to network topologies (like meshes or rings) that are not very richly interconnected. Endpoint contention is reduced by communication scheduling techniques such as having each processor i at step j exchange data with processor $i \oplus j$ so that no processor or node is overloaded.

Summary

Data mining differs from the other application case studies since disk access is a major bottleneck, and parallelization techniques aim primarily to minimize its cost. The technique we have examined treats the disk as simply another, explicitly managed level of the extended memory hierarchy. Load balance is an outstanding question that can compromise some of the local properties of the parallel program.

3.6 IMPLICATIONS FOR PROGRAMMING MODELS

We have seen throughout this and the previous chapter that while the decomposition and assignment of a parallel program are often (but not always) independent of

the programming model, the orchestration step is highly dependent on it. In Chapter 1, we learned about the fundamental design issues that apply to any layer of the communication architecture, including the programming model. We learned that the two major programming models—a shared address space and explicit message passing between private address spaces—are fundamentally distinguished by functional differences, such as naming, replication, and synchronization. While either programming model can be implemented on any communication abstraction and hardware, the positions taken on these functional issues at a given layer affect (and are influenced by) performance characteristics, such as overhead, latency, and bandwidth. At that stage, we only dealt with those issues in the abstract and could not appreciate the interactions with applications and the implications regarding which programming models are preferable under what circumstances. Now that we have an in-depth understanding of several interesting parallel applications and understand the performance issues in orchestration, we can compare the programming models in light of application and performance characteristics.

We will use the application case studies to illustrate the issues, assuming a generic multiprocessor architecture with physically distributed memory. For a shared address space, we assume that read (loads) and write (stores) to shared data are the only communication mechanisms exported to the user, and we call this a read-write shared address space. Of course, in practice nothing stops a system from providing support for explicit messages as well as these primitives in a shared address space model, but we ignore this possibility for now. The shared address space model can be supported in a wide variety of ways at the communication abstraction and hardware/software interface (recall the discussion of naming models at the end of Chapter 1) with different granularities and different efficiencies for supporting communication, replication, and coherence. These affect the success of the programming model and will be discussed in detail in Chapters 8 and 9. Here we focus on the most common case in which a cache-coherent shared address space is supported efficiently at fine granularity—for example, with direct hardware support for a shared physical address space as well as for communication, replication, and coherence at the fixed granularity of cache blocks. However, contrast this common case with a hardware-supported shared address space without coherent replication, as provided by the BBN Butterfly and CRAY T3D and T3E machines. For the message-passing programming model, we will assume that it too is supported efficiently by the communication abstraction and the hardware/software interface.

As application programmers, we view the programming model as our window to the communication architecture. Differences between programming models and how they are implemented have implications for ease of programming, for the structuring of communication, for performance, and for scalability. In addition to functional aspects (like naming, replication, and synchronization), there are organizational aspects (like the granularity at which communication is performed) and performance aspects (like the endpoint overhead of a communication operation) that differ across programming models and affect programming for performance. Other performance aspects (such as latency and available bandwidth) depend

largely on the network and network interface used and can be assumed to be equivalent. In addition, there are differences in the hardware overhead and complexity required to support the abstractions efficiently and in the ease with which they allow us to reason about or predict performance. Let us examine each of these aspects. The first three aspects we consider—naming, replication, and communication overhead—point to advantages of a read-write shared address space, whereas the others—message size, synchronization, hardware or design cost, and performance predictability—favor explicit message passing.

3.6.1 Naming

As seen, a shared address space makes naming logically shared data much easier for the programmer since any process can directly reference any data and the naming model is similar to that on a uniprocessor. Explicit messages are not necessary, and a process need not name other processes or know which processing node currently owns the data that it needs. In applications with regular, statically predictable communication needs, such as the equation solver kernel and Ocean, it is not difficult to determine which process's address space data resides in and to use explicit messages. However, matching ownership and use can be quite difficult, both algorithmically and for programming, in applications with irregular, unpredictable data needs. An example is Barnes-Hut, in which the parts of the tree that a process needs to traverse to compute forces on its bodies are not statically predictable and the ownership of bodies and tree cells changes with time. Determining which processes to communicate with requires extra work at run time. In Raytrace, rays shot by a process bounce unpredictably around scene data, so if data is distributed among the private address spaces of processes, then it is difficult to determine who owns the next set of data needed. These difficulties can be overcome, but this requires either altering and adding substantial complexity to the algorithm (e.g., adding an extra phase in every time-step to compute who needs what data and then transferring that data in Barnes-Hut [Salmon 1990] or using a scene-oriented rather than a ray-oriented approach in Raytrace), replicating the entire shared data structure on all nodes (not a scalable solution), or emulating an application-specific shared address space in software by hashing from bodies, cells, or scene data to processing nodes. These application-level naming solutions greatly change program appearance and are often among the greatest sources of run-time overhead. They are discussed further in (Singh, Hennessy, and Gupta 1995; Singh, Gupta, and Levoy 1994; Warren and Salmon 1993).

3.6.2 Replication

Several issues distinguish how replication of nonlocal data is managed: (1) Who is responsible for replication, that is, for making local copies of the data? (2) Where in the local memory hierarchy is the replication done? (3) At what granularity is data

allocated in replication store? (4) How are the values of replicated data kept coherent? (5) And how is the replacement of replicated data managed?

With the separate, private virtual address spaces of the message-passing model, the only way to replicate communicated data is to copy the data into a process's private address space explicitly in the application program. The replicated data is explicitly renamed in the new address space, so both the virtual and physical addresses may be different for the two processes; their copies have nothing to do with each other as far as the system is concerned. Data is always replicated in main memory first (when the copies are made), and only data from the local main memory enters the processor cache. The granularity of allocation in the local memory is variable and user dependent. Ensuring that the values of replicated data are kept up-to-date (coherent) must be done by the program through explicit messages. We shall discuss replacement shortly.

Recall that in a shared address space, since nonlocal data is accessed through ordinary processor reads and writes and communication is implicit, opportunities exist for the system to replicate data transparently to the user—without copying or explicit renaming in the program—just as caches do in uniprocessors. This opens up a wide range of possibilities. For example, in a shared physical address space system, nonlocal data transparently enters the processor's cache subsystem upon access, without being replicated in main memory. Replication happens very close to the processor and at the relatively fine granularity of cache blocks, and data is kept coherent by hardware. Other systems may replicate data transparently in main memory first—either at cache block granularity through additional hardware support or at page or object granularity through system software—and may preserve coherence through a variety of methods and granularities that we discuss in Chapter 9. Still other systems may choose not to support transparent replication and/or coherence, leaving them to the user (for example, in the CRAY T3D and T3E systems).

Finally, let us examine the replacement of locally replicated data due to finite capacity. How replacement is managed has implications for the amount of communicated data that needs to be replicated at a level of the local memory hierarchy at a time. For example, hardware caches manage replacement dynamically with every reference and at a fine spatial granularity so that the cache needs to be only as large as the active working set of the workload. When replication is managed by the user program, as in message passing, a similar effect can be achieved by maintaining a cache data structure in the application in local memory and using it to emulate a hardware cache for nonlocal data. However, managing this cache complicates programming, incurs run-time overhead for software lookups and address resolution, and naturally generates fine-grained messages upon cache misses. On the other hand, the software cache can be very large and managed in an application-specific manner.

Typically, message-passing programs manage replacement less dynamically. Explicit local copies of communicated data are allowed to accumulate in local memory and are flushed out explicitly at certain points in the program, typically after a

phase of computation when it can be determined that they are not needed for some time. This can require substantial extra memory for replication in some irregular applications, such as Barnes-Hut and Raytrace. For read-only data like the scene in Raytrace, many message-passing programs simply replicate the entire data set on every processing node, solving the naming problem and almost eliminating communication but also eliminating the ability to run larger problems on larger systems. In the Barnes-Hut application, in one prominent approach (Salmon 1990), a process first replicates locally all the data it needs to compute forces on all its assigned bodies and only then begins to compute forces. While this means that no communication takes place during the force calculation phase, the amount of data replicated in main memory is larger by several factors than the process's assigned partition of data and certainly much larger than the active working set, which is the data needed to compute forces on only one particle. This active working set in a shared address space typically fits in the processor cache, so there is no need for replication in main memory at all. In message passing, the large amount of replication can limit the scalability of the approach. For these reasons as well as for its generality, the approach of emulating a shared address space in software using hashing and of managing replication more dynamically using a fixed-size software cache (which is flushed at phase boundaries for coherence) is becoming increasingly popular for these irregular applications, especially as message passing becomes more efficient for small messages.

3.6.3 Overhead and Granularity of Communication

The overhead of initiating and receiving communication is greatly influenced by the extent to which the necessary tasks can be performed by hardware rather than being delegated to software, particularly to the operating system. Recall that in a shared physical address space the underlying uniprocessor hardware mechanisms suffice for address translation and protection (even when memory is physically distributed) since the shared address space is simply a large flat address space. Simply doing address translation for shared data accesses in software as opposed to hardware reduced Barnes-Hut performance by about 20% in one set of experiments (Scales and Lam 1994). The other major component of overhead is buffer management: incoming and outgoing communications need to be temporarily buffered in the network interface to allow multiple communications to be in progress simultaneously and to stage data through a communication pipeline. Communication at the fixed granularity of words or cache blocks makes it easy to manage buffers very efficiently in hardware. These factors combine to keep the overhead of communicating each cache block quite low on cache-coherent shared address space machines (a few cycles to a few tens of cycles, depending on the implementation and integration of the communication assist). On the other hand, automatic transfer of fixed-size blocks may lead to significant artifactual communication if spatial locality is poor.

In fact, the issue of communication granularity raises an important, if subtle, difference between a cache-coherent shared address space and one that provides transparent naming but not coherent replication (as in the CRAY T3D and T3E). In the former case, communication is performed transparently at a larger granularity than

the word being referenced (e.g., a cache block). The cost of communication is thus amortized without the programmer having to worry about preserving the coherence of the rest of the transferred data; the system takes this responsibility. In the latter case, however, replication and coherence are the programmer's responsibility; so on a miss, the system fetches only the referenced word (otherwise, the burden on the programmer may be too large).

In message-passing systems, local references incur no more overhead than on a uniprocessor. Communication messages, however, are very flexible and therefore incur a lot of overhead. The variety of message types requires software overhead to decode the type of message and execute the corresponding handler routine at the sending or receiving end. The flexible message length, together with the use of asynchronous and nonblocking messages, complicates buffer management so that system software must often be invoked to temporarily store messages. Finally, sending explicit messages between arbitrary address spaces requires that the operating system on a node (or hardware support) intervene to provide protection. The software overhead for buffer management and protection can be substantial, particularly when the operating system must be invoked. A lot of recent design effort has focused on streamlined network interfaces and message-passing mechanisms that significantly reduce per-message overhead. These approaches can restrict flexibility and are discussed in Chapter 7. Nevertheless, the overhead per message is likely to remain several times as large as that of hardware-supported read-write shared address space interfaces, limiting the effectiveness of approaches that naturally generate fine-grained communication in irregular applications.

These three issues—naming, replication, and communication overhead—have pointed to the advantages of an efficiently supported shared address space for parallel programming. Let us now examine issues that favor message passing.

3.6.4

Block Data Transfer

Implicit communication through reads and writes in a hardware-supported cache-coherent shared address space typically causes a message to be generated for each reference, or at least for each cache block, that requires communication. The communication is usually initiated by the process that needs the data, via a cache miss, and we call it receiver-initiated communication. While the hardware support provides efficient fine-grained communication, communicating one cache block at a time is not the most efficient way to communicate a large chunk of data from one processor to another. We would rather amortize the overhead and latency by communicating the data in a single message or a group of large messages, a method called *block data transfer*.

Explicit communication, as in message passing, allows greater flexibility in choosing the sizes of messages and in choosing whether communication is receiver initiated or sender initiated, thus naturally enabling block transfer. Explicit communication can even be added to a hardware-coherent shared address space naming model, giving the programmer a choice of communication methods, and it is also

possible for the system to make communication coarser grained transparently underneath a read-write programming model in some cases of predictable communication. However, the natural communication structure promoted by a shared address space is fine grained and usually receiver initiated. The advantages of block transfer in a hardware-supported shared address space are somewhat complicated by the availability of alternative latency tolerance techniques, but it clearly does have advantages.

3.6.5 Synchronization

The fact that synchronization can be contained in the (explicit) communication itself in message passing, while it is usually explicit and separate from the implicit data communication in a shared address space, tends to eliminate much of the programming concern over synchronization. Mutual exclusion is provided automatically, and few flags are used. Thus subtle race conditions and timing bugs may be less common in message passing. In addition, the difficulties of fine-grained sharing and replication tend to lead programmers to use more structured, sometimes more primitive algorithms with simpler orchestration. However, the advantage becomes less significant when asynchronous message passing is used, in which case separate event synchronization must be employed anyway to preserve correctness.

3.6.6 Hardware Cost and Design Complexity

The hardware cost and design time required to efficiently support the desirable features of a shared address space are greater than those required to support a message-passing abstraction. Since all memory transactions must be observed to determine when nonlocal cache misses occur, at least some functionality of the communication assist must be integrated quite closely into the processing node. A system with transparent replication and coherence in hardware caches requires further hardware support and the implementation of fairly complex coherence protocols. In the message-passing abstraction the assist does not need to see memory references and can be less closely integrated, for example, on the I/O bus. The actual hardware cost and complexity for supporting the different abstractions are discussed in Chapters 5, 7, and 8.

Cost and complexity, however, are more complicated issues than assist hardware cost and design time. For example, if the amount of replication needed in a message-passing program is indeed much larger than that needed in a cache-coherent shared address space (due to differences in how replacement is managed, as discussed earlier, or due to replication of the operating system), then the memory required for this replication should be compared to the hardware cost of supporting a shared address space. The same goes for the recurring cost or “design time” of developing effective programs on a machine. The design cost of protocols also diminishes with growing experience. In practice, cost and price are also determined largely by volume of sales, engineering design experience, and business rather than purely technical factors.

3.6.7 Performance Model

Finally, in designing parallel programs for an architecture we would like to have at least a rough performance model that we can use to predict whether one implementation of a program will be better than another and to guide the structure of communication. A performance model has three aspects. First, we must model the characteristics of the machine; for example, the key system granularities and the costs of primitive events, such as communication messages. Second, we must model the characteristics of the application; for example, the frequency and burstiness of the primitive events in the parallel program. And third, we must develop an analytical or numerical performance model that takes these two sets of characteristics as inputs and predicts the execution time. Modeling machine characteristics is usually not very difficult, and we have seen a simple model of communication cost in this chapter. Modeling application characteristics, however, can be quite difficult, especially when the application is complex and irregular. And developing a good analytical performance model is difficult when contention is a significant issue. It is the difficulty of modeling application characteristics that makes predicting performance in a shared address space more difficult than in message passing, since the events of interest are not explicit in the program. For a programmer, the performance guidelines in message passing are at least clear: messages are expensive; send them infrequently. In a shared address space, particularly one with coherent replication, performance modeling is complicated by the very same properties that make developing a program easier: naming, replication, and coherence are all implicit (i.e., transparent to the programmer), so it is difficult to determine how much communication occurs and when. Artifactual communication is also implicit and is particularly difficult to predict. (Consider cache mapping conflicts that generate communication!) The resulting programming guidelines are much more vague: try to exploit temporal and spatial locality and use data layout when necessary to keep communication levels low. The problem is similar to how implicit caching makes performance difficult to predict even on a uniprocessor, thus complicating the use of the simple von Neumann model of a computer, which assumes that all memory references have equal cost. However, it is of far greater magnitude here since the cost of communication is much larger than that of local memory access on a uniprocessor, and there is much greater opportunity for contention.

3.6.8 Summary

The major potential advantages of implicit communication in the shared address space model are programming ease and performance in the presence of fine-grained data sharing (at least when the model is supported in hardware). The major potential advantages of explicit communication, as in message passing, are the benefits of block data transfer, the fact that synchronization may be subsumed in message passing, better performance guidelines and prediction ability, and the ease of building machines.

Given these trade-offs, the questions that an architect has to answer are

- Is it worthwhile to provide hardware support for a shared address space (i.e., transparent naming), is software support enough, or is it easy enough for programmers to manage all communication explicitly?
- If a shared address space is worthwhile, is it also worthwhile to provide hardware support for transparent replication and coherence?
- If the answer to either of the preceding questions is yes, then is the implicit communication enough or should there also be hardware support for explicit message passing among processing nodes that can be used when desired?

The answers to these questions depend on both application characteristics and cost. Affirmative answers to any of the questions naturally lead to other questions regarding how efficiently the feature should be supported and at what granularities, which raises other sets of cost, performance, and programming trade-offs that will become clearer as we proceed through the book. Experience shows that as applications become more complex and irregular, the usefulness of transparent naming and replication increases, which argues for supporting a shared address space abstraction. However, since communication is naturally fine grained—especially in irregular applications—and since large granularities of communication and coherence cause performance problems, supporting a shared address space effectively requires an aggressive communication architecture with hardware support for most functions. Many computer companies are now building such machines as their high-end parallel systems. On the other hand, clusters of inexpensive workstations or multiprocessors are also increasingly popular. These systems are usually programmed using message passing because of its better-defined performance model, the tendency to use larger messages and amortize overhead, and the explicit control and lack of sensitivity to fixed-size machine granularities.

3.7 CONCLUDING REMARKS

The characteristics of parallel programs have important implications for the design of multiprocessor architectures. Certain key observations about program behavior led to some of the most important advances in uniprocessor computing: the recognition of temporal and spatial locality in program access patterns led to the design of caches, and an analysis of instruction usage led to streamlined instruction set design. In multiprocessors, the performance penalties for mismatches between application requirements and what the architecture provides are much larger, so it is all the more important that we understand the parallel programs and other workloads that are going to run on these machines.

Historically, many different parallel architectural genres led to many different programming styles and very little portability. Today, the architectural convergence has led to a common ground for the development of portable software environments and programming languages. The way we think about the parallelization process and many of the key performance issues is largely similar in both the shared address

space and message-passing programming models, although the specific granularities, performance characteristics, and orchestration techniques are different. While we analyze the trade-offs between shared address space and message passing, both are flourishing in different portions of the architectural design space.

Another effect of architectural convergence has been a clearer articulation of the performance issues against which software must be designed. Historically, the major focus of theoretical parallel algorithm development has been the PRAM model, which ignores data access and communication cost and considers only load balance and extra work (some variants of the PRAM model capture some serialization effects when different processors try to access the same data word). The PRAM model is very useful in understanding the inherent concurrency in an application, which is the first conceptual step in developing a parallel program; however, it does not take important realities of modern systems into account, such as the fact that data access and communication costs are often the dominant components of execution time. Historically, communication has been treated separately, and the major focus in its treatment has been mapping the communication to different network topologies. With a clearer understanding of the importance of communication and the important costs in a communication transaction on modern machines, two things have happened. First, models that help analyze communication cost and hence improve the structure of communication have been developed, such as the bulk synchronous programming (BSP) model (Valiant 1990) and the LogP model (Culler et al. 1993), with the hope of replacing the PRAM as the de facto model used for parallel algorithm analysis. These models strive to expose the important costs associated with a communication event—such as latency, bandwidth, or overhead—as we have done in this chapter, allowing an algorithm designer to factor them into the comparative analysis of parallel algorithms. The BSP model also provides an elegant framework that can be used to reason about communication and parallel performance. Second, the emphasis in modeling communication cost has shifted to the cost at the nodes that are the endpoints of the communication message, so the number of messages and contention at the endpoints have become more important than mapping to network topologies. In fact, both the BSP and LogP models ignore network topology completely, modeling network delay as a constant value!

Models such as BSP and LogP are important steps toward a realistic architectural model against which to design and analyze parallel algorithms. By changing the values of the key parameters in these models, we may be able to determine how an algorithm would perform across a range of architectures and how it might be best structured for different architectures or for portable performance. However, much more difficult than modeling the architecture as a set of parameters is modeling the behavior of the parallel algorithm or application, particularly when it is not regular in structure, which is the other side of the modeling equation (Singh, Rothberg, and Gupta 1994). The key questions here include the following: What is the communication-to-computation ratio? How does it change with replication capacity? How do the access patterns interact with the granularities of the extended memory hierarchy? How bursty is the communication? And how can this be incorporated into the performance

model? Modeling techniques that can capture these characteristics for realistic applications and integrate them with machine models like BSP or LogP have yet to be developed.

This chapter has discussed some of the key performance properties of parallel programs and their interactions with the basic provisions of a multiprocessor's extended memory hierarchy and communication architecture. These properties include load balance; the communication-to-computation ratio; aspects of orchestrating communication that affect communication cost; data locality and its interactions with replication capacity and with the granularities of allocation, transfer, and coherence to generate artifactual communication; and the implications for communication abstractions and the hardware/software interface that a machine may support. We have seen that the performance issues trade off with one another and that the art of producing a good parallel program lies in obtaining the right compromise between conflicting demands. Programming for performance is also a process of successive refinement; decisions made in the early steps may have to be revisited based on system or program characteristics discovered in later steps. Achieving the performance potential can take considerable effort, depending on both the application and the system. Further, the extent and manner in which different techniques are incorporated can greatly affect the characteristics of the workload presented to the architecture. We have examined in depth the four application case studies that were introduced in Chapter 2 and have seen how these issues play out in each of them. We shall encounter several of these performance issues again in more detail as we consider architectural design options, trade-offs, and evaluation in the rest of the book. However, with the knowledge of parallel programs that we have developed, we are now ready to understand how to use the programs as workloads to evaluate parallel architectures and trade-offs.

3.8 EXERCISES

- 3.1 For which of the applications that we have described (Ocean, Barnes-Hut, Raytrace, Data Mining) have we followed the view of decomposing data rather than computation and using an owner computes rule in our parallelization? What would be the problem(s) with using a strict data distribution and owner computes rule in the others? How would you address the problem(s)?
- 3.2 What are the advantages and disadvantages of using distributed task queues (as opposed to a global task queue) to implement load balancing? Do small tasks inherently increase communication, contention, and task management overhead in each case?
- 3.3 Draw one arc from each kind of memory system traffic (the list on the left) to the solution technique (on the right) that is the most effective way to reduce that source of traffic in a machine that supports a shared address space with physically distributed memory.

Kinds of Memory System Traffic

- Cold-start traffic
- Inherent communication
- Extra data communication on a miss
- Capacity-generated communication
- Capacity-generated local traffic

Solution Techniques

- Large cache sizes
- Data placement
- Algorithm reorganization
- Larger cache block size
- Data structure reorganization

- 3.4 Under what conditions would the sum of busy-useful time across processes (in the execution time breakdowns) not equal the busy-useful time for the sequential program, assuming both the sequential and parallel programs are deterministic? Provide examples.
- 3.5 As an example of hierarchical parallelism, consider an algorithm frequently used in medical diagnosis and economic forecasting. The algorithm propagates information through a network or graph such as the one in Figure 3.22. Every node represents a matrix of values. The arcs correspond to dependences between nodes and are the channels along which information must flow. The algorithm starts from the nodes at the bottom of the graph and works upward, performing matrix operations at every node encountered along the way. It affords parallelism at a minimum of two levels: nodes that do not have an ancestor-descendent relationship in the traversal can be computed in parallel, and the matrix computations within a node can be parallelized as well. How would you parallelize this algorithm, and what characteristics of the network or graph would most affect your decisions? What are the trade-offs that are most important?
- 3.6 To illustrate levels of parallelism, the chapter described an application that routes wires to connect pins in a VLSI chip or board. Three levels of parallelism are available: across wires, across segments within a wire that each touch only a pair of pins, and across the set of possible routes evaluated for a given segment. What are the trade-offs in determining which level to pick? What parameters of the input and of the machine affect your decision? What you would you pick for this case: 30 wires, 24 processors, 5 segments per wire, and 10 routes per segment, with each route evaluation taking the same amount of time? If you had to pick one level of parallelism and be tied to it for all cases, which would you pick? (You can make and state reasonable assumptions to guide your answer.)
- 3.7 If E is the set of sections of the algorithm that are enhanced through parallelism, f_k is the fraction of the sequential execution time taken up by the k th enhanced section when run on a uniprocessor, and s_k is the speedup obtained through parallelism on the k th enhanced section, derive an expression for the overall speedup obtained. Apply it to the broadcast approach for Gaussian elimination at element granularity. Draw a rough concurrency profile for the computation (a graph showing the amount of concurrency versus time where the unit of time is a logical operation, say, updating an interior active element). Assume a 100×100 element matrix. Estimate the speedup, ignoring memory referencing and communication costs.

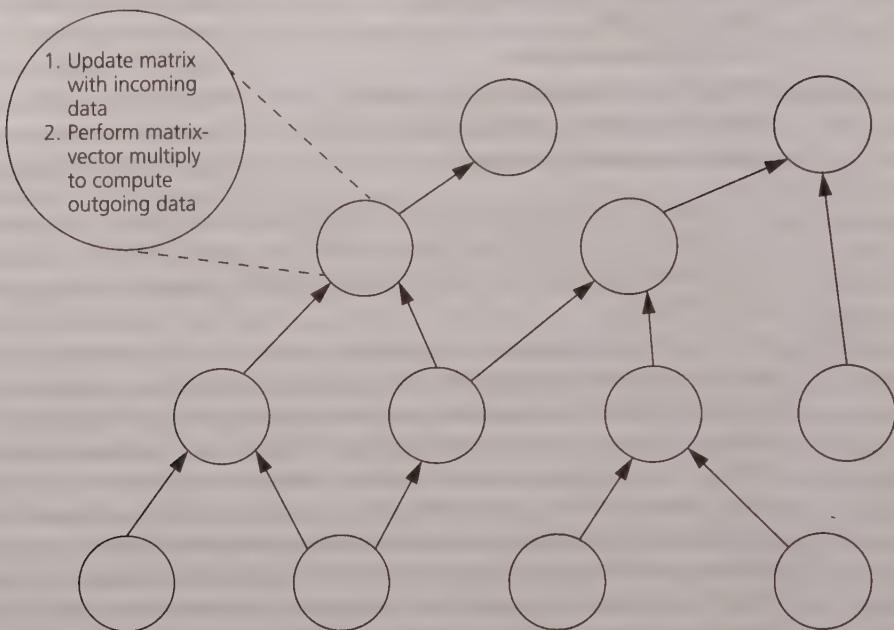


FIGURE 3.22 Levels of parallelism in a graph computation. The work within a graph node is shown in the expanded node on the left.

- 3.8 Consider the parallel Gaussian elimination algorithms discussed in Exercises 2.7–2.10.
- Draw a concurrency profile showing the available concurrency versus time for the “broadcast” version. Assume that each update to a grid element is a single unit of time and of computation.
 - For an n -by- n matrix and p processes, analyze the load imbalance and communication volume assuming an assignment in contiguous chunks of rows to processes.
 - Do the same for an interleaved assignment of rows to processes.
 - Now do the same for the pipelined version where the decomposition is still in rows.
- 3.9 The concurrency in Gaussian elimination can also be enhanced by decomposing into individual elements rather than rows. Why is this?
- Draw a concurrency profile for the broadcast version in this case.
 - A two-dimensional scatter (two-dimensional interleaved or cookie-cutter) assignment can be used at the granularity of individual elements instead of assignment in rows. Analyze the load imbalance and communication volume in the broadcast version in this case.

- c. Analyze the load imbalance and communication volume for a pipelined version assuming a two-dimensional interleaved assignment of elements. Is there a large difference now in load imbalance and communication volume compared to a broadcast version with the same assignment?
 - d. Which of the versions discussed in this and the previous exercise do you think would actually perform best on a real machine and why? Can you think of a better decomposition and assignment?
- 3.10 We have discussed the technique of blocking that is widely used in linear algebra algorithms to exploit temporal locality (see Section 3.3.1). Consider a sequential Gaussian elimination program.
- a. Write a blocked sequential version, using B -by- B blocks.
 - b. Provide an analytical expression for the read-miss rate for both the original (unblocked) and blocked sequential programs on a system in terms of n and B . Assume that in the unblocked version, a row of the matrix does not fit in the cache, while in the blocked version, B is chosen so that a B -by- B block is sized so that it fits in about half the cache. Ignore cache conflicts, and count only access to matrix elements. What would the read-miss rate be in the two cases with a cache size of 16 KB, a matrix size of 1,024-by-1,024 elements, and a block size of $B = 32$. Assume no reuse of blocks across block operations. If read misses cost 50 cycles, what is the performance difference between the two versions (counting each grid point update computation as one cycle and ignoring write accesses)?
 - c. How would you partition the blocked version for parallel execution, assuming a broadcast approach? Write pseudocode, treating the computation for a block as a single pseudo-operation.
 - d. Analyze the load imbalance and communication for this case, and compare with the previous partitioning approaches for the broadcast approach.
 - e. Considering all performance issues, not just algorithmic ones, would you use the best blocked or unblocked versions for the parallel broadcast approach on a shared address space machine? Which would you use for a message-passing machine? Why?
 - f. Considering pipelined approaches as well (with or without blocking), which approach would you choose overall for both a shared address space machine and a message-passing machine?
- 3.11 Termination detection is an interesting aspect of task stealing. Consider a task-stealing scenario in which processes produce tasks as the computation is ongoing. Design a good tasking method (where to take tasks from, how to put tasks into the pool, etc.) and think of some good termination detection heuristics. Perform worst-case complexity analysis for the number of messages needed by the termination detection methods you consider. Which one would you use in practice? Write pseudocode for one that is guaranteed to work and should yield good performance.

- 3.12 Consider transposing a matrix in parallel from a source matrix to a destination matrix (i.e., $B[i,j] = A[j,i]$).
- How might you partition the two matrices among processes? Discuss some possibilities and the trade-offs. Does it matter whether you are programming a shared address space or message-passing machine?
 - Why is the interprocess communication in a matrix transpose called all-to-all personalized communication?
 - Write simple pseudocode for the parallel matrix transposition in a shared address space and in message passing (just the loops that implement the transpose). What are the major performance issues you consider in each case other than inherent communication and load balance, and how do you address them?
 - Is there any benefit to blocking the parallel matrix transpose? Under what conditions? How would you block it? (It is not necessary to write out the full code.) What, if anything, is the difference between blocking here and in Gaussian elimination?
- 3.13 The communication needs of applications, even expressed in terms of bytes per instruction, can help us do back-of-the-envelope calculations to determine the impact of increased bandwidth or reduced latency. For example, a Fast Fourier Transform (FFT) is an algorithm that is widely used in digital signal processing and climate modeling applications. A simple parallel FFT on n data points has a per-process computation cost of

$$O\left(\frac{n \log n}{p}\right)$$

and per-process communication volume of $O(n/p)$, where p is the number of processes. The communication-to-computation ratio is therefore $O(1/\log n)$. Suppose for simplicity that all the constants in the preceding expressions are unity and that we are performing an $n = 1 \text{ M}$ (or 2^{20}) point FFT on $p = 1,024$ processes. Let the average communication latency for a word of data (a point in the FFT) be 200 processor cycles, and let the communication bandwidth between any node and the network be 100 MB/s. Assume no load imbalance or synchronization cost, and ignore contention in the network.

- With no latency hidden, for what fraction of the execution time is a process stalled due to communication latency?
- What would be the impact on execution time of halving the communication latency?
- What are the node-to-network bandwidth requirements without latency hiding?
- What are the node-to-network bandwidth requirements assuming all latency is hidden, and does the machine satisfy them? If it does not, then what (qualitatively) will be the impact?

- 3.14 Consider the use of replication to reduce data traffic.
- What kind of data (local, nonlocal, or both) can constitute the relevant working set in (i) main memory in a message-passing abstraction? (ii) a processor cache in a message-passing abstraction? (iii) a processor cache in a cache-coherent shared address space abstraction?
 - A proposal for cache-coherent machines has been to provide hardware support for fine-grained, coherent replication in main memory as well. Do you think this would be worthwhile? Under what conditions, and what do you think are the main drawbacks? For which of the case study applications in this chapter is it likely to be beneficial?
- 3.15 Write pseudocode for a reduction and a broadcast among p processes: first using a linear, $O(p)$ method and then using a tree-based, $O(\log p)$ method. Do this both for a shared address space and for message passing.
- 3.16 Write the equation solver kernel in a shared address space using a four-dimensional array representation for the grid in a manner such that the shape of the contiguous partitions (e.g., strips versus blocks or the number of processes along each dimension of the grid) can be specified as program input.
- 3.17 After the assignment of tasks to processors, the issue of scheduling the tasks that a process is assigned in some temporal order still remains. What are the major issues involved here? Which are the same as on uniprocessor programs and which are different? Construct examples that highlight the impact of poor scheduling in the different cases.
- 3.18 In the Data Mining case study, why are itemsets of size two computed from the original format of the database rather than from the transformed format? Analyze the computational complexity in each case.
- 3.19 You have been given the job of creating a word count program for a major book publisher. You will be working on a shared memory multiprocessor with 32 processors. Your only stated interface is `get_words`, which takes as its parameter an array and on return places in the array the book's next 1,000 words to be counted. The main work each processor will do should look like this:

```

while(get_words(word)) {
    for (i=0; i<1000; i++) {
        if word[i] is in list
            increment its count
        else
            add word to list
    }
}
/*Once all words have been logged, the list should be printed out*/

```

Using pseudocode, create a detailed description of the control flow and data structures that you will use for this parallel program. Your method should attempt to minimize space, synchronization overhead, and memory latency. This problem allows you a lot of flexibility, so state all assumptions and design decisions.

Workload-Driven Evaluation

The field of computer architecture is becoming increasingly quantitative. Design features are adopted only after detailed evaluations of trade-offs have been made. Once systems are built, they are evaluated and compared both by architects to understand the trade-offs and by users to make procurement decisions. In uniprocessor design, a rich base of existing machines and widely used applications supports the process of identifying and evaluating trade-offs; it is one of careful extrapolation from known quantities. Designers isolate performance characteristics of the machines by using microbenchmarks—small programs that stress a particular machine feature. Popular workloads are codified in standard benchmark suites, such as the Standard Performance Evaluation Corporation (SPEC) benchmark suite (SPEC 1995) for engineering workloads, and measurements are made on a range of existing design alternatives. Based on these measurements, assessments of emerging technology, and expected changes in the requirements of applications, designers propose new alternatives. The ones that appear promising are typically evaluated through simulation. First, a *simulator*—a program that simulates the design with and without the proposed feature of interest—is written. Then a number of programs or multiprogrammed workloads are chosen, either from the standard benchmark suites or other workloads representative of those that are likely to run on the machine. These workloads are run through the simulator and the performance impact of the feature determined. This, together with the estimated cost of the feature in hardware and design time, determines whether the feature will be included. Simulators are written to be flexible so that organizational and performance parameters can be varied to understand their impact as well.

Good workload-driven evaluation is a difficult and time-consuming process, even for uniprocessor systems. The workloads need to be renewed as technology and usage patterns change. Industry-standard benchmark suites are revised every few years. In particular, the input data sets used for the programs affect many of the key interactions with the systems and determine whether or not the important features of the system are stressed. These interactions must be understood and reflected in the use of the workloads. For example, to take into account the huge increases in processor speeds and changes in cache sizes, a major change from the SPEC92 benchmark suite to the SPEC95 suite was the use of larger input data sets to stress the memory system. Also, accurate simulators are costly to develop and verify, and the simulation runs consume huge amounts of computing time. However, these efforts are well rewarded because good evaluation yields good design.

As multiprocessor architecture has matured and greater continuity has been established from one generation of machines to the next, a similar quantitative approach has been adopted. Whereas early parallel machines were in many cases like bold works of art, relying heavily on the designer's intuition, modern design involves considerable evaluation of proposed design features. Here too, workloads are used both to evaluate real machines as well as to extrapolate to proposed designs and explore trade-offs through software simulation. For multiprocessors, the workloads of interest are either parallel programs or multiprogrammed mixes of sequential and parallel programs. Evaluation is a critical part of the new engineering approach to multiprocessor architecture; it is very important to understand the key evaluation issues before examining the core of multiprocessor architecture or the trade-offs evaluated in this book.

Unfortunately, the job of workload-driven evaluation for multiprocessor architecture is even more difficult than for uniprocessors, for several reasons:

- *Immaturity of parallel applications.* It is not easy to obtain "representative" workloads for multiprocessors, both because their use is relatively immature and because there are many new behavioral characteristics to represent.
- *Immaturity of parallel programming languages.* The software model for parallel programming has not stabilized, and programs written assuming different models can have very different behaviors.
- *Sensitivity of behavioral differences.* Different workloads, and even different decisions made in parallelizing the same sequential workload, can present vastly different execution characteristics to the architecture.
- *New degrees of freedom.* There are several new degrees of freedom in the architecture. The most obvious is the number of processors. Others include the organizational and performance parameters of the extended memory hierarchy, particularly the communication architecture. Together with the degrees of freedom of the workload (i.e., application parameters) and the underlying uniprocessor node, these parameters lead to a very large design space for experimentation, particularly when evaluating an idea or trade-off in a general context rather than evaluating a fixed machine. The high cost of communication makes performance much more sensitive to interactions among all these degrees of freedom than it is in uniprocessors, making it all the more important that we understand how to navigate the large parameter space.
- *Limitations of simulation.* Simulating multiprocessors in software to evaluate design decisions is more resource intensive than simulating uniprocessors. Multiprocessor simulations consume a lot of memory and time. Thus, although the design space we wish to explore is larger, the space that we can actually explore is often much smaller, and we must make careful trade-offs in deciding which parts of the space to simulate.

Our understanding of parallel programs from Chapters 2 and 3 will be critical in dealing with these difficulties. Throughout this chapter, we will learn that effective evaluation requires understanding the important properties of both workloads and

architectures as well as how these properties interact. In particular, the relationships among application parameters and the number of processors determine fundamental program properties such as communication-to-computation ratio, load balance, and temporal and spatial locality. These properties interact with parameters of the extended memory hierarchy to influence performance in application-dependent and often dramatic ways (see Figure 4.1). Choosing workload and machine parameter values (or sizes) and understanding their scaling relationships is a crucial aspect of workload-driven evaluation, with far-reaching implications. It affects the experiments we design for adequate coverage of behavioral characteristics as well as the conclusions of our evaluations, and it helps us restrict the number of experiments or parameter combinations we must examine.

An important goal of this chapter is to highlight the key interactions of these properties and parameters, to illustrate their significance, and to point out the important pitfalls. Although no universal formula exists for evaluation, the chapter articulates a methodology for both evaluating real machines and assessing trade-offs through simulation. This methodology is followed in characterizing several workloads at the end of this chapter and in the illustrative evaluations that use these workloads throughout the book. It is important that we not only perform good evaluations but also understand the limitations of evaluation studies so we can keep them in perspective as we make architectural decisions.

The chapter begins by discussing the fundamental issue of scaling workload parameters as the number of processors increases and considers the implications for performance metrics and for the key inherent behavioral characteristics of parallel programs. The interactions with organizational and performance parameters of the extended memory hierarchy, and how these interactions should be incorporated into the actual design of experiments, are discussed in the next two sections, which examine the two major types of evaluations.

Section 4.2 outlines a methodology for evaluating a real machine. This involves first understanding the types of benchmark workloads we might use and their roles in such evaluation—including microbenchmarks, kernels, applications, and multi-programmed workloads—as well as desirable criteria for choosing them. Then, given a workload, we examine how to choose its parameters to evaluate a given machine, illustrating the important considerations and pitfalls. The section ends with a discussion of various metrics that we might use to interpret and present results. Section 4.3 extends this methodological discussion to the more challenging problem of evaluating an architectural trade-off in a more general context through simulation.

Having understood how to perform workload-driven evaluation, we move on to Section 4.4, which provides the relevant characteristics of the workloads that will be used in the illustrative evaluations presented in the book. Some important publicly available workload suites for parallel computing, together with their philosophies, are described in the Appendix.

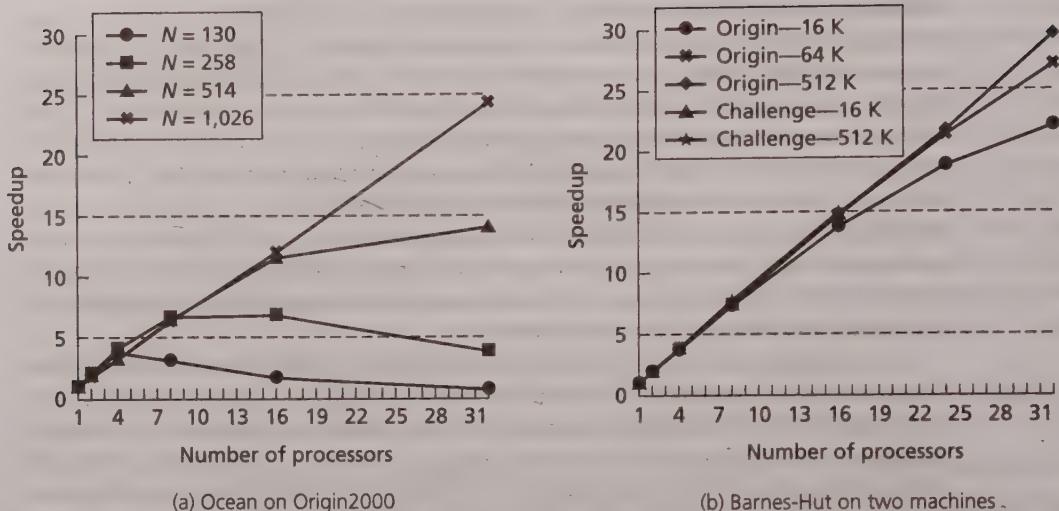


FIGURE 4.1 Impact of application parameters on parallel performance. For Ocean, the application parameter shown is the number of grid points (N) in each dimension, while in Barnes-Hut it is the number of bodies. These parameters determine the size of the data set used. For many applications, like Ocean in (a), the effect is dramatic, at least until the data set size becomes large enough for the number of processors. For the smallest problem, performance becomes worse rather than better in going from 4 to 8 processors and beyond; for the second smallest problem, performance drops when going from 8 to 16 processors; while for the largest problem, performance increases roughly linearly with processor count all the way to 32 processors. For other applications, like Barnes-Hut in (b), the effect of data set size is much smaller.

4.1 SCALING WORKLOADS AND MACHINES

Let us begin by discussing some basic measures of performance on a multiprocessor and using them to motivate the importance of proper scaling, before we examine scaling models and their implications.

4.1.1 Basic Measures of Multiprocessor Performance

Suppose we have chosen a parallel program as a workload and we want to use it to evaluate a machine. For a parallel machine, we can measure two performance characteristics: the *absolute performance* and the *performance improvement due to parallelism*. The latter is typically measured as the speedup, which was defined in Chapter 1 as the absolute performance achieved on p processors divided by that achieved on a single processor. Absolute performance (together with cost) is most important to the end user or buyer of a machine. However, in itself it does not tell us a great deal about how much of the performance comes from the use of parallelism and the

effectiveness of the communication architecture rather than from the performance of an underlying single-processor node. Speedup tells us how much of the performance comes from the use of parallelism but with the caveat that it is easier to obtain good speedup when the individual nodes have lower performance since communication costs are less important when computation is slower. Both metrics are important, and both should be measured.

Absolute performance is best measured as work done per unit of time. Given a program, the amount of work to be done is usually defined by the input configuration on which the program operates, which is called the problem size (we shall define problem size more precisely later). This input configuration may either be available to the program up front, or it may consist of a set of continuously arriving inputs to a “server” application, such as a system that processes a bank’s transactions or responds to inputs from sensors. Suppose the input configuration, and hence work, is kept fixed for a set of experiments. We can then treat the work as a fixed point of reference, measure the execution time, and define performance as the reciprocal of execution time.

In some application domains, users find it more convenient to have an explicit, domain-specific representation of work and use an explicit work-per-unit-time performance metric even when the input configuration is fixed. For example, in a transaction processing system, the metric could be the number of transactions serviced per minute; in a sorting application, the number of keys sorted per second; and in a chemistry application, the number of bonds computed per second. However, even though work is explicitly represented, performance is measured with reference to a particular input configuration or amount of work, and these performance metrics are nonetheless derived from measurements of execution time (together with the number of application events of interest). Given a fixed and known problem configuration, these domain-specific metrics present no fundamental advantage over execution time or its reciprocal. In fact, we must be careful to ensure that the explicit measure of work being used is indeed a meaningful measure from the application perspective, not something that we can cheat against. We discuss desirable properties of work metrics further as we go along and consider the more detailed issues concerning metrics in Section 4.2.5. For now, let us focus on evaluating the improvement in absolute performance due to parallelism, that is, the speedup due to using p processors instead of one.

Using execution time as our performance metric, we saw in Chapter 1 that we could simply run the program with the same input configuration on one and p processors and measure the improvement or speedup as

$$\frac{\text{Time}(1\text{ proc})}{\text{Time}(p\text{ procs})}$$

With operations per second as the performance metric, we can measure speedup as

$$\frac{\text{Operations per Second}(p\text{ procs})}{\text{Operations per Second}(1\text{ proc})}$$

A question arises about how we should measure performance on one processor; for example, is it more accurate to use the performance of the best sequential program running on one processor rather than the parallel program itself running on one processor? But this is quite easily addressed. As the number of processors is changed, we can simply run the problem on the different numbers of processors and compute speedups accordingly. Why then all the fuss about scaling?

4.1.2 Why Worry about Scaling?

Unfortunately, there are several reasons why measuring speedup with a fixed problem size is insufficient as the *only* way of evaluating the performance improvement due to parallelism across a range of machine scales.

Suppose the fixed problem size we have chosen is relatively small and is appropriate for a machine with a few processors. As we increase the number of processors for the same problem size, the overheads due to parallelism (communication, load imbalance) increase relative to useful computation. A point will come when the problem size is unrealistically small to evaluate the machine at hand. The high overheads will lead to uninterestingly small speedups, which reflect not so much the capabilities of the machine as the fact that an inappropriate problem size was used (say, one that does not have enough concurrency for the large machine). In fact, at some point using more processors may even hurt performance as the overheads begin to dominate useful work (see Figure 4.2[a]). A user would not run this problem on a machine that large, so it is not appropriate for evaluating this machine. The same is true if the problem takes a very small amount of time on the large machine.

On the other hand, if we choose a problem that is realistic for a machine with many processors, we might have the opposite problem in evaluating the performance improvement due to parallelism. This problem may be too big for a single processor because its data is too large to fit in the memory of a single node. On some machines, it may not be runnable on a single processor; on others, the uniprocessor execution will thrash severely to disk; and on still others, the overflow data will be allocated in other nodes' memories in the extended hierarchy, leading to a lot of artifactual internode communication. When enough processors are used, the data will fit in their collective memories, eliminating this artifactual communication if the data is distributed properly. The computation on each processor will be more efficient, and the result is a speedup far beyond the number of processors used. Once this has happened, further improvements in speedup will behave in a more usual way as the number of processors is increased, but the speedup over a uniprocessor is still *superlinear* in the number of processors.

This situation holds for any level of the memory hierarchy, not just main memory. For example, the aggregate cache capacity of the machine grows as each processor with its own cache hierarchy is added. If the working set per processor diminishes along with the data set, processors begin to use their caches more efficiently as the number of processors increases. An example using cache capacity is illustrated for the equation solver kernel in Figure 4.2(b). This greatly superlinear speedup due to memory system effects is not fake. Indeed, from a user's perspective the availability

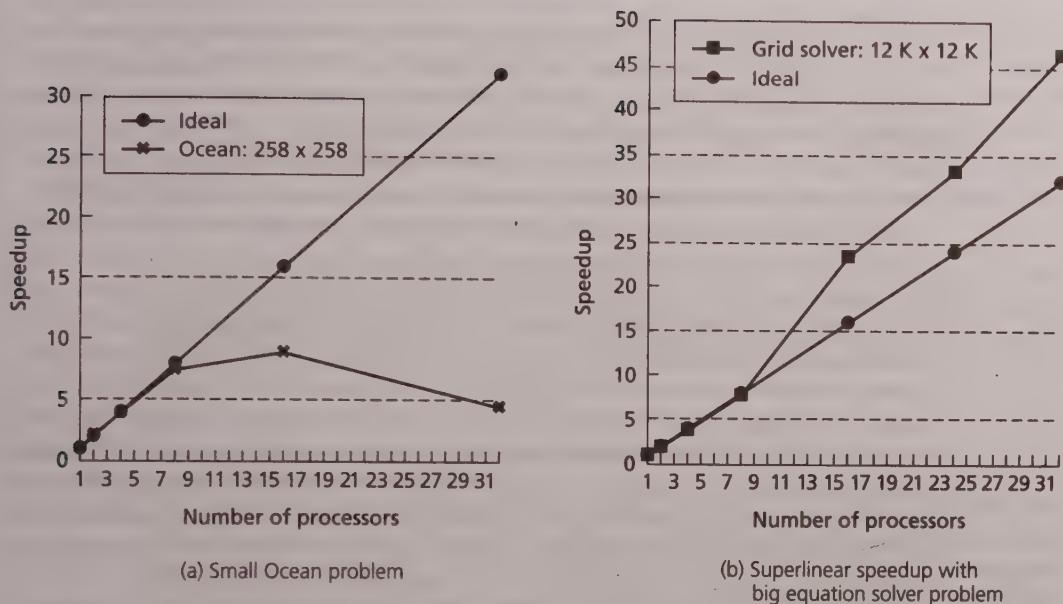


FIGURE 4.2 Speedups on the SGI Origin2000 as the number of processors increases. (a) shows the speedup for a small problem size in the Ocean application. The problem size is clearly very appropriate for a machine with about 8 processors. At a little beyond 16 processors, the speedup has saturated, and it is no longer clear that we would run this problem size on this large a machine. This is clearly the wrong problem size to run to evaluate a machine with 32 or more processors! (b) shows the speedup for the equation solver kernel, illustrating superlinear speedups when a processor's working set fits in the cache at 16 processors but does not fit when 8 or fewer processors are used.

of more, distributed memory is an important advantage of parallel systems over uniprocessor workstations since it enables them to run much larger problems and to run them much faster. However, the superlinear speedup does not allow us to separate the capacity effects from the usual improvements due to parallelism and as such does not help us evaluate the effectiveness of the machine's communication architecture.

A final limitation of maintaining the same problem size as the number of processors is increased is that this may not reflect realistic usage of the machine. Users often want to use more powerful machines to solve larger problems rather than to solve the same problem faster. In these cases, since problem size increases together with machine size in practical use of the machines, it should be scaled when evaluating the machines as well. Such scaling may overcome the problems arising from the size of mismatches just discussed, but the simplicity of comparing machine configurations on identical problems is lost.

We need well-defined scaling models for how problem size should be changed to accommodate changes in machine size so that we can evaluate machines against these models. The measure of performance is always work per unit of time, regardless of the scaling model. However, if the problem size is scaled, the work done does

not stay constant, so we can no longer simply compare execution times to determine speedup. Work must be represented and measured, and the question is how. Furthermore, we want to understand how the scaling model influences program characteristics, such as the communication-to-computation ratio, load balance, and data locality in the extended memory hierarchy. For simplicity, let us focus on a single parallel application, not a multiprogrammed workload. First we need clear definitions of terms that have been used informally: scaling a machine and problem size.

Scaling a machine means making it more (or less) powerful. This can be done by making any component of the machine bigger, more sophisticated, or faster—the individual processors, the caches, the memory, the communication architecture, or the I/O system. In general, the *machine size* is a vector characterizing the per-node processing capabilities, memory hierarchy, and communication and I/O capabilities. Scaling a machine involves changing an entry or entries in the vector. Since our interest is in parallelism, we define machine size as the number of processors, and we assume that the individual node, its local cache and memory system, and the per-node communication capabilities remain the same as the machine scaled. Scaling up a machine means adding more identical nodes. For example, scaling a machine with p processors and $p \times m$ megabytes of total memory by a factor of k results in a machine with $k \times p$ processors and $k \times p \times m$ megabytes of total memory.

Problem size refers to a specific problem instance or input configuration. It is usually specified by a vector of input parameters, not just a single parameter n (e.g., an n -by- n grid in Ocean or n particles in Barnes-Hut). For example, in Ocean the problem size is specified by a vector $V = (n, \varepsilon, \Delta t, T)$, where n is the grid size in each dimension (which specifies the spatial resolution of our representation of the ocean), ε is the error tolerance used to determine convergence of the multigrid equation solver, Δt is the temporal resolution (i.e., the physical time between time-steps), and T is the number of time-steps performed. In a transaction processing system, problem size is specified by the number of terminals used, the rate at which users at the terminals issue transactions, the mix of transactions, and so on. Problem size is a major factor that determines the work done by the program.

Problem size should be distinguished from data set size. The *data set size* is the amount of storage that would be needed to run the program on a single processor. This is itself distinct from the *memory usage* of the program, which is the amount of memory used by the parallel program including replication. The data set size typically depends on a small number of program parameters. In Ocean, for example, the data set size is determined solely by the grid size n . The number of instructions and the execution time, however, depend on the other problem size parameters as well. Thus, while the problem size vector V determines many important properties of the application program—such as its data set size, the number of instructions it executes, and its execution time—it is not identical to any one of these properties.

4.1.3 Key Issues in Scaling

Given these definitions, there are two major questions to address when scaling a problem to run on a larger machine:

1. Under what constraints should the problem be scaled? To define a scaling model, some property must be kept fixed as the machine scales. These properties might include data set size, memory usage per processor, execution time, number of transactions executed per second, and number of particles or rows of a matrix assigned to each processor.
2. How should the problem be scaled? That is, how should the parameters in the problem size vector V be changed to meet the chosen constraints?

To simplify the discussion, we begin by pretending that the problem size is determined by a single parameter n and examine scaling models and their impact under this assumption. Later, in Section 4.1.6, we examine the more subtle issue of scaling workload parameters relative to one another.

4.1.4 Scaling Models and Speedup Measures

The properties used as the basis for scaling constraints can be divided into two categories: *user-oriented properties* and *resource-oriented properties*. Examples of user-oriented properties are the number of particles per processor in Barnes-Hut, the number of rows of a matrix per processor in a matrix multiplication program, the number of transactions issued to the system per processor in transaction processing, and the number of I/O operations performed per processor. Examples of resource-oriented constraints are execution time and the total amount of memory used per processor. Each of these constraints defines a distinct scaling model, since the amount of work done for a given number of processors is different when scaling is performed under different constraints. Whether user- or resource-oriented constraints are more appropriate depends on the application domain. A critical job in constructing benchmarks is to ensure that the scaling constraints are meaningful for the domain at hand.

User-oriented constraints are usually much easier to follow when performing evaluations (e.g., simply change the number of particles linearly with the number of processors). However, large-scale programs are often run under tight resource constraints, and resource constraints are more universal across application domains (time is time and memory is memory regardless of whether the program deals with particles or matrices). We will therefore use resource constraints to illustrate the effects of scaling models. Let us examine the three most popular resource-oriented models for the constraints under which an application should be scaled to run on a k times larger machine: *problem-constrained* (PC) scaling, *time-constrained* (TC) scaling, and *memory-constrained* (MC) scaling.

In PC scaling, the problem size is kept fixed; that is, it is not scaled at all, despite the concerns discussed earlier regarding a fixed problem size. The same input configuration is used regardless of the number of processors on the machine. In TC scaling, the wall-clock execution time needed to complete the program is held fixed. The problem is scaled so that the new problem's execution time on the large machine is the same as the old problem's execution time on the small machine (Gustafson 1988). In MC scaling, the amount of main memory used per processor is

held fixed. The problem is scaled so that the new problem uses exactly k times as much main memory (including data replication) as the old problem. Thus, if the old problem just fit in the memory of the small machine, then the new problem will just fit in the memory of the large machine.

More specialized models are more appropriate in some domains. For example, in its commercial on-line transaction processing benchmark, the Transaction Processing Council (TPC) dictates a scaling rule in which the number of user terminals that generate transactions and the size of the database being accessed are scaled proportionally with the “computing power” of the system being evaluated, measured in a specified way. In this as well as in the TC and MC scaling models, scaling to meet resource constraints often requires some experimentation to find the appropriate input since resource usage may not scale in a simple way with input parameters. Memory usage is often quite predictable—especially if there is no need for replication in main memory—but it is difficult to predict the input configuration that would take the same execution time on 256 processors that another input configuration took on 16. Let us look at each of PC, TC, and MC scaling a little further and see what “work per unit of time,” and hence speedup, translates to under them.

Problem-Constrained Scaling

The assumption in PC scaling is that a user wants to use the larger machine to solve the same problem faster. This is not an unusual situation. For example, if a video compression algorithm handles only one frame per second, our goal in using parallelism may not be to compress a larger image in 1 second but rather to compress 30 frames per second and hence achieve real-time compression for that frame size. As another example, if a VLSI routing tool takes a week to route a complex chip, we may be more interested in using additional parallelism to reduce the routing time rather than to route a larger chip. Since useful work in the work/time definition of performance remains fixed, the formulation of the speedup metric is simply

$$\text{Speedup}_{PC}(p \text{ processors}) = \frac{\text{Time}(1 \text{ processor})}{\text{Time}(p \text{ processors})} \quad (4.1)$$

Time-Constrained Scaling

This model assumes that users have a certain amount of time that they can wait for a program to execute, and they want to solve the largest possible problem in that fixed amount of time. (Think of a user who can afford to buy eight hours of computer time at a computer center or one who is willing to wait overnight for a run to complete but needs to have the results ready to analyze the next morning.) Whereas in PC scaling the problem size is kept fixed and the execution time varies, in TC scaling the problem size increases but the execution time is kept fixed. Since performance is work divided by time and since time stays fixed as the system is scaled, speedup can be measured as the increase in the amount of work done in that fixed execution time:

$$\text{Speedup}_{TC}(p \text{ processors}) = \frac{\text{Work}(p \text{ processors})}{\text{Work}(1 \text{ processor})} \quad (4.2)$$

The question is how to measure work. If we measure it as actual execution time for that problem configuration on a single processor, then we would have to run the larger (scaled) problem size on a single processor of the machine to obtain the numerator. Unfortunately, for interesting problems this is likely to thrash, take a long time, or be impossible to run.

Desirable properties for a work metric are that it should be easy to measure and as architecture independent as possible. Ideally, it should be easily modeled with an analytical expression based only on the application, and we should not have to perform any additional experiments to measure the work in the scaled-up problem. The measure of work should also scale linearly with sequential time complexity of the algorithm (see Example 4.1).

EXAMPLE 4.1 Why is the linear scaling property important for a work metric?

Answer The linear scaling property is important if we want the ideal speedup (ignoring memory system artifacts) to be proportional to the number of processors. To see this, suppose we use as our work metric in a matrix multiplication program the number of rows n in square matrices. Let us ignore memory system interactions completely. If the uniprocessor problem has n_0 rows, then its execution “time,” or the number of multiplication operations it needs to execute, will be proportional to n_0^3 . Since the problem is deterministic, the best we can hope p processors to do in the same time is $n_0^3 \times p$ operations, which corresponds to $(n_0 \times \sqrt[3]{p})$ -by- $(n_0 \times \sqrt[3]{p})$ matrices. If we measure work as the number of rows, then the speedup according to Equation 4.2 even in this idealized case will be $(n_0 \times \sqrt[3]{p})/n_0$ or $\sqrt[3]{p}$ instead of p . Using the number of points in the matrix (n^2) as the work metric also does not work from this perspective, since it would result in an ideal time-constrained speedup of $p^{2/3}$. However, using n^3 (the number of multiplication operations) as the work metric leads to an ideal speedup of p , since this measure scales linearly with the $O(n^3)$ sequential time complexity of matrix multiplication. ■

The ideal work measure not only satisfies both of these properties but is also an intuitive parameter from a user’s perspective. For example, in sorting integer keys using a method called radix sorting, the sequential complexity grows linearly with the number of keys to be sorted, so we can use keys as the measure of work. However, such a measure is difficult to find in real applications, particularly when multiple application parameters are scaled and affect execution time in different ways. So how should we measure work in practice?

If a single intuitive parameter that has the desirable properties cannot be found, we can try to find a measure that can be easily derived from an intuitive parameter and that scales linearly with the sequential complexity. The popular LINPACK benchmark, which performs matrix factorization, does this. It is known that the benchmark should take $2n^3/3$ floating-point operations to factorize an n -by- n matrix, and the rest of the operations are either proportional to or completely dominated by these. As with matrix multiplication in Example 4.1, this number of operations is easily computed from the input matrix dimension n and clearly satisfies the linear scaling property, so it is used as the measure of work for the benchmark.

Real applications often have multiple parameters to scale and are therefore more complex. As long as we have a well-defined rule for scaling parameters at the same time, we may be able to construct an analytical measure of work that has the desired properties. However, such work counts may no longer be simple or intuitive, and they ask a lot of the evaluator or the benchmark provider. Furthermore, analytical predictions are usually simplified in complex applications (e.g., they are the average case, or they do not reflect “implementation” activities that can be quite significant), so the actual growth rates of instructions or operations executed can be different than expected.

In such cases, a generally applicable, empirical technique is to run the sequential program and measure the work in machine operations. If a certain type of high-level operation, such as a particle-particle interaction, is known to always be directly proportional to the sequential complexity, then we can count the number of operations executed at run time. More generally, we may arrange to measure the time taken to run the problem on a uniprocessor, assuming that all memory references are cache hits and take the same amount of time (say, a single cycle), thus eliminating artifacts due to the memory system. This work measure reflects which machine instructions are actually executed when running the program, yet avoids the thrashing and superlinearity problems; we call it the *perfect-memory execution time*. (Notice that it corresponds very closely to the sequential *busy-useful* time introduced in Section 3.4.) Many computers have system utilities that allow computations to be profiled and to obtain this perfect-memory execution time. If not, we must resort to measuring how many times some high-level operation occurs.

Once we have a work measure, we can compute speedup under TC scaling as in Equation 4.2. However, determining the input configuration that yields the desired execution time and hence satisfies TC scaling may take some iterative refinement.

Memory-Constrained Scaling

This model is motivated by the assumption that the user wants to run the largest problem possible without overflowing the machine’s memory, regardless of execution time. For example, it might be important for an astrophysicist to run an n -body simulation like Barnes-Hut with the largest number of bodies that the machine can accommodate in order to increase the resolution with which the bodies sample the universe. Results presented for MC scaling have often used a performance improvement metric called *scaled speedup*, which is defined as the ratio of the time that the larger (scaled) problem would take to run on a single processor to the time that it takes on the scaled machine. This metric is often attractive to vendors because such speedups tend to be high. Effectively, it measures the problem-constrained speedup on a very large problem, which tends to have a low communication-to-computation ratio and abundant concurrency and also to benefit from superlinearity effects due to memory and cache capacity. The scaled problem is not what we run on a uniprocessor anyway under MC scaling, so this is not an appropriate speedup metric.

Unlike the previous models, under MC scaling neither work nor execution time is held fixed. Using work divided by time as the performance metric as always, we can define speedup as

$$\begin{aligned}\text{Speedup}_{MC}(p \text{ processors}) &= \frac{\text{Work}(p \text{ processors})}{\text{Time}(p \text{ processors})} \times \frac{\text{Time}(1 \text{ processor})}{\text{Work}(1 \text{ processor})} \\ &= \frac{\text{Increase in Work}}{\text{Increase in Execution Time}}\end{aligned}\quad (4.3)$$

If the increase in execution time were only due to the increase in work and not due to overheads of parallelism—and if there were no memory system artifacts, which are usually less likely under MC scaling—then the speedup would be p , which is what we want. Work is measured as discussed previously for TC scaling.

Since data set size grows faster under MC scaling than under other models, parallel overheads grow relatively slowly and speedups often tend to be better (ignoring capacity artifacts). MC scaling is indeed how many users desire to use a parallel machine. However, for many types of applications, MC scaling leads to a serious problem: the execution time (for the parallel execution) can become intolerably large. This problem can occur in any application where the work done grows more rapidly with problem size than the memory usage (see Example 4.2).

EXAMPLE 4.2 Matrix factorization is a simple example in which the serial work grows more rapidly than the memory usage. Show how MC scaling leads to a rapid increase in parallel execution time for this application.

Answer While the data set size and the memory usage for an n -by- n matrix grow as $O(n^2)$ in matrix factorization, the execution time on a uniprocessor grows as $O(n^3)$. Assume that a $10,000 \times 10,000$ matrix takes about 800 MB of memory and can be factorized in 1 hour on a uniprocessor. Now consider a scaled machine consisting of 1,000 processors. On this machine, under MC scaling we can factorize a $320,000 \times 320,000$ matrix since little or no replication is needed in main memory. However, the execution time of the parallel program (even assuming perfect, thousand-fold speedup) will now increase to about 32 hours. ■

Of the three models, time-constrained scaling is increasingly recognized as being the most generally viable. However, no model can be claimed to be the most realistic for all applications and all users. Different users have different goals, work under different constraints, and are in any case unlikely to follow a given model very strictly. Nonetheless, these three models are useful, comprehensive tools for an analysis of scaled performance as machines scale.

4.1.5 Impact of Scaling Models on the Equation Solver Kernel

Let us now examine a simple example—the equation solver kernel from Chapter 2—to see how it interacts with different scaling models and how they affect its architecturally relevant behavioral characteristics. For an n -by- n grid, the memory requirement of the simple equation solver is $O(n^2)$. Computational complexity is $O(n^2)$

times the number of iterations to convergence, which we can conservatively assume to be $O(n)$ (the number of iterations taken for values to flow from one boundary of the grid to the other). This leads to a sequential computational complexity of $O(n^3)$.

Consider the execution time and memory requirements under the three scaling models, assuming speedups due to parallelism equal to the number of processors p in all cases. With PC scaling, as the same n -by- n grid is divided among more processors p , the memory requirements per processor decrease linearly with p , as does the execution time. In TC scaling, the execution time stays the same by definition. Assuming linear speedup, this means that if the scaled grid size is k -by- k , then $k^3/p = n^3$, so $k = n \times \sqrt[3]{p}$. The amount of memory needed per processor is therefore

$$\frac{k^2}{p} = \frac{n^2}{\sqrt[3]{p}}$$

which diminishes as the cube root of the number of processors. Using MC scaling, by definition, the memory requirements per processor stay the same at $O(n^2)$ where the base grid for the single processor execution is n -by- n . This means that the overall size of the grid increases by a factor of p , so the scaled grid is now $n\sqrt{p}$ -by- $n\sqrt{p}$ rather than n -by- n . Since it now takes $n\sqrt{p}$ iterations to converge, the sequential time complexity is $O((n\sqrt{p})^3)$. This means that even assuming perfect speedup due to parallelism, the execution time of the scaled problem on p processors is

$$O\left(\frac{(n\sqrt{p})^3}{p}\right)$$

or $n^3\sqrt{p}$. Thus, the parallel execution time is greater than the sequential execution time of the base problem by a factor of \sqrt{p} . Even under the linear speedup assumption, a problem that took 1 hour on one processor takes 32 hours on a 1,024-processor machine under MC scaling. For this simple equation solver, then, the execution time increases quickly under MC scaling, and the memory requirements per processor decrease under TC scaling.

Let us consider the effects of different scaling models on the concurrency, communication-to-computation ratio, synchronization and I/O frequency, temporal and spatial locality, and message size (in message passing).

The concurrency in this kernel is proportional to the number of grid points. It remains fixed under PC scaling, grows proportionally to p under MC scaling, and grows proportionally to $p^{0.67}$ under TC scaling.

The communication-to-computation ratio is the perimeter-to-area ratio of the grid partition assigned to each processor; that is, it is inversely proportional to the square root of the number of points per processor (n^2/p). Under PC scaling, the ratio grows as \sqrt{p} . Under MC scaling, the size of a partition does not change, so neither does the communication-to-computation ratio. Finally, under TC scaling, since the size of a processor's partition diminishes as the cube root of the number of processors, the ratio increases as the sixth root of p .

The equation solver synchronizes at the end of every grid sweep to determine convergence. Suppose that it also performed I/O then, for example, outputting the maximum error at the end of each sweep. Under PC scaling, the work done by each processor in a given sweep decreases linearly as the number of processors increases, so assuming linear speedup, the frequency of synchronization and I/O grows linearly with p . Under MC scaling the frequency remains fixed, and under TC scaling it increases as the cube root of p .

The size of the important working set, which indicates its temporal locality, in this equation solver is exactly the size of a processor's partition of the grid. Therefore, it and the cache requirements diminish linearly with p under PC scaling, stay constant under MC scaling, and diminish as the cube root of p under TC scaling. Thus, although the aggregate problem size grows under TC scaling, the working set size of each processor diminishes.

Spatial locality in the equation solver is best within a processor's partition and at row-oriented boundaries and worst at column-oriented boundaries. Thus, it decreases as a processor's partition becomes smaller and column-oriented boundaries become larger relative to partition area. It therefore remains constant under MC scaling, decreases quickly under PC scaling, and decreases less quickly under TC scaling.

Finally, an individual message in a message-passing model is likely to be a border row or column of a processor's partition, which is the square root of partition size. Hence, message size here scales similarly to the communication-to-computation ratio. The number of messages a process sends, however, depends only on the number of neighbor processes and is independent of n , p , or scaling model.

It is clear from the preceding discussion that as long as memory or cache capacity effects do not dominate, we should expect the lowest parallelism overhead and highest speedup under MC scaling and the next under TC scaling. We should expect speedups to degrade quite quickly under PC scaling, at least once the overheads become significant relative to useful work. It is also clear that the choice of application parameters and the scaling model greatly affect both fundamental program characteristics and architectural interactions with the extended memory hierarchy, such as spatial and temporal locality. Unless it is known that a particular scaling model is the right one for an application, or is particularly inappropriate, it is useful to evaluate a machine under all three scaling models. We examine the interactions with architectural parameters and their importance for evaluation in more detail when we discuss actual evaluations (Sections 4.2 and 4.3). First, let us take a brief look at the other important but more subtle aspect of scaling: how to scale application parameters to meet the constraints of a given scaling model.

4.1.6 Scaling Workload Parameters

In discussing the constraints under which problem size should be scaled, we made the simplifying assumption of a single application parameter n and did not examine how different application parameters that constitute the problem size vector should

be scaled relative to one another to meet the chosen constraint. Let us now take away this simplifying assumption. For instance, the Ocean application has a vector of four parameters: n , ϵ , Δt , and T . How workload parameters should be scaled relative to one another is not an issue under PC scaling, but it is under TC or MC scaling. Different parameters are often related to one another, and it may not make sense to scale one of them without scaling others, or to scale them independently. For example, in a realistic usage of the Barnes-Hut application, the parameters θ (the force calculation accuracy) and Δt (the physical interval between time-steps) should be scaled as n (the number of bodies) changes. All of these parameters may contribute to a given execution characteristic; for example, the execution time of Barnes-Hut grows not simply as $n \log n$ but as

$$\frac{1}{\theta^2 \Delta t} n \log n$$

As a result, the increase in the number of bodies n under TC scaling is not as large as would be inferred by scaling only n .

Even the simple equation solver kernel has another parameter, ϵ , which is the tolerance used to determine convergence of the solver. Making this tolerance smaller—as should be done as n scales in a real application—increases the number of iterations needed for convergence and, hence, increases execution time, but it does not affect the memory requirements. Compared with scaling only n , scaling ϵ and n causes the per-process grid size, memory requirements, and working set size to decrease much more quickly under TC scaling, the communication-to-computation ratio to increase more quickly under TC scaling but still remain unchanged under MC scaling, and the execution time to increase even more quickly under MC scaling. As architects using workloads, it is very important that we understand the relationships among parameters from an application user’s viewpoint and scale the parameters in our evaluations according to this understanding. Otherwise, we are liable to arrive at incorrect architectural conclusions.

The actual relationships among parameters and the rules for scaling them depend on the domain and the application. There are no universal rules, which makes good evaluation even more interesting. For example, in applications like Barnes-Hut and Ocean that model physical phenomena through discretization, the different application parameters usually govern different sources of error in the accuracy with which the phenomenon (such as galaxy evolution) is modeled; appropriate rules for scaling these parameters together are therefore driven by guidelines for scaling the different types of error. Ideally, benchmark suites will describe the scaling rules—and may even encode them in the application, leaving only one free parameter like n —so the architect as a user of the benchmarks does not have to worry about learning them. Exercises 4.12 and 4.13 illustrate the importance of proper application scaling by showing that scaling parameters appropriately can often lead to quantitatively and sometimes even qualitatively different architectural results than scaling only the data set size parameter n .

4.2 EVALUATING A REAL MACHINE

Now that we understand the importance of proper scaling and the effects that problem and machine size have on fundamental behavioral characteristics and architectural interactions, we are ready to develop specific guidelines for the two major types of workload-driven evaluation: evaluating a real machine and evaluating an architectural idea or trade-off in a general context. Evaluating a real machine is in many ways simpler: the organization, granularities, and performance parameters of the machine are fixed, and all we have to worry about is choosing appropriate workloads and workload parameters; also, we are not constrained by the limitations of software simulation. This section provides a prescriptive template for evaluating a real machine. We begin with the use of microbenchmarks to isolate performance characteristics. Then we look at the major issues in choosing workloads for an evaluation. This topic is followed by guidelines for evaluating a machine once a workload is chosen—first when the number of processors is fixed and then when it is allowed to be varied. The section concludes with a discussion of popular metrics for measuring the performance of a machine and for presenting the results of an evaluation. All these issues in evaluating a real machine are relevant to evaluating an architectural idea or trade-off as well.

4.2.1 Performance Isolation Using Microbenchmarks

A first step in evaluating a real machine is to understand its basic performance capabilities—that is, the performance characteristics of the primitive operations provided by the programming model, communication abstraction (user/system interface), or hardware/software interface. This is usually done with small, specially written programs called *microbenchmarks* (Saavedra, Gaines, and Carlton 1993) that are designed to isolate these performance characteristics (for example, latencies, bandwidths, overhead, etc.).

Five types of microbenchmarks are used in parallel systems; the first three are also used for uniprocessor evaluation:

1. *Processing microbenchmarks* measure the performance of the processor on operations that do not access memory, such as arithmetic operations, logical operations, and branches.
2. *Local memory microbenchmarks* determine the organization, latencies, and bandwidths of the levels of the memory hierarchy within the local node and measure the performance of local read and write operations satisfied at different levels, including those that cause TLB misses and page faults.
3. *Input-output microbenchmarks* measure the characteristics of I/O operations, such as disk reads and writes of various strides and lengths.

4. *Communication microbenchmarks* measure data communication operations, such as message sends and receives or remote reads and writes of different types.
5. *Synchronization microbenchmarks* measure the performance of different types of synchronization operations, such as locks and barriers.

The communication and synchronization microbenchmarks depend on the communication abstraction or programming model used. They may involve one or a pair of processors—for example, a single remote read miss, a send/receive pair, or the acquisition of a free lock—or they may be collective, such as broadcast, reduction, all-to-all communication, probabilistic communication patterns, many processors contending for a lock, or barriers. Different microbenchmarks may be designed to stress uncontended latency, bandwidth, overhead, and contention.

For measurement purposes, microbenchmarks are usually implemented as repeated sets of the primitive operations (e.g., 10,000 remote reads in a row). They often have simple parameters that can be varied to obtain a fuller characterization, for example, the number of participating processors in a collective communication microbenchmark or the stride between consecutive reads in a local memory microbenchmark. Figure 4.3 shows a typical profile of a machine obtained using a local memory microbenchmark. The main role of microbenchmarks is to isolate and understand the performance of basic system capabilities. A more ambitious hope, not achieved so far, is that if workloads can be characterized as weighted sums of different primitive operations, then a machine's performance on a given workload can be predicted from its performance on the corresponding microbenchmarks. We discuss some specific microbenchmarks and issues in designing them when we measure real systems in later chapters.

Having isolated the performance characteristics, the next step is to evaluate the machine on more realistic workloads. We must navigate three major axes: the workloads, their problem sizes, and the number of processors (or the machine size). Lower-level machine parameters are fixed. Let us begin with choosing workloads for an evaluation.

4.2.2 Choosing Workloads

Beyond microbenchmarks, workloads used for evaluation can be divided into three classes in increasing order of realism and complexity: kernels, complete applications, and multiprogrammed workloads. Each has its own role, advantages, and disadvantages.

Kernels are well-defined parts of real applications but are not complete applications themselves. They can range from simple kernels, such as a matrix transposition or a near-neighbor grid sweep, to more complex, substantial kernels that dominate the execution times of their applications, such as matrix factorization and iterative methods that solve partial differential equations. Examples of kernels for information processing include complex database queries used in decision support applications or sorting a set of numbers. Kernels expose higher-level interactions

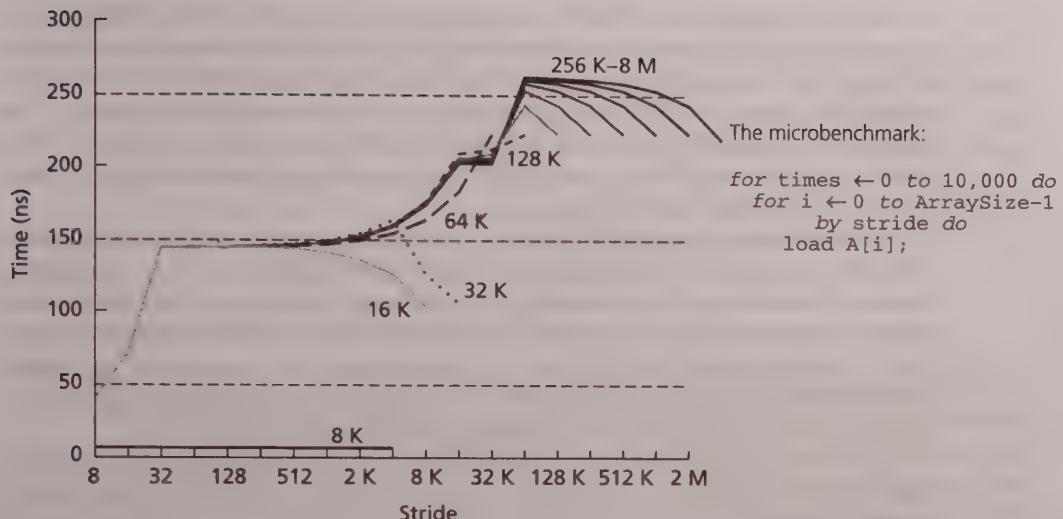


FIGURE 4.3 Results of a microbenchmark experiment on a single processing node of the CRAY T3D multiprocessor. Every processor has a small, single-level cache backed up by local main memory. The microbenchmark consists of a large number of reads from a local array. The y-axis shows the time per read in nanoseconds. The x-axis is the stride between successive reads in the loop (i.e., the difference in the addresses of the memory locations being accessed). The different curves correspond to and are labeled with the size of the array (ArraySize) being strided through. When ArraySize is less than 8 KB, the array fits in the processor cache so that all reads are hits and take 6.67 ns to complete. For larger arrays, we see the effects of cache misses. The average access time is the weighted sum of hit and miss time, until there is an inflection when the stride is longer than a cache block (32 words or 128 bytes) and every reference misses. The next rise occurs as a result of some references causing page faults, with an inflection when the stride is large enough (16 KB) that every consecutive reference causes a page fault. The final rise is due to conflicts at the memory banks in the four-bank main memory, with an inflection at 64-K stride when consecutive references always hit the same bank and the other banks remain idle.

that are not present in microbenchmarks and, as a result, lose a degree of performance isolation. Their key property is that their performance-relevant characteristics—communication-to-computation ratio, concurrency, and working sets, for example—can be easily understood and often analytically determined, so that observed performance as a result of the interactions can be explained in light of these characteristics.

Complete applications consist of multiple kernels and exhibit higher-level interactions among kernels that an individual kernel cannot reveal. Unlike kernels, complete applications are run by users to obtain an answer that they care to look at. The same large data structures may be accessed in different ways by multiple kernels in an application, and different data structures accessed by different kernels may interfere with one another in the memory hierarchy. In addition, the data structures that are optimal for a kernel in isolation may not be best in the complete application. The same holds for partitioning techniques. For example, if there are two independent

kernels in an application, then we may decide not to partition each among all processes but rather to share processes among them. Different kernels that share a data structure may be partitioned in ways that strike a balance between their different access and communication patterns, leading to the maximum overall locality. The presence of multiple kernels in an application introduces many subtle interactions, and the performance-related characteristics of complete applications usually cannot be exactly determined analytically.

Multiprogrammed workloads consist of multiple sequential and parallel applications that run together on the machine. The different applications may either time-share the machine or *space-share* it (i.e., different applications run on disjoint subsets of the machine's processors) or both, depending on the operating system's multiprogramming policies. Just as whole applications are complicated by higher-level interactions among the kernels that comprise them, multiprogrammed workloads involve complex interactions among whole applications themselves.

As we move from kernels to complete applications and multiprogrammed workloads, we gain in realism, which is very important. Many critical bugs and performance problems are not revealed by microbenchmarks and even kernels but are discovered by these workloads. However, we lose in our ability to describe the workloads concisely, to explain and interpret the results unambiguously, and to isolate performance factors. In the extreme, multiprogrammed workloads are difficult not only to interpret but also to design: which applications should be included in such a workload and in what proportion? It is also difficult to obtain repeatable results from multiprogrammed workloads because of subtle timing-dependent interactions with the operating system. Each type of workload has its place. However, the higher-level interactions exposed only by complete applications and multiprogrammed workloads (and the fact that they are the workloads that will actually be run on the machine by users) make it important that we use them to ultimately determine the overall performance of a machine.

Let us examine the desirable properties in choosing such workloads (applications, multiprogrammed loads, and even complex kernels) for an evaluation. These properties include representativeness of application domains, coverage of behavioral properties, and adequate concurrency.

Representativeness of Application Domains

If we are performing an evaluation as users looking to procure a machine, and we know that the machine will be used to run only certain types of applications, then choosing a representative workload is easy. On the other hand, if the machine may be used to run a wide range of workloads, or if we are designers trying to evaluate a general-purpose machine to learn lessons for the next generation, we should choose a mix of workloads representative of a wide range of domains.

Some important domains for parallel computing today include scientific applications that model physical phenomena; engineering applications such as those in computer-aided design, digital signal processing, automobile crash simulation, and

even simulations used to evaluate architectural trade-offs; graphics and visualization applications that render scenes or volumes into images; media processing applications such as image, video, and audio analysis and processing, and speech and handwriting recognition; information management applications such as databases, data mining, and transaction processing; optimization applications such as crew scheduling for an airline and transport control; artificial intelligence applications such as expert systems and robotics; multiprogrammed workloads; and a multiprocessor operating system, which is itself a complex parallel application.

Coverage of Behavioral Properties

Workloads may vary substantially along the entire range of performance-related characteristics discussed in Chapter 3. As a result, a major problem in evaluation is that it is very easy to lie with, or be misled by, workloads. For example, a study may choose workloads that stress the feature for which an architecture has an advantage (say, communication latency) but not those aspects it performs poorly (say, local access, contention, or communication bandwidth). For general-purpose evaluation, it is important that the workloads we choose, taken together, stress a range of important performance characteristics. For example, we should choose workloads with low and high communication-to-computation ratios, small and large working sets, regular and irregular access patterns, and localized and long-range or collective communication. If we are especially interested in evaluating particular architectural characteristics, such as aggregate bandwidth for all-to-all communication among processors, then we should choose at least some workloads that stress those characteristics.

Another important issue is the level of program optimization. Real parallel programs will not always be highly optimized for good performance along the lines discussed in Chapter 3, not just for the specific machine at hand but even in more general ways like reducing the communication-to-computation ratio or increasing temporal and spatial locality. This may be either because the effort involved in optimizing programs is more than the user is willing to expend or because the programs are generated with the help of automated parallelization tools. The level of optimization can greatly affect key execution characteristics and hence the degree to which architectural capabilities are stressed. In particular, four types of optimization are important to consider:

1. *Algorithmic.* The decomposition and assignment of tasks may be less than optimal—for example, strip-oriented versus block-oriented assignment for a grid computation (see Section 2.3.3)—and certain algorithmic enhancements for data locality, such as blocking, may not be implemented.
2. *Data structuring.* The data structures used may not interact optimally with the architecture, increasing artifactual communication—for example, two-dimensional versus four-dimensional arrays to represent a two-dimensional grid in a shared address space (see Section 3.3.1).

3. *Data layout, distribution, and alignment.* Even if appropriate data structures are used, they may not be distributed or aligned appropriately to pages or cache blocks, causing excess local traffic or artifactual communication.
4. *Orchestrating of communication and synchronization.* The resulting communication and synchronization may be structured in less than optimal ways—for example, sending small instead of large messages in message passing.

While optimizations can often be ad hoc, these categories impose some structure. Where appropriate, we should compare the robustness of machines or features to workloads with different levels of optimization.

Concurrency

The dominant performance bottleneck in a workload may be the computational load imbalance, either inherent to the partitioning method or due to the way synchronization is orchestrated (e.g., using barriers instead of point-to-point synchronization). If this is true, then the workload may not be appropriate for evaluating a machine's communication architecture since the architecture can do little about this bottleneck; even great improvements in communication performance may not affect overall performance much. In order to evaluate communication architectures, we should ensure that our workloads and their problem sizes exhibit adequate concurrency and load balance. A useful concept here is that of *algorithmic speedup*—the speedup that assumes that all memory references and communication operations take zero time (see the discussion of the PRAM architectural model in Chapter 3). By completely ignoring the performance impact of data access and communication, algorithmic speedup measures the computational load balance in the workload, together with the extra work done in the parallel program.

In general, we should isolate performance limitations due to workload characteristics that a machine cannot do much about from those that it can. It is also important that the workload take long enough to run to be realistic for a machine of the size being evaluated, though both this and concurrency are often more a function of the input problem size than of the workload itself.

Many efforts have been made to define standard benchmark suites of parallel applications to facilitate workload-driven architectural evaluation, taking some of the preceding criteria into account. The benchmark suites cover different application domains and have different philosophies; some of them are described in the Appendix. While the workloads used for the illustrative evaluations in the book are a very limited set, they are chosen with the preceding criteria in mind. For now, let us assume that a particular parallel program has been chosen as a workload and see how we might use it to evaluate a real machine. First the number of processors is kept fixed, which both simplifies the discussion and exposes the important interactions more cleanly. Then the number of processors is varied.

4.2.3 Evaluating a Fixed-Size Machine

Having fixed the workload and the machine size, we only have to choose the workload parameters. We have already seen that, for a fixed number of processors, changing the problem size can dramatically affect all the important execution characteristics and hence the results of an evaluation. In fact, it may even change the nature of the dominant bottleneck—that is, whether it is communication, load imbalance, or local data access. This already tells us a most significant but often ignored point: it is usually insufficient to use only a single problem size in an evaluation, even when the number of processors is fixed.

We can use our understanding of application-architecture interactions to choose problem sizes for a study. Our goal is to obtain adequate coverage of realistic inherent behaviors and architectural interactions while at the same time restricting the number of problem sizes we need. We do this in a set of structured steps, demonstrating the pitfalls of choosing only a single size in the process. The discussion will proceed one step at a time. In each step, the simple equation solver kernel will be used to illustrate the issues quantitatively. For the quantitative illustration, let us assume that we are evaluating a cache-coherent shared address space machine with 64 single-processor nodes, each with 1 MB of cache and 64 MB of main memory. The steps are as follows.

Step 1: Determine a Range of Problem Sizes

One way to choose problem sizes, applicable in some fortunate cases, is to appeal to higher powers. The high-level goals of the study may choose the problem sizes for us. For example, we may know that users of the machine are interested in only a few specified problem sizes. This simplifies our job but is uncommon and is not a general-purpose methodology. It does not apply to the equation solver kernel.

Knowledge of real usage may identify a range below which problems are unrealistically small for the machine at hand and above which the execution time is too large or users would not be interested. This too is not particularly useful for the equation solver kernel. Once we have identified a range, we can go on to the next step.

Step 2: Use Inherent Behavioral Characteristics

Inherent behavioral characteristics (such as communication-to-computation ratio and load balance) help us further restrict the range and choose problem sizes within the selected range. Since the inherent communication-to-computation ratio usually decreases with increasing data set size, large problems may not stress the communication architecture enough—at least with inherent communication—whereas small problems may stress it unrepresentatively and potentially hide other bottlenecks. Since concurrency usually increases with data set size, we would like to choose at least some problem sizes that are large enough to be load balanced but not so large

that the inherent communication becomes too small (see Example 4.3). The size of the problem may also affect the fractions of execution time spent in different phases of the application, which may have very different load balance, synchronization, and communication characteristics. For example, in the Barnes-Hut application case study, smaller problems cause more of the time to be spent in the tree-building phase, which doesn't parallelize very well and has less desirable properties than the force calculation phase that usually dominates in practice. We should be careful not to choose unrepresentative scenarios in this regard.

EXAMPLE 4.3 How would you use inherent behavioral characteristics to select a range of problem sizes for the equation solver kernel?

Answer For this kernel, enough work and load balance might dictate that we have partitions that are at least 32×32 points. For a machine with 64 (8×8) processors, this means a total grid size of at least 256×256 . This grid size requires the communication of 4×32 , or 128, grid points per process in each iteration for a computation of 32×32 or 1 K points. At five floating-point operations and 8 bytes per grid point, this is an inherent communication-to-computation ratio of 1 byte every five floating-point operations. Assuming a processor that can deliver 200 MFLOPS on this calculation, this implies a bandwidth requirement of 40 MB/s. This is quite small for modern multiprocessor networks, even if it is bursty. Let us assume that below 5 MB/s communication is asymptotically small for our system. From the viewpoint of inherent properties only, there is no need to run problems larger than 256×256 points (only $64 \text{ K} \times 8 \text{ B}$ or 512 KB of data) per processor, or $2 \text{ K} \times 2 \text{ K}$ grids overall. ■

Inherent characteristics like load balance and communication vary smoothly with problem size, so to deal with them alone, we can pick a few sizes that span the interesting spectrum. If their rate of change is very slow, we may not need to choose many sizes. Experience shows that about three is a good number in most cases. For example, for the equation solver kernel we might have chosen 256×256 , $1 \text{ K} \times 1 \text{ K}$, and $2 \text{ K} \times 2 \text{ K}$ grids.

On the other hand, the interactions of temporal and spatial locality with the architecture exhibit thresholds in their performance effects, including the generation of artifactual communication as problem size changes. We may need to extend our choice of problem sizes to obtain enough coverage with respect to these thresholds. At the same time, the threshold nature can help us prune the parameter space. The next step in choosing problem sizes is to examine temporal locality and working sets.

Step 3: Use Temporal Locality and Working Sets

Working sets fitting or not fitting in a local cache or replication store can dramatically affect execution characteristics, such as local memory traffic and artifactual communication, even if the inherent communication and computational load balance do not change much. In applications like Raytrace, the important working sets are large and consist of data that is mostly assigned to remote nodes, so artifactual communication due to limited replication capacity may dominate inherent commu-

nication. This artifactual communication tends to grow rather than diminish with increasing problem size. In other applications (like Ocean), working sets not fitting in the cache can generate dramatically more local memory traffic instead of artifactual communication. We should include problem sizes that represent both sides of the threshold (fitting and not fitting) for the important working sets if such problem sizes are realistic in practice. In fact, when realistic for the application, we should also include a problem size that is very large for the machine, for example, one that almost fills the memory, even though this problem size may be uninteresting from the viewpoints of load balance and inherent communication. Large problems often exercise architectural and operating system interactions that smaller problems do not, such as TLB misses, page faults, and a large amount of traffic due to cache capacity misses. Examples 4.4 and 4.5 help illustrate how we might choose problem sizes based on working sets.

EXAMPLE 4.4 Suppose that an application has the miss rate versus cache size curve shown in Figure 4.4(a) for a fixed problem size and number of processors and for the lowest-level cache in a node of our machine (i.e., the cache that is farthest from the processor and closest to memory). If C is the cache size, how should this curve influence the choice of problem sizes used to evaluate the machine?

Answer We can see from Figure 4.4(a) that for the problem size (and number of processors) shown, the first working set fits in the cache of size C , the second fits only partially, and the third does not fit. Each of these working sets scales with problem size in its own way. This scaling determines at what problem size that working set might no longer fit in a cache of size C and therefore what problem sizes we should choose to cover the representative cases. In fact, if the curve truly consists of sharp knees, then we can draw a different type of curve, this time one for each important working set. This curve, shown in Figure 4.4(b), depicts whether or not that working set fits in our cache of size C as the problem size changes. If the problem size at which a knee in this curve occurs is within the range of problem sizes that we have determined to be realistic, then we should ensure that we include a problem size on each side of that knee. Not doing this may cause us to miss important effects related to stressing the memory or communication architecture. The fact that the curves are flat on both sides of a knee in this example means that if all we care about from the cache is the miss rate, then we need to choose only one problem size on each side of each knee for this purpose and can prune out the rest.¹ ■

EXAMPLE 4.5 How might working sets influence our choice of problem sizes for the equation solver?

Answer The most important working sets for the equation solver are encountered when two subrows of a partition fit in the cache and when a processor's entire partition fits in the cache. Both are very sharply defined in this simple kernel. Even with the largest grid we chose based on inherent communication-to-computation ratio ($2\text{ K} \times 2\text{ K}$), the data set size per processor is only 0.5 MB, so both of these

1. Pruning a flat region of the miss rate curve is not necessarily appropriate if we also care about aspects of cache behavior other than miss rate that are also affected by cache size. We shall see an example when we discuss trade-offs among cache coherence protocols in Chapter 5.

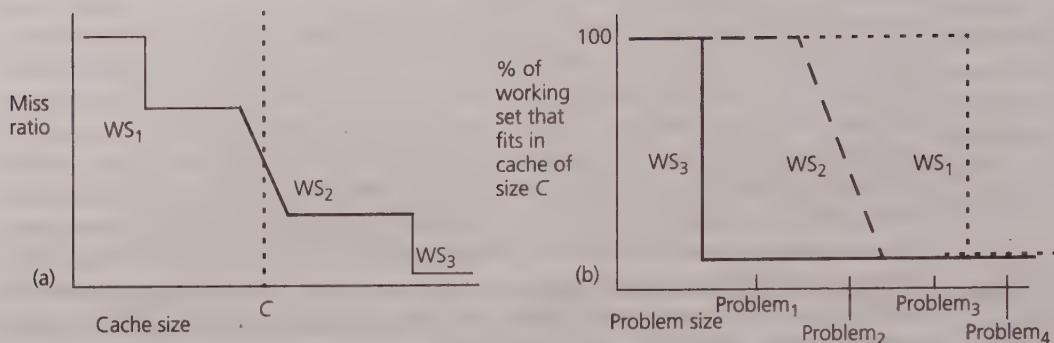


FIGURE 4.4 Choosing problem sizes based on working sets fitting in the cache. The graph in (a) shows the miss rate versus cache size curve for a fixed problem size with our chosen number of processors. C is the size of the cache or replication store under consideration. This curve identifies three knees or working sets, two very sharply defined and one less so. The graph in (b) shows, for each of the working sets, a curve depicting whether or not they fit in the cache of size C as the problem size increases. A knee in a curve represents the problem size at which that working set no longer fits. We can see that a problem of size Problem₁ fits WS₁ and WS₂ but not WS₃, Problem₂ fits WS₁ and part of WS₃ but not WS₂, Problem₃ fits WS₁ only, and Problem₄ does not fit any working set in the cache.

working sets fit comfortably in the cache (if a 4D array representation is used, there are essentially no conflict misses). Thus, we may need to choose some larger problem sizes as well. For the first working set of two subrows to exceed a 1-MB cache would imply a subrow of 64 K points, so a total grid of $64 \text{ K} \times \sqrt{64}$ or 512 K rows (or columns) for our 64-processor machine. This is a data set of 32 GB per processor, which is far too large to be realistic. However, having the other important working set—a process's whole partition—not fit in a 1-MB cache is realistic. It leads to either a lot of local memory traffic or a lot of artificial communication (if data is not placed properly) and we would like to represent such a situation. We can do this by choosing a problem size of, say, 512×512 points (2 MB) per processor or $4 \text{ K} \times 4 \text{ K}$ points overall. This does not come close to filling the machine's memory, so we might choose one more problem size for that purpose, say, $16 \text{ K} \times 16 \text{ K}$ points overall or 32 MB per processor. We now have five problem sizes: 256×256 , $1 \text{ K} \times 1 \text{ K}$, $2 \text{ K} \times 2 \text{ K}$, $4 \text{ K} \times 4 \text{ K}$, and $16 \text{ K} \times 16 \text{ K}$. ■

Step 4: Use Spatial Locality

Suppose that the data structure used to represent the grid in a shared address space implementation of the equation solver kernel is a two-dimensional array. A processor's partition, which is its important working set, may not remain in its cache across grid sweeps. Even if cache capacity is sufficient, cache conflicts may be quite frequent since the subrows of a processor's partition are not contiguous in the address space. In either case, if the working set does not fit, then it is important that a processor's partition be allocated in its local memory on a distributed-memory machine. The granularity of allocation in main memory is a page, which is typically 4–16 KB. If the size of a subrow is less than the page size, proper allocation becomes very dif-

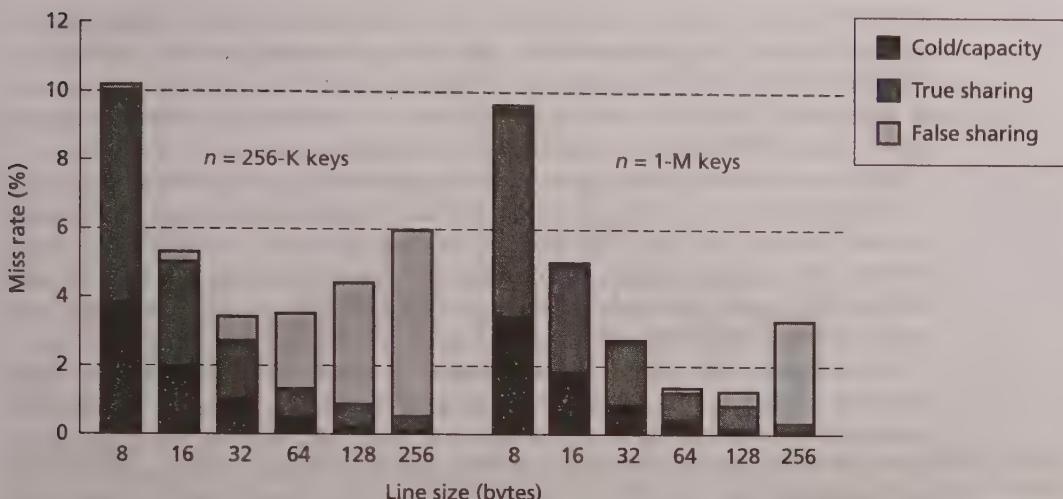


FIGURE 4.5 Impact of problem size and number of processors on the spatial locality behavior of Radix sorting. The miss rate is broken down into cold/capacity misses, true sharing (inherent communication) misses, and misses due to false sharing of data. As the block size increases for a given problem size and number of processors, there comes a point when the critical ratio discussed in the text becomes smaller than a threshold multiple of block size, and substantial false sharing is experienced. This threshold effect occurs at different block sizes for different problem sizes. A similar effect would have been observed if the problem size were kept constant and the number of processors changed.

ficult and a lot of artifactual communication may result. However, if a subrow is a multiple of the page size, allocation is not a problem and there is little artifactual communication. Both scenarios may be realistic, so we should try to represent both. If the page size is 4 KB, the first three problem sizes we have chosen so far have a subrow smaller than 4 KB, so they cannot be distributed properly; the last two have subrows greater than or equal to 4 KB, so they can be distributed well provided the grid is aligned to a page boundary. Thus, we do not need to expand our set of problem sizes for this purpose. With a 4D array representation of the grid, a process's partition of the grid is contiguous in the address space, so proper allocation is easy when partitions are large enough to make it necessary.

A more stark example of spatial locality interactions is found in a different program and architectural interaction. The program, called Radix, is a sorting program described later in this chapter, and the architectural interaction, called false sharing, was defined in Chapter 3 and is discussed further in Chapter 5. However, it is useful to look at the result here to illustrate the importance of considering spatial interactions in our choice of problem sizes. Figure 4.5 shows how the miss rate for this program running on a cache-coherent shared address space machine changes with cache block size for two different problem sizes n (sorting 256-K integers and 1-M integers) using the same number of processors p . The false sharing component of the miss rate tends to increase with cache block size. When it becomes significant, it leads to a lot of artifactual communication and can destroy the performance of this

application. For the given cache block size on our machine, false sharing may or may not destroy the performance of radix sorting depending on the problem size (compare the bars for 64-byte blocks). It turns out that for a given cache block size, false sharing is large if the ratio of problem size to number of processors is smaller than a certain threshold and insignificant if it is bigger.

Many applications display these threshold effects in spatial locality interactions with problem size; in others, especially in many irregular applications, like Barnes-Hut and Raytrace, the data structures and access patterns are such that spatial locality does not increase much with problem size. Identifying the presence of such thresholds requires understanding the application's locality and its interaction with architectural parameters and illustrates some of the subtleties in evaluation.

To summarize, the simple equation solver illustrates the dependence of many execution characteristics on problem size, some exhibiting knees on threshold in interaction with architectural parameters and some not. With n -by- n grids and p processes, if the ratio n/p is large, then the communication-to-computation ratio is low, important working sets are unlikely to fit in the processor caches leading to a high-capacity miss rate, but spatial locality is good even with a two-dimensional array representation. The situation is the opposite when n/p is small: high communication-to-computation ratio, poor spatial locality and false sharing (with the 2D representation), and few local capacity misses. The dominant performance bottleneck thus changes from local access in one case to communication in the other. Figure 4.6 illustrates these effects for the Ocean application as a whole, which uses kernels similar to the equation solver.

Other applications may exhibit different specific dependences on problem size. While there are no universal formulas for choosing problem sizes to evaluate a machine, and the equation solver kernel is a trivial example, the steps presented in this chapter provide a useful methodology and should ensure that the results obtained for a machine are not due to artifacts that can be easily removed in the program. If we are to compare two machines, it is useful to choose problem sizes that exercise the preceding scenarios on both machines. Despite the variety of issues to consider, experience shows that the number of problems sizes needed to evaluate a fixed-size machine with an application is usually quite small since there are only a few important thresholds.

4.2.4 Varying Machine Size

Now suppose we want to evaluate the machine's performance as the number of processors changes. We have already seen how to scale the problem size under different scaling models and what metrics to use for performance improvement due to parallelism. The issue that remains is how to choose the problem size, at some machine size, as a starting point from which to scale. One strategy is to start from the problem sizes we chose previously for a fixed number of processors and scale each of them up or down according to the different scaling models. We may narrow down our range of base problem sizes to three—a small, a medium, and a large—which with three

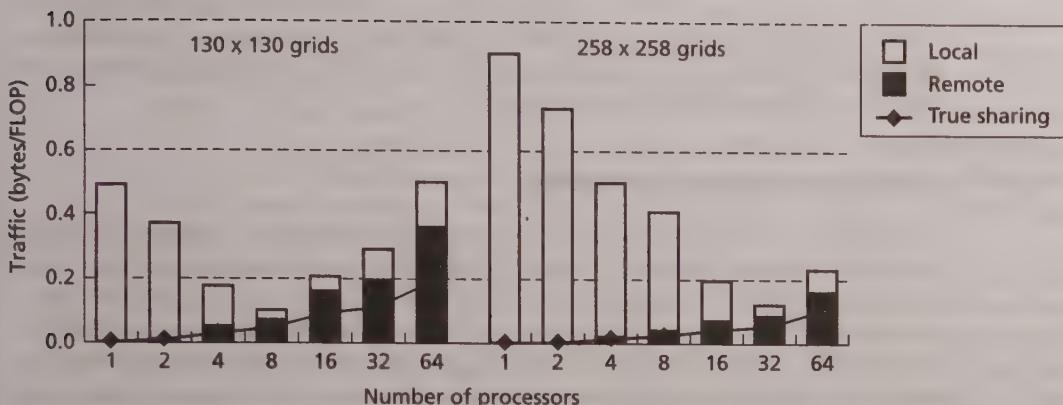


FIGURE 4.6 Effects of problem size, number of processors, and working set fitting in the cache. This figure shows the effects on the memory behavior of the Ocean application in a shared address space. The cache miss traffic (in bytes per floating-point operation or FLOP) is broken down into traffic that is local or contained in the node and traffic that is remote or traverses the network (i.e., communication). The traffic due to true sharing of data (inherent communication) is also shown separately. Remote traffic increases with the number of processors and decreases from the smaller problem to the larger. As the number of processors increases for a given problem size, the working set starts to fit in the cache, and a domination by local misses is replaced by a domination by communication. This change occurs at a larger number of processors for the larger problem since the working set is proportional to n^2/p . If we focus on the 8-processor breakdown for the two problem sizes, we see that for the small problem the traffic is dominantly remote (since the working set fits in the cache), whereas for the larger problem it is dominantly local.

scaling models will result in nine sets of performance data and speedup curves. However, it may require care to ensure that the problem sizes, when scaled down, stress the capabilities of smaller machines.

An alternative strategy is to start with a few well-chosen problem sizes on a uniprocessor and scale up under all three models. Here too, it is reasonable to choose three uniprocessor problem sizes. The small problem should be such that its working set fits in the cache on a uniprocessor. This problem will not be very useful under PC scaling on large machines but should remain fine under MC and perhaps even TC scaling. The large problem should be such that its important working set does not fit in the cache on a uniprocessor, if this is realistic for the application. Under PC scaling, the working set may fit at some point (if it shrinks with an increasing number of processors), whereas under MC scaling it is less likely to fit and is likely to keep generating capacity traffic. A reasonable choice for a large problem is one that fills most of the memory on a single node or takes a large amount of time on it. Thus, it will continue to almost fill the memory even on large systems under MC scaling. The medium-sized problem can be chosen in some judicious way in between; if possible, even it should take a substantial amount of time on the uniprocessor. The outstanding issue is how to explore PC scaling for problem sizes

that don't fit in a single node's memory without experiencing superlinear speedup problems. Here the solution is to simply choose such a problem size and measure speedup not relative to a single processor but relative to a number of processors for which the problem indeed fits in memory.

4.2.5 Choosing Performance Metrics

An important question in evaluating or comparing machines is the specific metrics that should be used. Just as it is easy to mislead by not choosing workloads and parameters appropriately, it is also easy to convey the wrong impression by not measuring and presenting results in meaningful ways. In general, both cost and performance are important metrics for comparing machines or evaluating performance. And in evaluating how well a machine scales as resources (for example, processors and memory) are added, it is not only how performance increases that matters but also how cost increases. Even if speedup increases much less than linearly, if the cost of the resources needed to run the program doesn't increase much more quickly than that, then it may indeed be cost-effective to use the larger machine (Wood and Hill 1995). Overall, some measure of "cost-performance" is more appropriate than simply performance. However, cost and performance can be measured separately, and cost is very dependent on the marketplace. The focus here is therefore on metrics for measuring performance.

Absolute performance and performance improvement due to parallelism are both useful metrics. Here, we examine the subtler issues in using these metrics to evaluate and especially compare machines and consider the role of other metrics that are based on processing rate (e.g., megaflops), resource utilization, and problem size rather than directly on work and time. Some metrics are clearly important and should always be presented, whereas the utility of others depends on what we are after and the environment in which we operate.

Absolute Performance

To a user of a system the absolute performance is the performance metric that matters the most. Suppose that execution time is our metric for absolute performance. Time can be measured in different ways. First, there is a choice between user time and wall-clock time for a workload. *User time* is the time the machine spends executing the workload, excluding system activity and other programs that might be time-sharing the machine; *wall-clock time* is the total elapsed time for the workload—including all intervening activity. Second, there is the issue of whether to use the average or the maximum execution time over all processes of the program.

Since users ultimately care about wall-clock time, we must measure and present this when comparing systems. However, if other user programs—not just the operating system—interfere with a program's execution as a result of multiprogramming, then wall-clock time does not help us understand performance bottlenecks. Note

that user time for that program may not be very useful in this case either, since interleaved execution with unrelated processes disrupts the memory system interactions of the program as well as its synchronization and load balance behavior. We should therefore always present wall-clock time and describe the execution environment (batch or multiprogrammed), whether or not we present more detailed information geared toward enhancing understanding. And if we want to understand performance on a particular application, we should run it in isolation with only the operating system perhaps intervening.

Similarly, since a parallel program is not finished until the last process has terminated, it is the time to this point that is important, not the average over processes. Averages tend to deemphasize imbalances. Of course, if we truly want to understand performance bottlenecks, we would like to see the execution profiles of all processes—or at least a sample—broken down into different components of time (Figure 3.12, for example). The components of execution time tell us why one system outperforms another and whether the workload is appropriate for the investigation (e.g., is not limited by load imbalance).

Performance Improvement or Speedup

A question in measuring speedup for any scaling model is what the denominator in the speedup ratio—the performance on one processor—should actually measure. We have four choices:

1. Performance of the parallel program on one processor of the parallel machine
2. Performance of a sequential implementation of the same algorithm on one processor of the parallel machine
3. Performance of the “best” sequential algorithm and program for the same problem on one processor of the parallel machine
4. Performance of the “best” sequential program on an agreed-upon standard machine

The difference between (1) and (2) is that the parallel program incurs overhead even when run on a uniprocessor, since it executes synchronization operations, parallelism management instructions, or partitioning code, or even the tests to omit these. This overhead can sometimes be significant. The distinction between (2) and (3) is that the best sequential algorithm may not be possible or easy to parallelize effectively, so the algorithm used in the parallel program may be different from the best sequential algorithm.

Using performance as defined by (3) clearly leads to a better and more accurate speedup metric than (1) and (2) from a user’s perspective. From an architect’s point of view, however, in many cases it may be okay to use definition (2). Definition (4) fuses the machine’s uniprocessor performance back into the picture and thus results in a comparison metric that is similar to absolute performance.

Processing Rate

A metric that is often quoted to characterize the performance of machines is the number of computer operations that they execute per unit time (as opposed to operations that have meaning at the application level, such as transactions or chemical bonds). Classic examples are MFLOPS (millions of floating-point operations per second) for floating-point-intensive programs and MIPS (millions of instructions per second) for general programs. Much has been written about why these are not good general metrics for performance even though they are popular in the marketing literature of vendors. The basic reason is that, unless we use an unambiguous machine-independent measure of the number of FLOPs or instructions that are fundamentally needed to solve a problem; rather than the number actually executed, these measures can be artificially inflated: inferior, brute-force algorithms that perform many more FLOPs and take much longer may produce higher MFLOPS ratings. In fact, we can even inflate the metric by artificially inserting useless but cheap operations in the code. If the number of operations needed is unambiguously known, then using these rate-based metrics is no different from using execution time. Other problems with MFLOPS include that different floating-point operations have different costs; that even in FLOP-intensive applications, modern algorithms use smarter data structures that have many integer operations; and that these metrics are burdened with a legacy of misuse (e.g., for publishing peak hardware rates rather than rates achieved on actual applications). When used appropriately, rate-based metrics like MFLOPS and MIPS may be useful for understanding basic hardware capabilities; however, we should be very wary of using them as the main indication of a machine's performance.

Utilization

Architects sometimes measure success by how well they are able to keep their processing engines busy executing instructions rather than stalled as a result of various overheads. It should be clear by now, however, that processor utilization is not a metric of interest to a user and not a good sole performance metric. It too can be arbitrarily inflated, is biased toward slower processors, and does not say much about end performance or performance bottlenecks. However, it may be useful as a starting point to decide whether to start looking more deeply for performance problems in a program or machine. Similar arguments hold for the utilization of other resources; utilization is useful for determining whether a machine design is balanced among resources and where the bottlenecks are, but it is not useful for measuring and comparing performance.

Problem Size

Another interesting metric is the smallest problem size of a given application that obtains a specified *parallel efficiency*, which is defined as speedup divided by number of processors (under a given scaling model). Since overheads due to parallelism gen-

erally decrease with problem size, the benefit of an improved communication architecture can often be seen in the ability to run smaller problems well. Keeping parallel efficiency fixed as the number of processors increases, in a sense, introduces a new scaling model that we might call *efficiency-constrained scaling*. Of course, this metric must be used with care since capacity effects may dominate communication differences and small problems may fail to stress important aspects of the system. Parallel efficiency is useful but is not a general performance metric.

Percentage Improvement in Performance

A metric that is sometimes used to evaluate the improvement in performance due to an architectural feature is the percentage improvement in execution time or speedup delivered by the feature. Without mention of the original parallel performance (e.g., the original speedup), this metric can be misleading in parallel systems. For example, improving the speedup from 400 to 800 on a 1,024-processor system is the same percentage improvement as improving speedup from 1.1 to 2.2, but the latter is unlikely to be interesting for a 1,024-processor system. If problem size is the reason for poor speedup, then it is often the case that increasing the problem size to yield decent speedup often dramatically reduces the improvement achieved by the feature. Here too, the metric has value but must be supplemented with other metrics to avoid misleading.

In summary, both cost and performance are important to consider. From a user's viewpoint in comparing machines, the performance metric of greatest interest is wall-clock execution time. However, from the viewpoint of an architect, or of a programmer trying to understand a program's performance, or even of a user interested more generally in a machine's performance aspects, it is best to look at both execution time and speedup. Both of these metrics should be presented in the results of any study. Ideally, execution time should be broken up into its major components as discussed in Section 3.4. To understand performance bottlenecks, it is very useful to see these component breakdowns on a per-process basis or as an average and some measure of dispersion over processes (simply an average is not enough). In evaluating the impact of changes to the communication architecture, or in comparing parallel machines based on equivalent underlying nodes, size- or configuration-based metrics like the minimum problem size needed to achieve a certain goal can be useful. Metrics like MFLOPS, MIPS, and processor utilization can be used for specialized purposes, but using only them to represent performance requires a lot of assumptions about the knowledge and integrity of the presenter, and they are burdened with a legacy of misuse.

4.3 EVALUATING AN ARCHITECTURAL IDEA OR TRADE-OFF

Imagine that you are an architect at a computer company, getting ready to design the next-generation multiprocessor. You have a new architectural idea that you would like to decide whether or not to include in the machine. You may have a wealth of

information about performance and bottlenecks from the previous-generation machine, which in fact may have been what prompted you to pursue this idea in the first place. However, your idea and this data are not all that is new. Technology has changed since the last machine, ranging from the level of integration to the cache sizes and organizations used by microprocessors. The processor you will use may be not only a lot faster but also a lot more sophisticated (e.g., four-way issue and dynamically scheduled versus single issue and statically scheduled), and it may have new capabilities that affect the idea at hand. The operating system has likely changed too, as has the compiler and perhaps even the workloads of interest, and these software components may change further by the time the machine is actually built and sold. The feature you are interested in has a significant cost in hardware and particularly in design time. And you have deadlines to contend with.

In this sea of change, the relevance of the data that you have for making decisions about performance and cost is questionable. At best, you could use it together with your intuition to make informed and educated guesses. But if the cost of the feature is high, you probably want to do more. What you can do is to build a simulator that models your system. You fix everything else—the compiler, the operating system, the processor, the technological and architectural parameters—to their expected configurations and simulate the system with the feature of interest absent and then present to judge its performance impact. Then perhaps you examine the sensitivity to some of the aspects that you had held fixed, but that may not be so predictable.

Building accurate simulators for parallel systems is difficult. Many complex interactions in the extended memory hierarchy are difficult to model correctly, particularly those having to do with resource occupancy and contention. Processors themselves are becoming much more complex, and accurate simulation demands that they, too, be modeled in detail. However, even if you can design a very accurate simulator that mimics your design, you still have a big problem. Simulation is expensive; it takes a lot of memory and time, especially when larger problems and machines are simulated. The implication is that you cannot simulate life-sized problem and machine sizes, and you will have to scale down your simulations somehow.

Even your technological parameters may not be fixed. You are starting with a clean slate and want to know how well the idea would work with different technological assumptions. Now, in addition to the earlier axes of workload, problem size, and machine size, the parameters of the machine are also variable. These parameters include the sizes and organizations of the levels in the local memory hierarchy; the granularities of allocation, communication, and coherence; and the performance characteristics of the communication architecture, such as latency, occupancy, and bandwidth. These parameters, together with those of the workload, lead to a vast parameter space that you must navigate. The high cost and the limitations of simulation make it all the more important that you prune this design space while not losing too much coverage. This section discusses the methodological considerations in choosing parameters and pruning the design space for simulation studies, using a particular evaluation as an example. First, let us take a quick look at multiprocessor simulation.

4.3.1

Multiprocessor Simulation

Although multiple processes and processing nodes are being simulated, the simulation itself may be run on a single processor. A *reference generator* plays the role of the processors on the parallel machine. It simulates the activities of the processors and issues memory references (together with a process identifier that tells which processor the reference is from) or commands (such as send or receive) to a simulator of the memory system and interconnection network (see Figure 4.7). If the simulation is being run on a uniprocessor, the different simulated processes time-share the uniprocessor, scheduled by the reference generator. One example of scheduling would be to deschedule a process every time it issues a reference to the memory system simulator and allow another process to run until that process issues its next reference; another example would be to reschedule processes every simulated clock cycle. The memory system simulator simulates all the caches and main memories on the different processing nodes, as well as the interconnection network itself. It can be arbitrarily complex in its simulation of datapaths, latencies, and contention.

The coupling between the reference generator (processor simulator) and the memory system simulator can be organized in various ways, depending on the accuracy needed in the simulation and the complexity of the processor model. One option is *trace-driven simulation*. In this case, a trace of the instructions executed by each process is first obtained by running the parallel program on one system, perhaps a different system than the one being evaluated. This trace takes the place of the reference generator: instructions from the trace are fed into the simulator that simulates the extended memory hierarchy of the target multiprocessor. Here, the coupling or flow of information is only in one direction: from the reference generator (here just a trace) to the memory system simulator.

The more popular form of simulation is *execution-driven simulation*, which provides coupling in both directions. In execution-driven simulation, when the memory system simulator receives a reference or command from the reference generator (which is now a program rather than a predetermined trace), it simulates the path of the reference through the extended memory hierarchy—including contention with other references—and returns to the reference generator the time that the reference took to be satisfied. This information, together with concerns about fairness and preserving the semantics of synchronization events, is used by the reference generator program to determine which simulated process to schedule next and when to issue the next instruction from that process. Thus, feedback takes place from the memory system simulator to the reference generator, influencing the activity of the latter as in a real machine and providing more accuracy than trace-driven simulation. To allow for maximum concurrency in simulating events and references, most components of the memory system and network are also modeled as separate communicating threads scheduled by the simulator. A global notion of *simulated time*—that is, the virtual time that would have been seen by the simulated machine, not real time that the simulator itself runs for—is maintained by the simulator. It is this time that we look up in determining the performance of workloads on the simulated

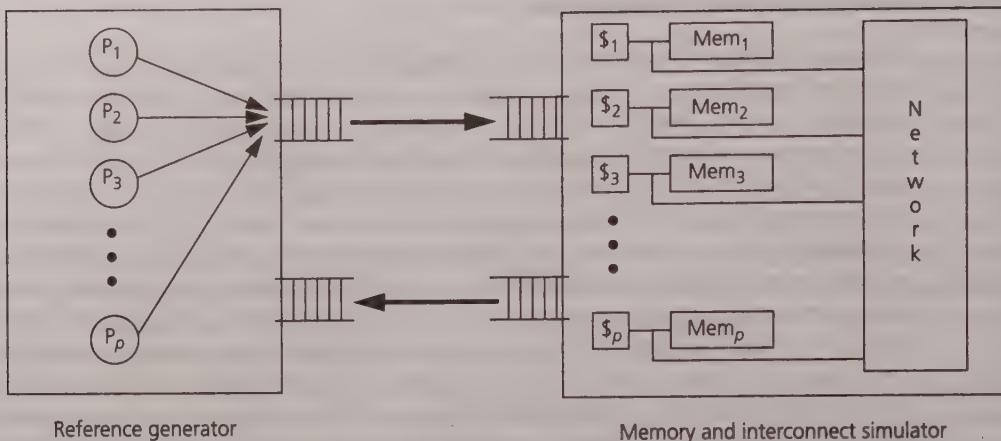


FIGURE 4.7 Execution-driven multiprocessor simulation. Simulated processors issue references to the memory system simulator, which simulates the extended memory hierarchy and feeds back timing information to the simulated processors (reference generators). $\$_1, \$_2$, etc. represent caches.

architecture and that is used to make scheduling decisions. In addition to time, simulators usually keep extensive statistics about the occurrence of various events of interest. This provides a wealth of detailed performance information that would be difficult, if not impossible, to obtain on a real system. However, the results may be tainted by a lack of credibility since it is, after all, a simulation. Accurate execution-driven simulation is also much more difficult when complex, dynamically scheduled, multiple-issue processors have to be modeled. Some of the trade-offs in simulation techniques are discussed in Exercise 4.9.

4.3.2 Scaling Down Problem and Machine Parameters for Simulation

Given that the simulation is done in software and involves many processes or threads that are very frequently being rescheduled (more often for more accuracy), it is not surprising that simulation is very expensive. Research is being done in simulation itself to speed it up and in using hardware emulation instead of simulation (Reinhardt et al. 1993; Goldschmidt 1993; Barroso et al. 1995), but progress has not been significant enough to change the fact that parameters must be scaled down substantially.

The tricky part about scaling down problem and machine parameters is that we want the scaled-down machine running the smaller problem to be representative of the full-scale machine running the larger problem. Unfortunately, there are no good formulas for this. Nonetheless, it is an important issue since it is the reality of most architectural trade-off evaluation. We should at least understand the limitations of such scaling, recognize which parameters can be scaled down with confidence and which cannot, and develop guidelines that help us avoid major pitfalls. Let us first

examine scaling down the problem size and number of processors and then explain some further difficulties associated with lower-level machine parameters. Again, for concreteness the focus is on a cache-coherent shared address space communication abstraction.

Problem Parameters and Number of Processors

Consider problem parameters first. We should first look for those problem parameters, if any, that affect simulation time greatly but have little impact on execution characteristics related to parallel performance. An example is the number of time-steps executed in many scientific computations, like Ocean or even Barnes-Hut, or the number of iterations in the simple equation solver. The data values manipulated can change a lot across time-steps, but the behavioral characteristics don't change very much. In such cases, we can run the simulation for only a few time-steps.²

Unfortunately, many application parameters affect execution characteristics related to parallelism. When scaling these parameters, we must also scale down the number of processors since otherwise we may obtain highly unrepresentative behavioral characteristics. However, this is difficult to do in a representative way because we are faced with many constraints that are individually difficult to satisfy and that might be impossible to reconcile with one another. These include the following:

- *Preserving the distribution of time spent in program phases.* The relative amounts of time spent performing different types of computation—for example, in the tree-building and force calculation phases of Barnes-Hut—will most likely change with problem and machine size.
- *Preserving key behavioral characteristics.* These include the communication-to-computation ratio, load balance, and temporal and spatial locality, which may all scale in different ways!
- *Preserving scaling relationships among application parameters.*
- *Preserving contention and communication patterns.* This is particularly difficult, since burstiness, for example, is difficult to predict or control.

Rather than preserving true representativeness when scaling down, more realistic goals are to at least cover a range of realistic operating points with regard to the behavioral characteristics that matter most for a study and to avoid unrealistic scenarios. Thus, scaled-down simulations are not even claimed to be quantitatively representative but can be used to gain insight and rough estimates. With this more

2. Of course, we should now omit the initialization and cold-start periods of the application from the measurements since their impact is much larger in the run with reduced time-steps than it would be in practice. If we expect that the behavior over long periods of time may in fact change substantially, as is possible in Barnes-Hut or applications whose characteristics change more dynamically, then we can dump out the program state periodically from an execution on a real machine of the problem configuration we are simulating, and start a few sample simulations with these dumped-out states as their input data sets (again not measuring cold start in each sample). Other sampling techniques can also be used to reduce simulation cost.

modest goal, let us assume that we have scaled down the application parameters and the number of processors in some reasonable way and see how to scale other machine parameters.

Other Machine Parameters

Scaled-down problem and machine sizes interact differently with low-level machine parameters than the full-scale problems would. We may therefore have to scale these parameters carefully as well.

Consider the size of the cache or replication store. Suppose that the largest problem and machine configuration that we can simulate for the equation solver kernel is a 512×512 grid with 16 processors (i.e., 128 KB per processor). If we don't scale down the 1-MB cache per processor, we will never be able to represent the situation where the important working set doesn't fit in the cache. The key point with regard to scaling caches is that it should be done based on an understanding of how the relevant working sets scale as per our discussion of realistic and unrealistic operating points (Figure 4.4). Not scaling the cache size at all, or simply scaling it down proportionally with data set size or problem size, is inappropriate in general since cache size interacts most closely with working set size, not with data set or problem size. Example 4.6 and Figure 4.8 illustrate how to choose cache sizes given a problem and machine size. We should also ensure that the caches we simulate don't become extremely small since these can suffer from unrepresentative mapping and fragmentation artifacts. Similar arguments apply to replication stores other than processor caches, including those that hold only communicated data.

EXAMPLE 4.6 In the Barnes-Hut application, suppose that the size of the most important working set when running a full-scale problem with $n = 1\text{-M}$ particles is 150 KB, that the target machine has 1-MB caches per processor, and that you can only simulate an execution with $n = 16\text{-K}$ particles. Would it be appropriate to scale the cache size down proportionally with the data set size? How would you choose the cache size?

Answer Recall from Chapter 3 that the size of the most important working set in Barnes-Hut scales as $\log n$, where n is the number of particles and is proportional to the size of the data set. The working set of 150 KB fits comfortably in the full-scale 1-MB cache on the target machine. Given its slow growth rate, this working set is likely to always fit in the cache for realistic problems. If we scale the cache size proportionally to the data set for our simulations, we get a cache size of $1\text{ MB} \times 16\text{ K}/1\text{ M}$, or 16 KB. The size of the working set for the scaled-down problem is

$$150\text{ KB} \times \frac{\log 16\text{ K}}{\log 1\text{ M}}$$

or 70 KB, which clearly does not fit in the scaled-down 16-KB cache. Thus, this form of cache scaling has brought us to an operating point that is not representative of reality. Since we expect the working set to fit in the cache in reality, we should rather choose a cache size large enough to always hold this working set. ■

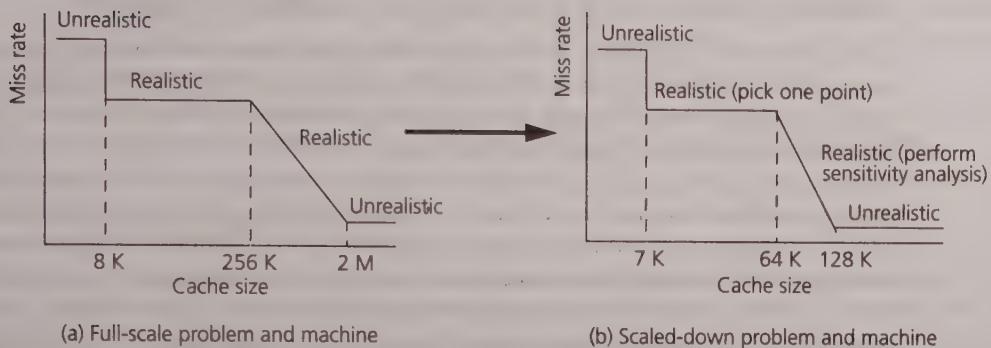


FIGURE 4.8 Choosing cache sizes for scaled-down problems and machines. (a) Based on our understanding of the sizes and scaling of working sets, we first decide what regions of the working set curve are realistic for full-scale problems running on the machine with full-scale caches. (b) We then project or measure what the working set curve looks like for the smaller problem and machine size that we are simulating, prune out the corresponding unrealistic regions in it, and pick representative operating points (cache sizes) for the realistic regions in a manner similar (but complementary) to that discussed in Example 4.4. For regions that cannot be pruned, we can perform sensitivity analysis as necessary.

As we move to still lower-level parameters of the extended memory hierarchy, scaling them representatively becomes increasingly difficult. For example, interactions with cache associativity are very difficult to predict, and usually the best we can do is leave the associativity as it is. The main danger is with retaining a direct-mapped cache when cache sizes are scaled down very low since this situation is particularly susceptible to mapping conflicts that wouldn't occur in the full-scale cache. Interactions with other organizational parameters of the memory and communication architectures—such as the granularities of data allocation, transfer, and coherence—are also complex and unpredictable unless there is near perfect spatial locality, but keeping them fixed can lead to serious, unrepresentative artifacts in many cases. We shall see some examples in the exercises. Finally, performance parameters like latency, occupancy, and bandwidth are also very difficult to scale down appropriately to preserve representativeness as the frequencies and patterns of communication change.

In summary, the best approach to simulation is to try to run realistic (if small) problem sizes to the extent possible. When scaling down is necessary, we should heed the guidelines and pitfalls we have discussed to ensure that the important types of operating points are covered, and we should extrapolate with caution. Our confidence in scaling down relies on our understanding of the application. In general, using scaled-down scenarios is okay for understanding whether certain architectural features are likely to be beneficial or not, but it is dangerous to use them to try to draw precise quantitative conclusions about full-scale situations.

4.3.3 Dealing with the Parameter Space: An Example Evaluation

Consider now the problem of the large parameter space opened up by trying to evaluate an idea in a general context. To keep the discussion concrete, let us examine an actual evaluation that we might perform using simulation. Assume again a cache-coherent shared address space machine with physically distributed memory. The default mechanism for communication is implicit communication in cache blocks through loads and stores, but we want to explore reducing the impact of endpoint communication overhead and communication delay by communicating in larger messages. We therefore wish to understand the utility of adding to such an architecture a facility to explicitly send larger messages, called a *block transfer* facility, which programs can use in addition to the standard transfer mechanisms for cache blocks (thus merging the shared address space and message-passing programming models). In the equation solver, for example, a process might send an entire border subrow or subcolumn of its partition to its neighbor process in a single block transfer.

In choosing workloads for such an evaluation, we should choose at least some with communication that is amenable to being structured in large messages, such as the equation solver. The more difficult problem is navigating the parameter space. Our goals are threefold:

1. *To avoid unrealistic execution characteristics.* We should avoid combinations of parameters (or operating points) that lead to unrealistic behavioral characteristics—that is, behavior that wouldn't be encountered in practical use of the machine.
2. *To obtain good coverage of realistic execution characteristics.* We should try to ensure that important characteristics that may arise in real usage are represented.
3. *To prune the parameter space.* Even in the realistic subspaces of parameter values, we should try to prune out points when possible based on application knowledge, in order to save time and resources without losing much coverage, and to determine when explicit sensitivity analysis is necessary.

We can prune the space based on the goals of the study, the restrictions on parameters imposed by technology (or the use of specific building blocks), and an understanding of parameter interactions.

Let us go through the process of choosing parameters, using the equation solver kernel as an example. Although we shall examine the parameters one by one, issues that arise in later stages may make us revisit decisions we made earlier. We begin with choosing the problem size and number of processors since these are limited the most by simulation resources.

Problem Size and Number of Processors

We choose the problem sizes and the numbers of processors based on the considerations of inherent program characteristics that we have discussed for evaluating a

real machine and for scaling down for simulation. For example, if the problem is large enough that the communication-to-computation ratio is very small, then block transfer is not going to help overall performance much; nor will it help if the problem is small enough that load imbalance is the dominant bottleneck.

Let us now fix the problem size for the equation solver at a 514×514 grid, the number of processors at 16, and examine how to choose other parameters.

Cache/Replication Size

As usual, we choose cache sizes based on knowledge of the working set curve. Given a working set curve and knowledge of how working sets scale, the process of choosing cache sizes for a given problem size is analogous to that of choosing problem sizes for a given cache size and is illustrated in Example 4.7.

EXAMPLE 4.7 Figure 4.9 shows the well-defined working sets of the equation solver. How might you choose the cache sizes in this case?

Answer Although the sizes of important working sets often depend on application parameters and the number of processors, their nature and hence the general shape of the working set curve usually do not change with these parameters. Since we know the size of each important working set in the equation solver and how it scales with these parameters, if we know the range of cache sizes that are realistic for target machines, then we can tell whether it is (1) unrealistic to expect that working set to fit in the cache in practical situations, (2) unrealistic to expect that working set not to fit in the cache, or (3) realistic to expect it to fit for some practical combinations of parameter values and not to fit for others.³ Thus, we can tell which of the regions between knees in the curve may be representative of realistic situations and which are not. For a given problem size and number of processors, we can use the (fixed) working set curve to choose cache sizes that avoid unrepresentative regions, cover representative ones, and prune flat regions by choosing only a single cache size from them (if all we care about from a cache is its miss rate). ■

Whether or not an important working set fits in the cache affects the benefits from block transfer greatly and in interesting ways. The effect depends on whether the working set consists of locally or nonlocally allocated data. If it consists mainly of local data—as in the equation solver when data is placed properly—but it doesn't fit in the cache, the processor spends more of its time stalled on the local memory system. As a result, communication time becomes relatively less important, and block transfer is likely to help less (block-transferred data also interferes more with the local traffic in a node, causing contention). However, if the working sets are mostly nonlocal data, we have the opposite effect: if they don't fit in the cache, then there is more communication and hence a greater opportunity for block transfer to help performance.

3. Whether a working set of a given size fits in the cache may depend on cache associativity and perhaps even block size in addition to cache size, but it is usually not a major issue in practice if we assume at least two-way associativity (as we shall see later). Thus, we can ignore these effects for now.

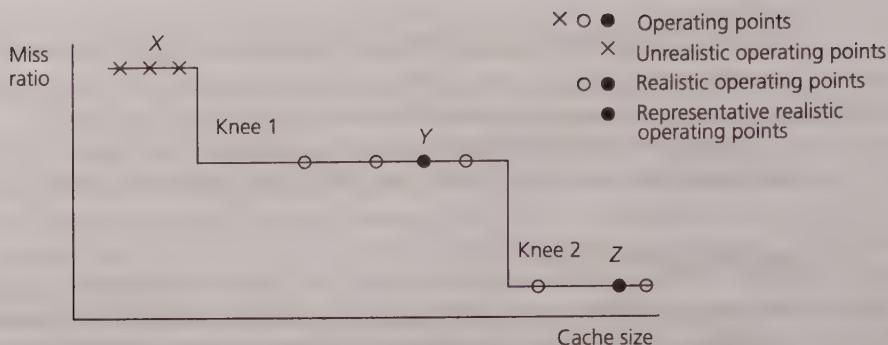


FIGURE 4.9 Picking cache sizes for an evaluation using the equation solver kernel. Knee 1 corresponds roughly to a couple of subrows of either B or n/\sqrt{p} elements, depending on whether the grid traversal is blocked (with block size $B \times B$) or not. Knee 2 corresponds to a processor's partition of the matrix (i.e., data set n^2 divided by p). The latter working set may or may not fit in the cache depending on n and p , so both Y and Z are realistic operating points and should be represented. For the first working set, it is conceivable that it will not fit in the caches if the traversal is not blocked, but as we have seen in realistically large second-level caches, this is very unlikely. If the traversal is blocked, the block size B is chosen so the former working set always fits. Operating point X is therefore representative of an unrealistic region and is ignored. Blocked matrix computations are similar in this respect.

Of course, a working set curve is not always composed of relatively flat regions separated by sharply defined knees. If the curve has knees but the regions they separate are not flat (see Figure 4.10[a]), we can still prune out entire regions as before if we know they are unrealistic. However, if a region is realistic but not flat, or if there aren't any knees until the entire data set fits in the cache (as in Figure 4.10[b]), then we must resort to sensitivity analysis, picking points close to the extremes as well as perhaps some in between. Again, proper evaluation requires that we understand the key characteristics of the applications well.

The remaining question is how to determine the sizes of the knees and the shape of the working set curve between them. In simple cases, we may be able to do this analytically. However, algorithms are complex, constant factors are difficult to predict, and the effects of cache block size and associativity may be difficult to analyze. In these cases, we can obtain the curve for a given problem size and number of processors by measurement (simulation) with different cache sizes. The simulations needed are relatively inexpensive since working set sizes do not depend on detailed timing-related issues, such as latencies, bandwidths, occupancies, and contention, which therefore do not need to be simulated carefully (or at all). How the working sets change with problem size or number of processors can then be analyzed or measured again as appropriate. Fortunately, analysis of growth rates is usually easier than predicting constant factors. Also, lower-level issues like block size and associativity often don't change working sets too much for large enough caches and reasonable cache organizations (other than direct-mapped caches) (Woo et al. 1995).

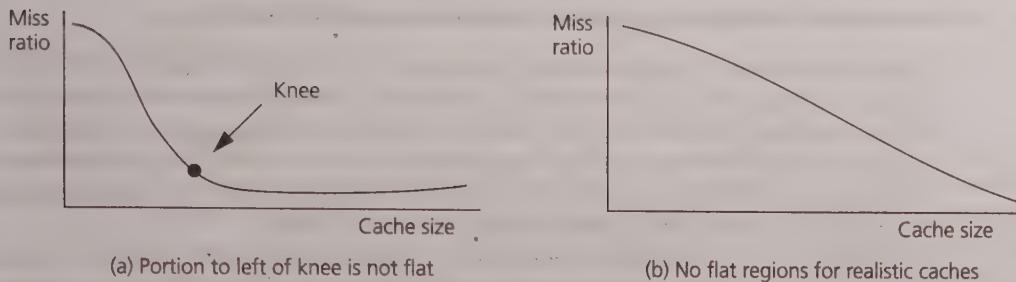


FIGURE 4.10 Miss rate versus cache size curves that do not consist of sharp knees separated by flat regions

Cache Block Size and Associativity

In addition to the problem size, the number of processors, and the cache size, the cache block size is another important parameter for determining the benefits of block transfer. The issues are a little more complicated, however. Long cache blocks themselves act like small block transfers for programs with good spatial locality, making explicit block transfer relatively less effective in these cases. On the other hand, if spatial locality is poor, then the extra traffic caused by long cache blocks (due to fragmentation or false sharing) can consume a lot more bandwidth than necessary when communicating through reads and writes. Whether poor spatial locality wastes bandwidth for block transfer as well depends on whether block transfer is implemented by pipelining whole cache blocks through the network or just the necessary words. Note that block transfer itself increases bandwidth requirements since it causes the same amount of communication to be performed (hopefully) in less time. Thus, if block transfer is implemented using pipelined cache block transfers and if spatial locality is poor, using block transfer may hurt rather than help when available bandwidth is limited since it may increase contention for the available bandwidth.

Fortunately, we are usually able to restrict the range of interesting cache block sizes either because of constraints of current technology or because of limits imposed by the set of available building blocks. For example, almost all microprocessors today support cache blocks between 32 and 128 bytes, and we may have already chosen a microprocessor that has a 64-byte cache block. When thresholds occur in the interactions of problem size and cache block size (for instance, in the radix sorting example discussed earlier), we should ensure that we cover both sides of the threshold.

While the magnitude of the impact of cache associativity is very difficult to predict, real caches are built with small associativity (usually at most four-way), so the number of choices to consider is small. If we must choose a single associativity, we are best advised to avoid direct-mapped caches (at least at the lowest level of the hierarchy, furthest from the processor) unless we know that the machines of interest will have them.

Performance Parameters of the Communication Architecture

Having discussed the organizational parameters of the extended memory hierarchy, let us consider the key performance parameters of the communication architecture—overhead, network delay or transit time, and bandwidth—and how they affect the benefits of block transfer. We should choose the base values of these parameters according to what we expect for real systems of interest, but this understanding helps us decide which parameters should be varied and how.

The higher the overhead component of a communicating cache block (on a miss, say), the more important it is to amortize it by structuring communication in larger block transfers. This is true as long as the overhead of initiating a block transfer is not so high as to drown out the benefits, since the overhead of explicitly initiating a block transfer may be larger than that of implicitly initiating the transfer of a cache block.

By the same token, the higher the network transit time between nodes, the greater the benefit of amortizing it over large block transfers (there are limits to this, which will be discussed when we examine block transfer in detail in Chapter 11). The effects of changing latency usually do not exhibit knees or thresholds, so in order to examine a range of possible latencies, we simply have to perform sensitivity analysis by choosing a few points along the range. In practice, we would usually choose latencies based on the target latencies of the machines of interest; for example, tightly coupled multiprocessors typically have much smaller latencies than workstations on a local area network.

Available bandwidth is also an important issue for our block transfer study. Bandwidth exhibits a strong knee effect as well, which is in fact a saturation effect: either enough bandwidth is available for the needs of the application, or it is not. If it is, then it may not matter too much whether the available bandwidth is four times what is needed or ten times. We can therefore pick one bandwidth that is less than that needed and one that is much more. Since the block transfer study is particularly sensitive to bandwidth, we may also choose one that is closer to the borderline. In choosing bandwidth values, we should be careful to consider the burstiness in the bandwidth demands of the application; the average bandwidth needs over the whole application may be small, but the application may still saturate a higher bandwidth during its periods of bursty communication, leading to contention.

Revisiting Choices

Finally, we may often need to revise our earlier choices for parameter values based on interactions with parameters considered later. For example, if we are forced to use small problem sizes due to lack of simulation time or resources, then we may be tempted to choose a very small cache size to represent a realistic situation where an important working set does not fit in the cache. However, choosing a very small cache may lead to severe artifacts, especially if we use a direct-mapped cache or a large cache block size (since this will lead to very few blocks in the cache and potentially a lot of fragmentation and mapping conflicts). We should therefore reconsider

our choice of problem size and number of processors for which we want to represent this situation.

4.3.4 Summary

The preceding discussion shows that the results of an evaluation study can be misleading if we don't cover the space adequately: we can easily choose a combination of parameters and workloads that demonstrates good performance benefits from a feature such as block transfer (for example, a relatively small problem size, big caches, and a small cache block size), and we can just as easily choose a combination that doesn't. It is therefore very important that we incorporate sound methodological guidelines in our architectural studies and understand the relevant interactions between hardware and software.

In spite of a significant number of relevant interactions, we can fortunately identify certain parameters and properties that are at a high enough level for us to reason about, and that do not depend on lower-level timing details of the machine, upon which key behavioral characteristics of applications depend crucially. We should ensure that we cover realistic regimes of operation with regard to these parameters and properties—namely, application parameters, the number of processors, and the relationship between working sets and cache/replication size (that is, whether or not the important working sets fit in the caches). Benchmark suites should provide the basic characteristics such as concurrency, communication-to-computation ratio, and data locality for their applications, together with their dependence on these parameters, so that architects do not have to reproduce them (Woo et al. 1995).

It is also important to look for knees and flat regions in the interactions of application characteristics and architectural parameters, since these are especially useful for both coverage and pruning. Finally, the high-level goals and constraints of a study can also help us prune the parameter space.

This concludes our discussion of methodological issues in workload-driven evaluation. The remainder of this chapter introduces the rest of the parallel workloads that we shall use most often in the book. It also describes the basic methodologically relevant characteristics of all our workloads.

4.4 ILLUSTRATING WORKLOAD CHARACTERIZATION

Workloads are used extensively in this book both to quantitatively illustrate some of the architectural trade-offs we discuss and to evaluate our case study machines. Systems designed primarily to support a coherent shared address space communication abstraction are discussed in Chapters 5, 6, and 8, while message-passing and noncoherent shared address space systems are discussed in Chapter 7. Our programs for the abstractions are written in the corresponding programming models. Since programs in the two models are written very differently (as described in Chapter 2) and some of the important characteristics are different, we illustrate our characterization

of workloads with the programs written for a coherent shared address space. In particular, we use six parallel applications and computational kernels that run in batch mode (i.e., one at a time) and do not include operating system activity and a multiprogrammed workload that does include operating system activity. While the number of workloads we use is small, the applications represent important classes of computation and have widely varying characteristics.

4.4.1 Workload Case Studies

All of the parallel programs we use for shared address space architectures are taken from the SPLASH-2 application suite (see Appendix). Three (Ocean, Barnes-Hut, and Raytrace) have already been described and used as case studies in previous chapters. This section briefly describes the workloads we use but haven't yet discussed: LU, Radix, Radiosity, and Multiprog. LU and Radix are computational kernels, Radiosity is a real application, and Multiprog is a multiprogrammed workload. In Section 4.4.2, we measure some methodologically relevant execution characteristics of the workloads, including the breakdown of data accesses, the communication-to-computation ratio and how it scales, and the size and scaling of the important working sets. We use this characterization to choose memory system parameters for these applications and data sets in later chapters.

LU

Dense LU factorization is the process of converting a dense matrix A into two matrices L , U that are lower and upper triangular, respectively, and whose product equals A (i.e., $A = LU$).⁴ Its utility is in solving linear systems of equations, and it is encountered in scientific applications as well as optimization methods such as linear programming. It is a well-structured computational kernel that is nontrivial yet familiar and fairly easy to understand (Golub and Van Loan 1997).

LU factorization works like Gaussian elimination, eliminating one variable at a time by subtracting rows of the matrix from scalar multiples of other rows. The computational complexity of LU factorization is $O(n^3)$, while the size of the data set is $O(n^2)$. As we know from the discussion of temporal locality in Chapter 3, this is an ideal situation to exploit temporal locality by blocking. In fact, we use a blocked LU factorization, which is far more efficient both sequentially and in parallel than an unblocked version. The n -by- n matrix to be factored is divided into B -by- B blocks, and the idea is to reuse a block as much as possible before moving on to the next block. We now think of the matrix as consisting of n/B -by- n/B blocks rather than n -by- n elements and eliminate and update blocks one at a time just as we

4. A matrix is called *dense* if a substantial proportion of its elements are nonzero (matrices that have mostly zero entries are called *sparse* matrices). A *lower-triangular matrix* such as L is one whose entries are all zero above the main diagonal, whereas an *upper-triangular matrix* such as U has all zeros below the main diagonal. (The *main diagonal* is the diagonal that runs from the top left corner of the matrix to the bottom right corner.)

```

for k ← 0 to N-1 do           /*loop over all diagonal blocks*/
    factorize block  $A_{k,k}$ ;
    for j ← k+1 to N-1 do
         $A_{k,j} \leftarrow A_{k,j} * (A_{k,k})^{-1}$ ; /*for all blocks in the row of, and
        for i ← k+1 to N-1 do      to the right of, this diagonal block*/
            for j ← k+1 to N-1 do /*divide by diagonal block*/
                 $A_{i,j} \leftarrow A_{i,j} - A_{i,k} * (A_{k,j})^T$ ; /*for all rows below this diagonal block*/
            endfor
        endfor
    endfor
endfor

```

endfor

endfor

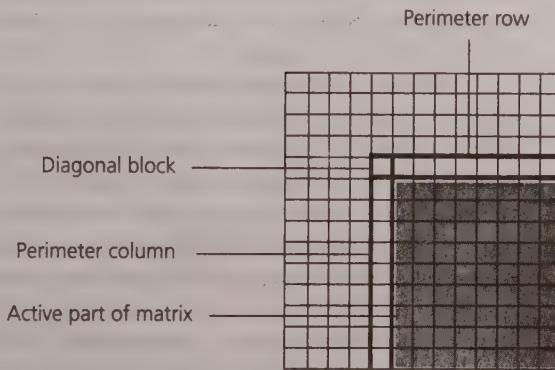


FIGURE 4.11 Pseudocode describing sequential blocked dense LU factorization. N is the number of blocks in each dimension ($N = n/B$), and we think of the matrix as an N -by- N matrix of blocks rather than an n -by- n matrix of elements. Then, $A_{i,j}$ represents the block in the i th row and j th column of matrix A . In the k th iteration of this outermost loop, we call the block $A_{k,k}$ on the main diagonal of A the diagonal block, and the k th row and column of blocks the perimeter row and perimeter column, respectively. Note that the k th iteration does not touch any of the blocks in the first $k-1$ rows or columns of the matrix; that is, only the shaded part of the matrix in the square region to the right of and below the diagonal block is “active” in the current outermost loop iteration. The rest of the matrix has already been computed in previous iterations and will be inactive for the rest of the factorization. In an unblocked LU factorization, we would refer similarly to a diagonal element and a perimeter row and column of elements.

would elements. Matrix operations like multiplication and inversion are used on small B -by- B blocks rather than scalar operations on elements. Sequential pseudocode for this blocked LU factorization is shown in Figure 4.11, which also defines some relevant terms.

Consider the benefits of blocking. If we did not block the computation, a processor would compute an element, then compute its next assigned element to the right, and so forth until the end of the current row, after which it would proceed to the next row. When it returned to the first active element of the next row, it would re-reference a perimeter row element (the one it used to compute the corresponding active element of the previous row). However, by this time it has streamed through

data proportional to an entire row of the matrix and, given large matrices, that perimeter row element might no longer be in the cache. In the blocked version, within the block-level computations in each iteration of the innermost loop in Figure 4.11 (i.e., the computation in the line $A_{i,j} \leftarrow A_{i,j} - A_{i,k} * (A_{k,j})^T$), we proceed only B elements in a direction before returning to previously referenced data that are still in the cache and can be reused. The operations (matrix multiplications and factorizations) on the B -by- B blocks each involve $O(B^3)$ computation and data accesses, with each block element being accessed B times. If the block size B is chosen such that a block of B -by- B or B^2 elements (plus some other data) fits in the cache, then in a given block computation only the first access to an element misses in the cache. Subsequent accesses hit in the cache, resulting in B^2 misses for B^3 accesses or a miss rate of $1/B$.

In the parallel version, we can think of every computation that updates a block as a task. Figure 4.12 provides a pictorial depiction of the flow of information among blocks within an outermost loop iteration and shows how we assign blocks (and hence tasks) to processors in the parallel version. Because of the nature of the computation, blocks toward the top left of the matrix are active only in the first few outermost loop iterations of the computation, whereas blocks toward the bottom right have a lot more work associated with them. Assigning contiguous rows or squares of blocks to processes (a simple domain decomposition of the matrix) would therefore lead to poor load balance. Consequently, we interleave blocks among processes in both dimensions, leading to a partitioning called a two-dimensional scatter decomposition of the matrix: the processes are viewed as forming a two-dimensional \sqrt{p} -by- \sqrt{p} grid, and this grid of processes is repeatedly stamped over the matrix of blocks like a cookie cutter. A process is responsible for computing the blocks that are assigned to it in this way, and only it writes those blocks. The interleaving alleviates—but does not eliminate load imbalance, whereas the blocking preserves locality and also allows us to use larger data transfers on message-passing systems.

The drawback of decomposing the computation into blocks rather than individual elements is that it increases task granularity and hurts load balance: concurrency is reduced since there are fewer blocks than elements, and the maximum load imbalance per iteration is the work associated with a block rather than a single element. In sequential LU factorization, the only constraint on block size is that the two or three blocks used in a block computation fit in the cache. In the parallel case, the ideal block size B is determined by a trade-off between data locality and communication overhead (particularly on a message-passing machine) pushing toward larger blocks on one hand and load balance pushing toward smaller blocks on the other. The ideal block size therefore depends on the problem size, number of processors, and other architectural parameters. In practice, block sizes of 16×16 or 32×32 elements appear to work well on large parallel machines.

Blocking provides data reuse within a block computation. Data can also be reused across different block computations. To reuse data from remote blocks, we can either copy blocks explicitly in main memory and keep them around, or on cache-coherent machines, we can perhaps rely on caches being large enough to do this automatically. However, reuse across block computations is typically not nearly as important

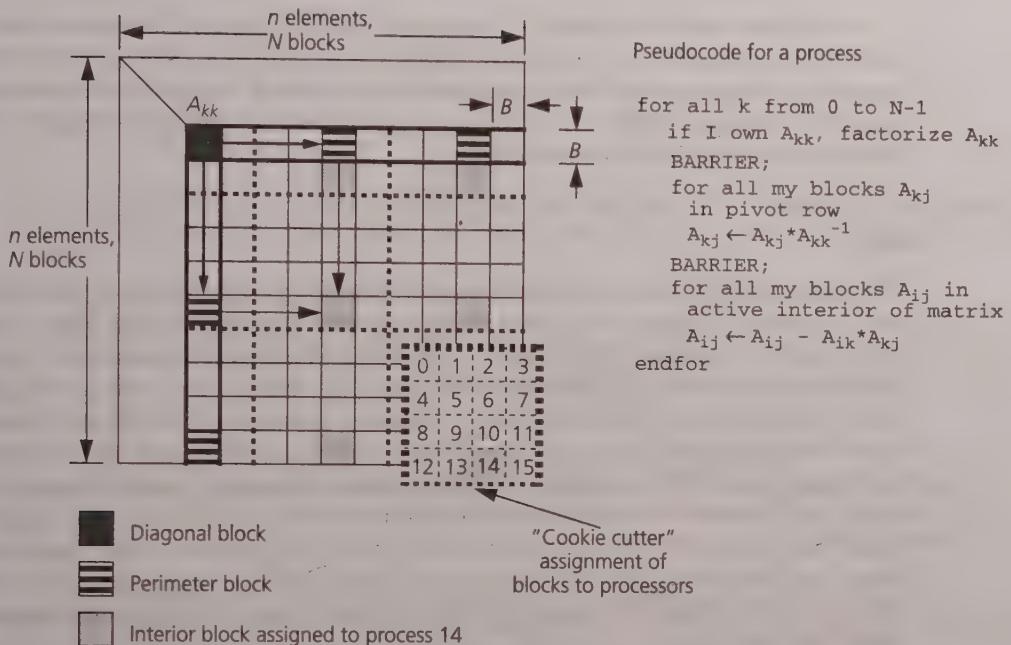


FIGURE 4.12 Parallel blocked LU factorization: flow of information, partitioning, and parallel pseudocode. The flow of information within an outer (k) loop iteration is shown by the solid arrows. Information (data) flows from the diagonal block (which is first factorized) to all blocks in the perimeter row in the first phase. In the second phase, a block in the active part of the matrix needs the corresponding elements from the perimeter row and perimeter column.

to performance as reuse within them (and explicit copying has a cost), so in our program we do not make explicit copies of blocks in main memory.

For spatial locality, since the unit of decomposition is now a two-dimensional block, the issues are quite similar to those discussed for the simple equation solver kernel in Section 3.3.1. We are therefore led to a four-dimensional array data structure to represent the matrix in a shared address space so that the data in a block is contiguous in the address space. The first two dimensions specify a block, and the next two specify an element within the block. This allows us to distribute blocks appropriately among memories at page granularity. (If blocks are smaller than a page, we can use one more array dimension to ensure that all the blocks assigned to a process are contiguous in the address space.) However, with blocking the capacity miss rate is small enough that data distribution in main memory is not a major problem in LU factorization. The more important reason to keep a block's data contiguous by using high-dimensional arrays is to reduce cache mapping conflicts across subrows of a block as well as across blocks, as we discuss in Section 5.6. Cache conflicts are very sensitive to the size of the array and number of processors, especially with direct-mapped first-level caches, and can easily negate most of the benefits of blocking.

No locks are used in parallel LU factorization. Barriers are used to separate outermost loop iterations as well as phases within an iteration (e.g., to ensure that the perimeter row is computed before the blocks in it are used). Point-to-point synchronization at the block level could have been used to exploit more concurrency, but barriers make programming much easier.

Radix

The Radix program sorts a series of integers, called *keys*, using the popular radix sorting method. Suppose there are n integers to be sorted, each of size b bits. The algorithm uses a radix of r bits, where r is chosen by the user. This means the b bits representing a key can be viewed as a set of $\lceil b/r \rceil$ groups of r bits each (see Figure 4.13). The algorithm proceeds in $\lceil b/r \rceil$ phases or iterations. Each phase, starting with the lowest-order group, sorts the keys according to their values in the corresponding group of r bits, called a *digit*.⁵ The keys are completely sorted at the end of these $\lceil b/r \rceil$ phases. Two one-dimensional arrays of size n integers are used in each phase: one, called the *input array*, stores the keys as they appear in the input to a phase, and the other, called the *output array*, stores the keys as they appear in the output from the phase. The input array for one phase is the output array for the next phase, and vice versa.

Consider the parallel computation within a phase, which sorts all n keys according to their values in a particular digit. The parallel algorithm partitions the n keys in each array among the p processes so that process 0 is assigned the first n/p keys, process 1 the next n/p keys, and so on. The portion of each array assigned to a process is allocated in the corresponding processor's local memory. The n/p keys in the input array for a phase that are assigned to a process are called its *local keys* for that phase. Within a phase, a process performs the following steps:

1. Make a pass over the local n/p keys to build a local (per-process) histogram of key values. The histogram has 2^r entries, where r is the number of bits in a digit. If a key encountered has the value i in the current phase, then the i th bin of the histogram is incremented.
2. When all processes have completed step 1 (determined by barrier synchronization in this program), accumulate the local histograms into a global histogram. This is done with a parallel prefix computation, as discussed in Exercise 4.14. The global histogram keeps track of both how many keys there are of each value for the current digit and also, for each of the p process-ID values j , how many keys of a given value are owned by processes whose ID is less than j .
3. Make another pass over the local n/p keys. For each key, use the global and local histograms to determine which (sorted) position in the output array this

5. The reason for starting with the lowest-order group of r bits rather than the highest-order one is that this leads to a “stable” sort; that is, keys with the same value appear in the output in the same order relative to one another as they appeared in the input.

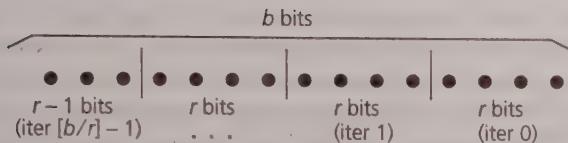


FIGURE 4.13 A b -bit number (key) divided into $\lceil b/r \rceil$ groups of r bits each. The first iteration of radix sorting uses the least significant r bits and so on.

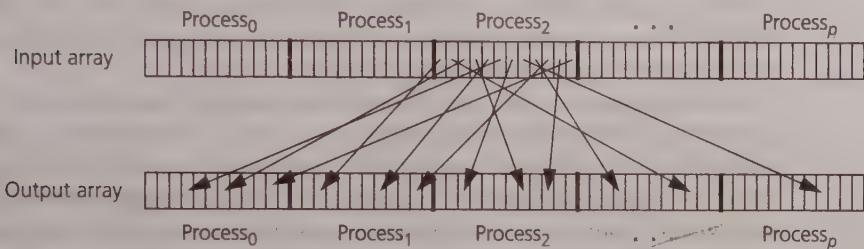


FIGURE 4.14 The permutation step of a radix sorting phase. In each of the input and output arrays (which change places in successive phases), keys (entries) assigned to a process are allocated in the corresponding processor's local memory.

key should go to, and write the key value into that entry of the output array. Note that the array element that will be written is very likely to be nonlocal, with expected likelihood $(p-1)/p$ (see Figure 4.14). This step is called the *permutation* step.

A more detailed description of radix sorting algorithms and implementations can be found in (Blelloch et al. 1991; Culler et al. 1993). In a shared address space implementation, communication occurs when writing the keys in the permutation phase (or reading them in the histogram-generation phase of the next iteration if they stay in the writers' caches) and in constructing the global histogram from the local histograms. The permutation-related communication is all-to-all personalized (i.e., every process communicates disjoint subsets of its keys to every other) but is irregular and scattered, with the exact patterns depending on the distribution of keys. The synchronization includes global barriers between phases as well as finer-grained synchronization in the phase that builds the global histogram. The latter may take the form of either mutual exclusion or point-to-point event synchronization, depending on the implementation of this phase (see Exercise 4.14).

Radiosity

The radiosity method is used in computer graphics to compute the global illumination in a scene that contains diffusely reflecting surfaces. In the hierarchical radiosity method, a scene is initially modeled as consisting of k large input polygons, or

patches. For example, the top of a table or the back of a chair may be an input patch. Light transport interactions are computed pairwise among these patches. In a simplified view of the algorithm, if the light transfer between a pair of patches is larger than a threshold, one of them (the larger one, say) is subdivided, and interactions are computed recursively between the resulting subpatches and the other patch. This process continues until the light transfer between all pairs is sufficiently low. Thus, patches are hierarchically subdivided as necessary to improve the accuracy of computing illumination. Each subdivision results in four subpatches, leading to a quadtree per patch. If the resulting final number of undivided subpatches is n , then with k original patches the complexity of this algorithm is $O(n + k^2)$. A brief description of the steps in the algorithm follows. Details can be found in (Hanrahan, Salzman, and Aupperle 1991; Singh 1993).

The input patches that comprise the scene are first inserted into a binary space partitioning (BSP) tree (Fuchs, Abram, and Grant 1983), which is a data structure that facilitates the efficient computation of visibility between pairs of patches. Every input patch is initially given an *interaction* list of other input patches that are potentially visible from it and with which it must therefore compute interactions. Then, radiosities are computed by the following iterative algorithm:

1. For every input patch, compute its radiosity due to all patches on its interaction list, subdividing it or other patches hierarchically and computing their interactions recursively as necessary (see Figure 4.15).
2. Starting from the patches at the leaves of the quadtrees, add all the patch radiosities together (weighted by their areas) to obtain the total radiosity of the scene, and compare it with that of the previous iteration to check for convergence within a fixed tolerance. If the radiosity has not converged, return to step 1. Otherwise, go to step 3.
3. Smooth the solution for display.

Most of the time in an iteration is spent in step 1, so let us examine it further. Suppose a patch i is traversing its interaction list to compute interactions with other patches (quadtree nodes). The interaction with another patch, say, j , involves computing the intervisibility of the two patches as well as the light transfer between them. (The actual light transfer is the product of the actual intervisibility and the light transfer that would have happened if there were no occlusion and hence full intervisibility.) Computing intervisibility involves traversing the BSP tree several times from one patch to the other;⁶ in fact, visibility computation is a very large portion of the overall execution time. If the result of an interaction says that the “source” patch i should be subdivided, then four children are created for patch i if

6. Visibility is computed by conceptually shooting a number of rays between the two patches and seeing how many of the rays reach the destination patch without being occluded by intervening patches in the scene. For each such conceptual ray, determining whether it is occluded or not is done efficiently by traversing the BSP tree from the source to the destination patch.

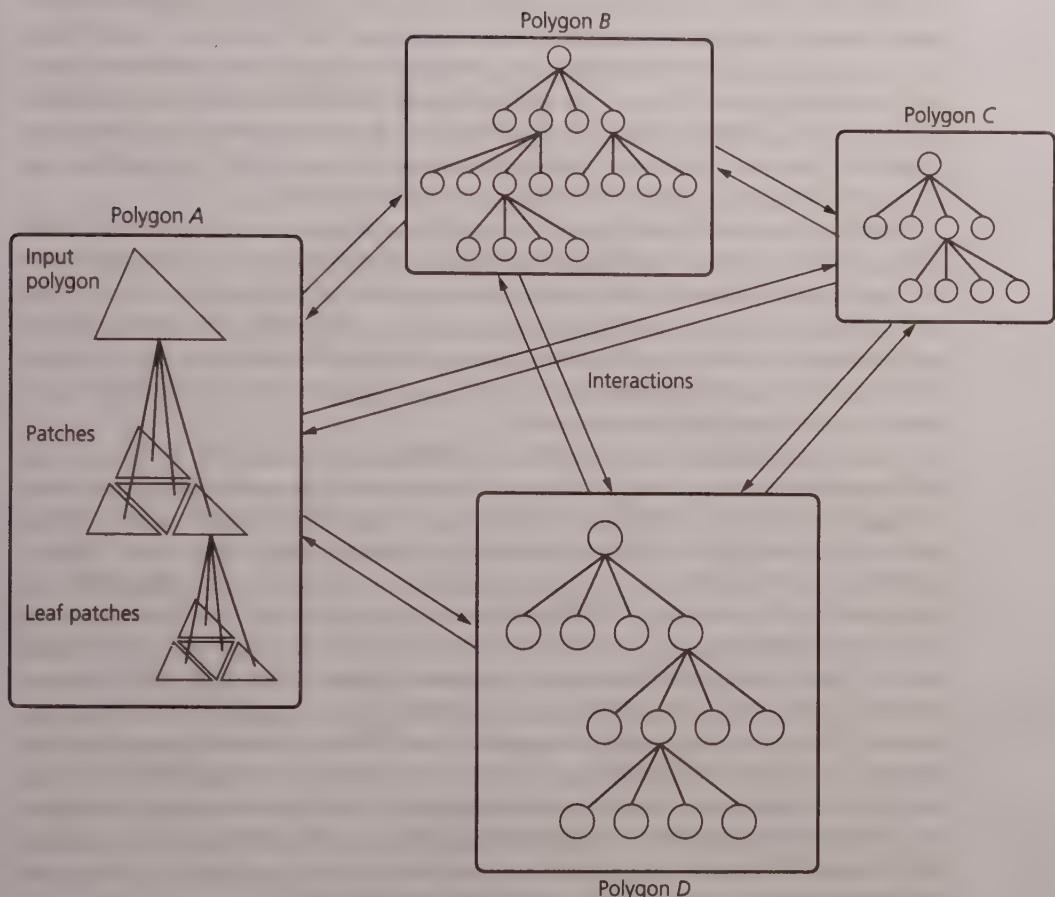


FIGURE 4.15 Hierarchical subdivision of input polygons into quadtrees as the radiosity computation progresses. Every input polygon generates a quadtree of patches that interact with patches from other quadtrees.

they don't already exist due to a previous interaction; patch j is removed from i 's interaction list and added to each of i 's children's interaction lists so that those interactions will be computed later. If the result is that patch j should be subdivided, then patch j is replaced by its children on patch i 's interaction list. This means that interactions may themselves cause subdivisions, so the process continues recursively (i.e., if patch j 's children are further subdivided in the course of computing these interactions, patch i ends up computing interactions with a tree of patches below j). Since the four children patches from a subdivision replace the parent

in place on i 's interaction list, the traversal of the tree comprising patch j 's descendants is depth first. Patch i 's interaction list is traversed fully in this way before moving on to the next patch (which may be a descendant of patch i or a different patch) and its interaction list. Figure 4.16 shows an example of this hierarchical refinement of interactions. After one iteration of computing all interactions and refinements is completed, the next iteration of the iterative algorithm starts with the quadtrees and interaction lists as they are at the end of the previous iteration.

Parallelism is available at three levels in this application: across the k input polygons, across the patches that these polygons are subdivided into (i.e., the patches in the quadtrees), and across the interactions computed for a patch. All three levels involve communication and synchronization among processors. We obtain the best performance by defining a task to be either a patch and all its interactions or a single patch-patch interaction, depending on the size of the problem, the number of processors, and the machine characteristics.

Since the computation and subdivisions are highly unpredictable, we have to use task queues and task stealing to balance the workload. The parallel implementation provides every processor with its own task queue. A processor's task queue is initialized with a subset of the initially available polygon-polygon interactions. When a patch is subdivided due to an interaction, new tasks for the subpatches are enqueued on the task queue of the processor that computed the interaction and hence did the subdivision. A processor executes tasks from its queue until no tasks are left. Then it steals tasks from other processors' queues. Locks are used to protect the task queues and to provide mutually exclusive access to patches as they are subdivided. (Note that two patches assigned to two different processes may have the same patch on their interaction list, so both processes may try to subdivide the latter patch at the same time.) Barrier synchronization is used between steps in an iteration. The parallel algorithm is nondeterministic due to task stealing and the order in which interactions and subdivisions are computed, and it has highly unstructured and unpredictable communication and data access patterns.

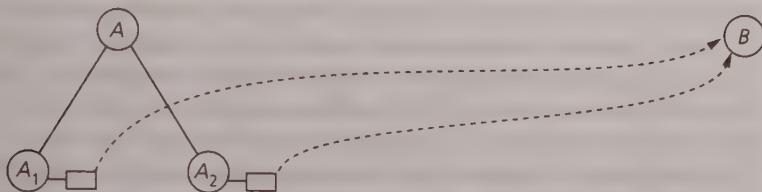
Multiprog

The workloads we have discussed so far include only parallel applications that run one at a time. However, a common use of multiprocessors, particularly small-scale shared address space multiprocessors, is as throughput engines for multiprogrammed workloads. The fine-grained resource sharing supported by these machines allows a single operating system image to service the multiple processors efficiently. Operating system activity is often a substantial component of such workloads, and the operating system itself constitutes an important, complex parallel application. The final workload we study is a multiprogrammed (time-shared) workload, consisting of a number of sequential applications and the operating system itself. The applications are two UNIX file compress jobs and two parallel compilations—or *pmakes*—in which multiple files needed to create an executable are compiled and assembled in parallel. The operating system is a version of UNIX produced by Silicon Graphics, called IRIX (version 5.2).

(1) Before refinement



(2) After the first refinement



(3) After three more refinements: A2 subdivides B; then A2 is subdivided due to B2; then A22 subdivides B1.

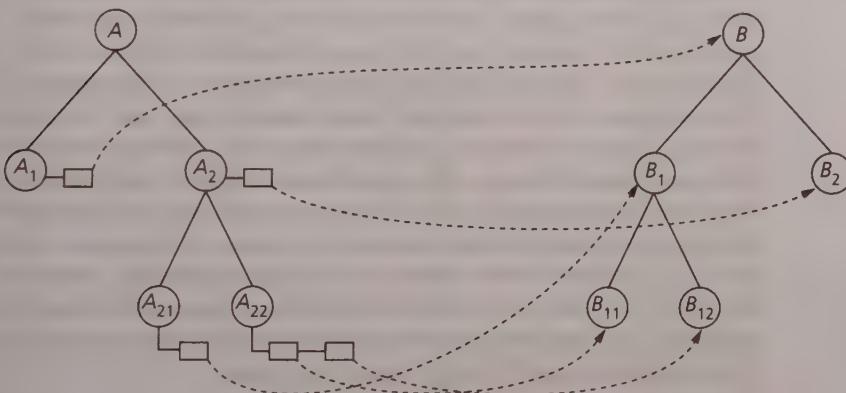


FIGURE 4.16 Hierarchical refinement of interactions and interaction lists. Binary trees are shown instead of quadtrees for clarity, and only one input polygon's interaction lists are shown.

4.4.2 Workload Characteristics

We now quantify some important basic characteristics of all our workloads, including the breakdown of data accesses into read and write or shared and private, the concurrency and inherent load balance, the inherent communication-to-computation ratio and how it scales, and the size and scaling of the important working sets. Characteristics related to spatial locality are measured in the context of specific architectural styles in later chapters. In this section, we present quantitative characterization data for 16-processor executions for our parallel applications and 8-processor executions of the multiprogrammed workload. How the characteristics of

interest scale with problem size is discussed qualitatively or analytically and is sometimes measured.

Data Access and Synchronization Characteristics

Table 4.1 summarizes the basic reference counts and dynamic frequency of synchronization events (locks and global barriers) in the different workloads. The input data sets are the default problem sizes used throughout the book, unless otherwise noted. The chosen problem sizes are large enough to be of practical interest for a machine of up to about 64 processors but small enough to simulate in a reasonable time. They are therefore at the small end of the data sets we might run in practice on 64-processor machines but are quite appropriate for smaller-scale systems.

We keep track of behavioral and timing statistics only after the child processes are created by the parent. Previous references (by the main process) are simulated but are not included in the statistics. In most of the applications, measurement begins exactly after the child processes are created. The exceptions are Ocean and Barnes-Hut. In both these cases, we are able to take advantage of the opportunity to drastically reduce the number of time-steps for the purpose of simulation (as discussed in Section 4.3.2); however, we then have to ignore cold-start misses and allow the application to settle down before starting measurement. We simulate a small number of time-steps—six for Ocean and five for Barnes-Hut—and start tracking behavioral and timing statistics after the first two time-steps. For the Multiprog workload, statistics are gathered from a checkpoint taken close to the beginning of the pmake. While for all other applications we consider only application data references, for the Multiprog workload we also consider the impact of instruction references and furthermore partition kernel references and user application references into separate categories. The table shows that the breakdown of operations into integer and floating point, read and write, and shared and private varies substantially across workloads, indicating good coverage along these axes.

Concurrency and Load Balance

We characterize load balance by measuring the algorithmic speedups—that is, speedups on the PRAM architectural model (discussed in Chapter 3) that assumes that data accesses and communication have zero latency (they just cost the instruction it takes to issue the reference). Deviations from ideal speedup are attributable to load imbalance, serialization at critical sections, and extra work due to redundant computation and parallelism management.

Figure 4.17 shows the algorithmic speedups for the six parallel programs for up to 64 processors with the default data sets. Three of the programs (Barnes-Hut, Ocean, and to a lesser extent Raytrace) speed up very well all the way to 64 processors even with the relatively small data sets. The dominant phases in these programs are data parallel across a large data set (all the particles in Barnes-Hut, an entire grid

Table 4.1 General Statistics about Application Programs

Application	Input Data Set	Total Instructions (M)	Total FLOPS (M)	Total References (M)	Total Reads (M)	Total Writes (M)	Shared Reads (M)	Shared Writes (M)	Barriers	Locks
		LU 16 × 16 blocks	512 × 512 matrix	489.52	92.20	151.07	103.09	47.99	92.79	44.74
Ocean	258 × 258 grids tolerance = 10^{-7} 4 time-steps	376.51	101.54	99.70	81.16	18.54	76.95	16.97	364	1,296
Barnes-Hut	16-K particles $\theta = 1.0$ 3 time-steps	2,002.74	239.24	720.13	406.84	313.29	225.04	93.23	7	34,516
Radix	256-K points radix = 1,024	84.62	—	14.19	7.81	6.38	3.61	2.18	11	16
Raytrace	Car scene	833.35	—	290.35	210.03	80.31	161.10	22.35	0	94,456
Radiosity	Room scene	2,297.19	—	769.56	486.84	282.72	249.67	21.88	10	210,485
Multiprog:	SGIRIX 5.2, User	1,296.43	—	500.22	350.42	149.80	—	—	—	—
Multiprog:	two compress jobs	668.10	—	212.58	178.14	34.44	—	—	—	621,505
Kernel										

For the parallel programs, shared reads and writes simply refer to all nonstack references issued by the application processes. All such references do not necessarily point to data that is truly shared by multiple processes. The Multiprog workload is not a parallel application, so it does not access shared data. A dash in a table entry means that this measurement is not applicable to or is not measured for that application (e.g., Radix has no floating-point operations). (M) denotes that measurement in that column is in millions.

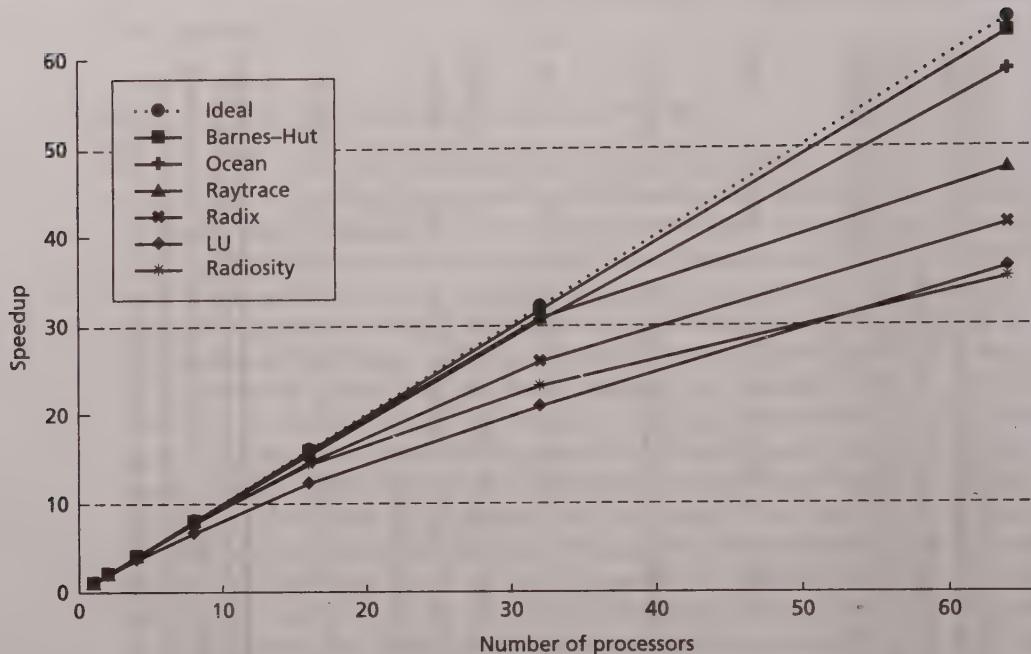


FIGURE 4.17 Algorithmic speedups for the six parallel applications. The Ideal speedup curve denotes a speedup of p with p processors.

in Ocean, and the image pixels in Raytrace). They suffer from limited parallelism and serialization only in some global reduction operations and in portions of some particular phases that are not dominant in terms of the number of instructions executed (e.g., tree building and near the root of the upward pass in Barnes, and the higher levels of the multigrid hierarchy in Ocean). Raytrace has one troublesome critical section that is heavily contended, causing serialization (in fact, this critical section is not strictly necessary for correct execution but is used to keep track of some important statistics).

All six programs display good algorithmic speedups for up to 16 or 32 processors. The programs that do not speed up very well for the higher numbers of processors with these data sets are LU, Radiosity, and Radix. In each case, this is due to the size of the input data sets rather than the inherent nature of load imbalance in the applications. In LU, the default data set results in considerable load imbalance for 64 processors, despite the block-oriented decomposition. Larger data sets (or fewer processors) reduce the imbalance by providing more blocks per processor in each step of the factorization. For Radiosity, the imbalance is also due to the use of a small data set, though it is very difficult to analyze. Finally, for Radix the poor speedup at 64 processors is due to the prefix computation when accumulating local histograms into a global histogram (see Section 4.4.1), which cannot be completely parallelized.

The time spent in this prefix computation is $O(\log p)$, while the time spent in the other phases is $O(n/p)$, so the fraction of total work in this unbalanced phase will decrease as the number of keys being sorted is increased. Thus, even these three programs can be used to evaluate larger machines, when larger data sets are chosen.

We have satisfied our criteria of not choosing parallel programs that are inherently unsuitable for the machine sizes we want to evaluate and of understanding how to choose appropriate data sets for these programs for the machine scale at hand. Let us now examine the inherent communication-to-computation ratios and working set sizes of the programs.

Communication-to-Computation Ratio

We include in the communication-to-computation ratio inherent communication as well as communication due to the first time a word is accessed by a processor, if it happens to be nonlocally allocated (i.e., cold-start misses that occur after measurement is started). Where possible, data is distributed appropriately among physically distributed memories, so we can consider this cold-start communication to be fundamental rather than artifactual. To avoid artifactual communication due to finite capacity or poor spatial locality, we simulate infinite per-processor caches and a single-word cache block. We measure communication-to-computation ratio as the number of bytes of application data communicated per instruction, averaged over all processors. For floating-point-intensive applications (LU and Ocean), we use bytes per FLOP (floating-point operation) instead of per instruction since the number of FLOPs is less sensitive to the vagaries of the compiler than the number of total instructions.

We will first look at the measured communication-to-computation ratio for the base problem size shown in Table 4.1 versus the number of processors used. This shows how the ratio increases with the number of processors under constant problem size scaling. Then, where possible, we will examine analytically how the ratio depends on the data set size and the number of processors (Table 4.2). The effects of other application parameters on the communication-to-computation ratio are discussed separately and usually qualitatively.

Figure 4.18 shows the measured results for the base problem size for our six parallel programs. The first thing we notice is that the average inherent communication-to-computation ratios are generally quite small. With processors operating at 400 million instructions per second (MIPS), a ratio of 0.1 byte per instruction is about 40 MB/s of data traffic, which is quite small for modern high-performance multiprocessor networks. Actual traffic is much higher than inherent both because of artifactual communication and because control information is sent along with data in each transfer. This indicates that it is the burstiness of communication, the other sources of communication, and the pattern of communication (e.g., all-to-all or long-range) that are likely to be the causes of communication bandwidth problems, if any. The only application for which the average ratio is quite high is Radix, so for this application, communication bandwidth is especially important to model carefully in evaluations. One reason

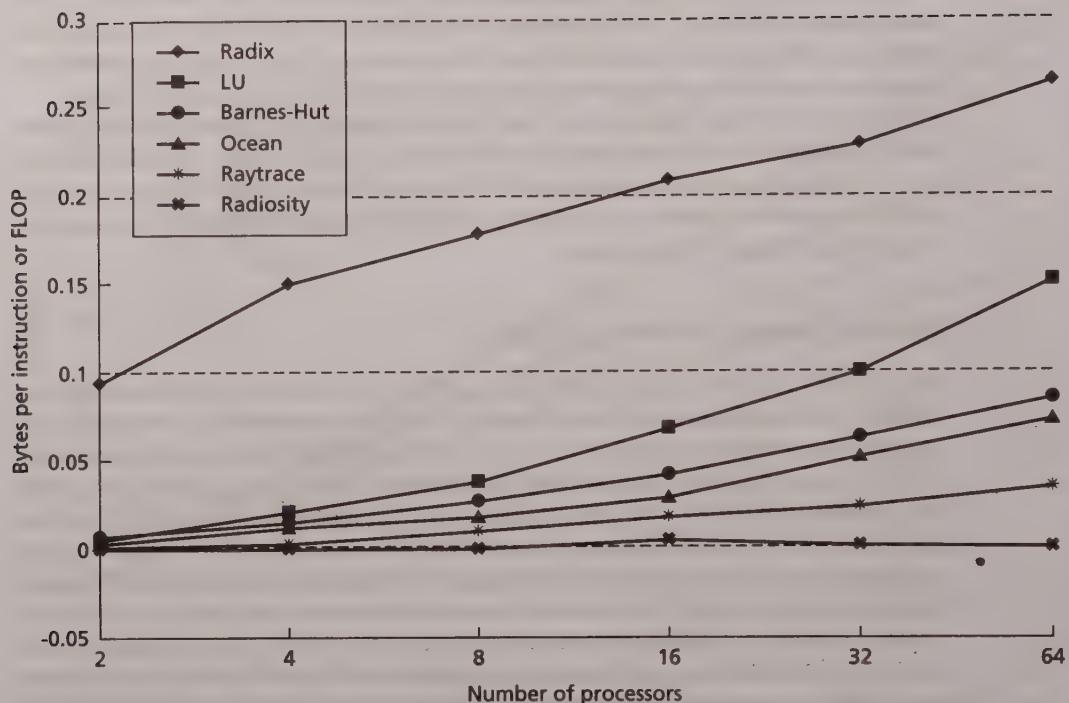


FIGURE 4.18 Communication-to-computation ratio versus processor count for the base problem size in the six parallel applications

for the low communication-to-computation ratios is that the applications we are using have been very well optimized in their assignments for parallel execution. Applications used in practice, including other versions of these applications, may exhibit higher communication-to-computation ratios.

The next observation from the figure is that the growth rates of communication-to-computation ratios are very different across applications, indicating good coverage of this behavioral property as well. These growth rates with the number of processors and with data set size (not shown in the figure) are summarized analytically in Table 4.2. The communication-to-computation ratio would change dramatically if we used a different data set size in some applications (e.g., Ocean), but at least for inherent communication it would not change much in others. Artifactual communication is a whole different story, and we shall examine communication traffic due to it in the context of different architectural types in later chapters.

While growth rates are clearly fundamental, it is important to realize that they do not reveal the constant factors in the expressions for the communication-to-computation ratio, which can be more important than asymptotic growth rates in practice. For example, if a program's ratio increases only as the logarithm of the number of processors, then asymptotically its ratio will indeed become smaller

Table 4.2 Growth Rates of Inherent Communication-to-Computation Ratio

Application	Growth Rate
LU	\sqrt{P}/\sqrt{DS}
Ocean	\sqrt{P}/\sqrt{DS}
Barnes-Hut	Approximately \sqrt{P}/\sqrt{DS}
Radiosity	Unpredictable
Radix	$(P - 1)/P$
Raytrace	Unpredictable

DS is the data set size (in bytes, say), and P is the number of processes.

than that of an application whose ratio grows as the square root of the number of processors; however, it may actually be much larger for all practical machine sizes if its constant factors are much larger. The constant factors for our applications can be determined from Figure 4.18.

Working Set Sizes

The inherent working set sizes of a program are best measured using fully associative caches and a one-word cache block and simulating the program with different cache sizes to find knees in the miss rate versus cache size curve. Smaller associativity can make the size of the cache needed to hold the working set larger than the inherent working set size, as can the use of multiword cache blocks (due to fragmentation in the cache). In our measurements, we come close to measuring inherent working set sizes by using one-level fully associative caches per processor, with a least recently used (LRU) cache replacement policy and 8-byte cache blocks. We generally use cache sizes that are powers of two, but to identify knees we change cache sizes at a finer granularity in areas where the change in miss rate with cache size is substantial.

Figure 4.19 shows the resulting miss rate versus cache size curves for our six parallel applications, with the working sets labeled as level 1 working set (L_1 WS), level 2 working set (L_2 WS), and so on. An application like Ocean has several working sets, as we discussed in Chapter 3, but we focus on the two most sharply defined and important ones. In addition to these working sets, some of the applications also have tiny working sets that do not scale with problem size or the number of processors and are therefore always expected to fit even in the cache closest to the processor; we call these the level 0 working sets (L_0 WS). They typically consist of stack data that is used by the program as temporary storage for a given primitive calculation (such as a particle-cell interaction in Barnes-Hut) and reused across these calculations. These are marked on the graphs when they are visible, but we will not discuss them further.

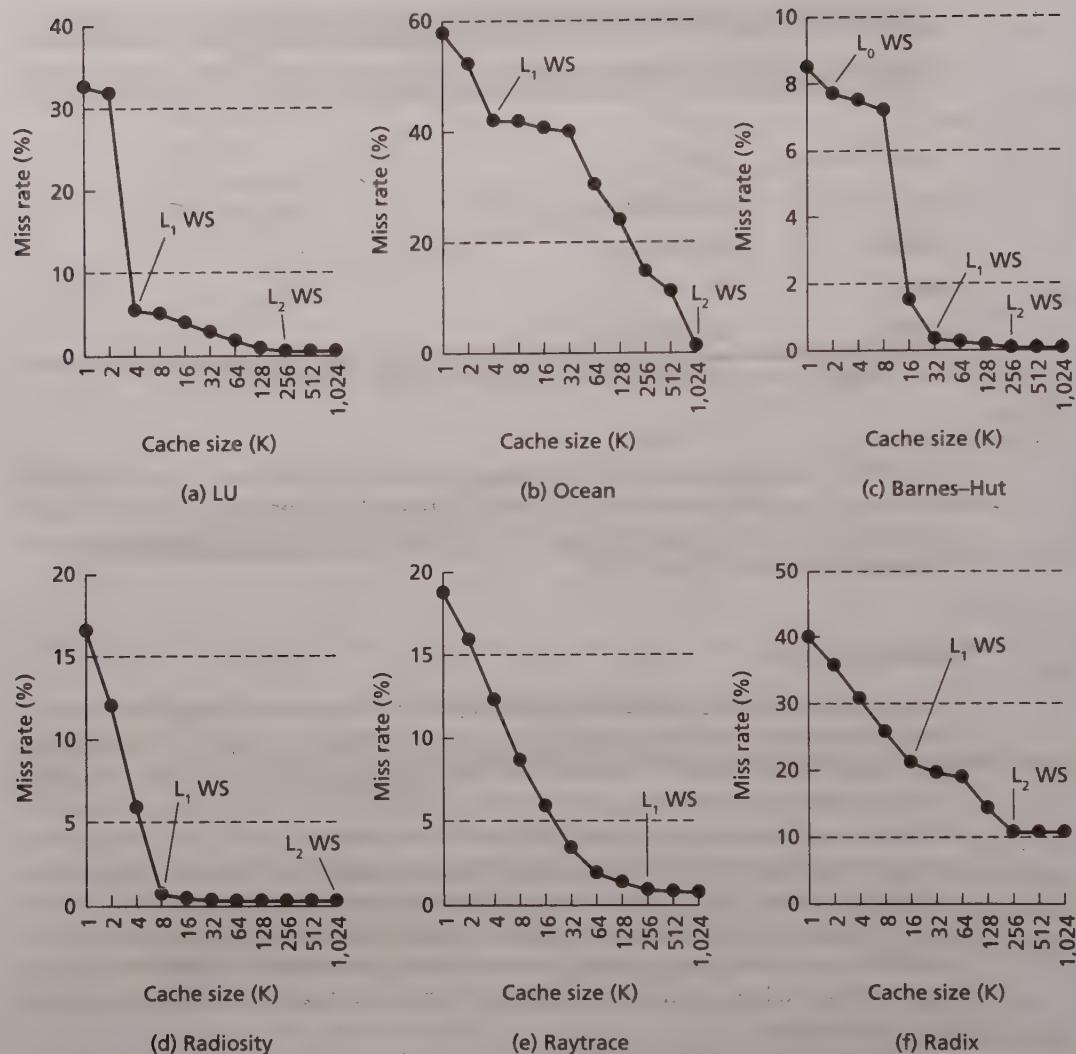


FIGURE 4.19 Working set curves for the six parallel applications in 16-processor executions. The graphs show miss rate versus cache size for fully associative first-level caches per processor and an 8-byte cache block.

We see that in most cases the working sets are very sharply defined. Table 4.3 summarizes for the different working sets how their sizes scale with application parameters and the number of processors, whether they are important to performance (at least on efficient cache-coherent machines), and whether they can be expected not to fit in a modern secondary cache for realistic problem sizes (with a

Table 4.3 Important Working Sets and Their Growth Rates for the SPLASH-2 Suite

Program	Working Set 1	Growth Rate	Important?	Realistic Not to Fit in Cache?		Working Set 2	Growth Rate	Important?	Realistic Not to Fit in Cache?
				Yes	No				
LU	One block	Fixed(B)	Yes	No	Partition of DS	DS/P	DS/P	No	Yes
Ocean	A few subrows	\sqrt{P}/\sqrt{DS}	Yes	No	Partition of DS	DS/P	DS/P	Yes	Yes
Barnes-Hut	Tree data for 1 body	$(\log DS)/\theta^2$	Yes	No	Partition of DS	DS/P	DS/P	No	Yes
Radiosity	BSP tree	$\log(\text{polygons})$	Yes	No	Unstructured	Unstructured	Unstructured	No	Yes
Radix	Histogram	Radix r	Yes	No	Partition of DS	DS/P	DS/P	Yes	Yes
Raytrace	Scene and grid data reused across rays	Unstructured	Yes	Yes	—	—	—	—	—

DS represents the data set size, and *P* is the number of processes.

reasonable degree of cache associativity, at least beyond direct mapped). The applications for which a working set has a “Yes” in each of the last two columns are Ocean, Radix, and Raytrace. Recall that in Ocean, all the major computations stream through a process’s partition of one or more grids. The large working set consists of a process’s partitions of entire grids that it might benefit from reusing. Whether or not this large working set fits in a modern secondary cache therefore depends on the grid size and the number of processors. In Radix, a process streams through all its n/p keys, at the same time heavily accessing the histogram data structure (of size proportional to the radix used). Fitting the histogram in the cache is therefore important, but this working set is not sharply defined since the keys are being streamed through at the same time. The larger working set consists of a process’s entire partition of the key data set, which may or may not fit in the cache, depending on n and p . Finally, in Raytrace we have seen that the working set is diffuse and ill defined and can become quite large, depending on the characteristics of the scene being traced and the viewpoint. For the other applications, we expect the important working sets to fit in the cache for realistic problem and machine sizes. We shall take this into account according to our methodology when we evaluate architectural trade-offs in the following chapters. In particular, for Ocean, Radix, and Raytrace, we shall choose scenarios that fit and do not fit the larger working set since both situations exist in practice.

4.5 CONCLUDING REMARKS

We now have a good understanding of the major issues in workload-driven evaluation for multiprocessors: choosing workloads, scaling problems and machines, dealing with the large parameter space, and choosing metrics. For each issue, we have a set of guidelines and steps to follow, an understanding of how to avoid pitfalls, and a means to understand the limitations of investigations. We also have a basis for our own quantitative illustration of architectural trade-offs in the rest of the book. The experiments in the book illustrate important points rather than evaluate trade-offs comprehensively, since the latter would require a much wider range of workloads and parameter variations.

We've seen that workloads should be chosen to represent a wide range of applications, behavioral patterns, and levels of optimization. While complete applications and perhaps multiprogrammed workloads are indispensable, a role exists for simpler workloads, such as microbenchmarks and kernels, as well.

We have also seen that proper workload-driven evaluation requires an understanding of the relevant behavioral properties of the workloads as well as their interactions with architectural parameters. Although this problem is complex, we have examined guidelines for dealing with the large parameter space—for evaluating both real machines and architectural trade-offs—and pruning it while still obtaining coverage of realistic situations.

The importance of understanding relevant properties of workloads was underscored by the scaling issue, which affects all important characteristics and interactions. Both execution time and memory may be constraints on scaling, and applications often have more than one parameter that determines key execution properties. We should scale programs based on an understanding of these parameters, their relationships, and their impact on execution time and memory requirements. We saw that realistic scaling models driven by the needs of applications lead to very different results of architectural significance than naive models that scale only a single application parameter. In fact, scaling is important for design as well as for evaluation. Together with an appreciation for how technology scales (for example, processor speeds relative to memory and network speeds), understanding application scaling and its implications is very important for determining appropriate resource distributions for future machines (Rothberg, Singh, and Gupta 1993).

Many interesting issues also arose in our discussion of choosing metrics for evaluation and presentation. For example, we saw that execution time (preferably with a per-processor breakdown into its major components) and speedup are both very useful metrics to present, whereas rate-based metrics such as MFLOPS or MIPS or utilization metrics can be useful for specific purposes but are too susceptible to problems as general-purpose metrics.

Finally, this chapter has described the main workloads that we use in our own illustrative workload-driven evaluation of real shared address space systems and architectural trade-offs in the rest of the book and has quantified their basic characteristics. (For message-passing systems, we examine briefly the performance of a

standard message-passing benchmark suite, the NAS Parallel Benchmarks II [NPB2] in Chapter 7.) We are now on a firm footing to proceed to core architecture and design.

4.6 EXERCISES

- 4.1 a. You are to perform a study evaluating a new feature proposed for the communication architecture of a cache-coherent machine. Your manager tells you that you may use no more than three parallel programs for the evaluation. Even though this goes against your better judgment, you have to agree. Of the seven parallel programs (excluding the multiprogrammed workload) we have examined in this chapter and in Chapter 3, which three would you choose and why?
b. Suppose you knew that the feature was designed to improve the machine's communication bandwidth. How would this affect your choice?
c. Suppose instead that the feature was designed to increase the effective replication storage for nonlocally allocated data. What programs would you choose now?
- 4.2 Identify a fundamental problem with TC scaling compared to MC scaling. Illustrate it with an example.
- 4.3 Suppose you had to evaluate the scalability of a system. One possibility is to measure the speedups under different scaling models as defined in this chapter. Another is to determine how the problem size needed to get, say, 70% parallel efficiency scales. What are the advantages and, particularly, the disadvantages or caveats for each of these? What would you actually do?
- 4.4 Your manager asks you to compare two types of systems based on the same uniprocessor node but with some interesting differences in their communication architectures. She tells you that she cares about only 10 particular applications. She instructs you to come up with a single numeric measure of which is better, given a fixed number of processors and a fixed problem size (of your choice) for each application, despite your arguments based on reading this chapter that averaging over parallel applications is not such a good idea. What additional questions would you ask her before choosing problem sizes? What measure of average would you report to her and why?
- 4.5 Often, a system may display good speedups on an application even though its communication architecture is not well suited to the application. Why might this happen? Can you design a metric alternative to speedup that measures the effectiveness of the communication architecture for the application? Discuss some of the issues and alternatives in designing such a metric.
- 4.6 A research paper you read proposes a communication architecture mechanism and tells you that starting from a given communication architecture on a machine with 32 processors, the mechanism improves performance by 40% on some workloads that are of interest to you. Is this enough information for you to decide to include that mechanism in the next machine you design? If not, list the major reasons why

- not, and say what other information you would need. Assume that the machine you are to design also has 32 processors.
- 4.7 Suppose you had to design experiments to compare different methods for implementing locks on a shared address space machine. What performance properties would you want to measure, and what “microbenchmark” experiments would you design? What would be your specific performance metrics? Now answer the same questions for global barriers.
- 4.8 You have designed a method for supporting a shared address space communication abstraction transparently in software across bus-based shared memory multiprocessors like the Intel Pentium Pro “quad” discussed in Chapter 1. Within a node or quad, coherent shared memory is supported with high efficiency in hardware; across nodes, it is supported much less efficiently and in software. Given a set of applications and the problem sizes of interest, you are about to write a research report evaluating your system. What are the interesting performance comparisons you might want to perform to understand the effectiveness of your cross-node architecture? What experiments would you design, what would each type of experiment tell you, and what metrics would you use? You have 16 bus-based multiprocessor nodes with 4 processors in each, for a total of 64 processors. Assume that you use problem-constrained scaling and that you have already chosen the problem sizes.
- 4.9 As discussed in this chapter, two types of simulations are often used in practice to study architectural trade-offs: trace-driven and execution-driven. What are the major trade-offs between trace-driven and execution-driven simulation? Under what conditions do you expect the results (say, a program’s execution time) to be significantly different?
- 4.10 Consider the difficulty and accuracy of multiprocessor simulation.
- a. What aspects of a system do you think are most difficult to simulate accurately and which are relatively easier—processor, memory system, network, communication assist, latency, bandwidth, or contention? What are the key difficulties in each case? Which of these do you think are most important to simulate very accurately, and which would you compromise on?
 - b. Consider the importance of simulating the processor pipeline appropriately when trying to evaluate the impact of trade-offs in the communication architecture. While many modern processors are superscalar and dynamically scheduled, a single-issue, statically scheduled processor is much easier to simulate. Suppose the real processor you want to model is 200 MHz with two-way issue but achieves a perfect memory CPI of 1.5. Could you model it as a single-issue 300-MHz processor for a study that wants to understand the impact of changing network transit latency on end performance? What are the major issues to consider?
- 4.11 Consider the familiar iterative nearest-neighbor grid computation on a two-dimensional grid, with subblock partitioning. Suppose we use a four-dimensional array representation, where the first two dimensions indicate the appropriate parti-

tion. The full-scale problem we would like to evaluate is an $8,192 \times 8,192$ grid of double-precision elements with 256 processors and 256 KB of direct-mapped cache per processor with a 128-byte cache block size. We cannot simulate this but instead simulate a 512×512 grid problem with 64 processors.

- a. What cache sizes would you choose and why?
 - b. List some of the dangers of choosing too small a cache.
 - c. What cache block size and associativity would you choose, and what are the issues and caveats involved?
 - d. To what extent would you consider the results representative of the full-scale problem on the larger machine? Would you use this setup to evaluate the benefits of a certain communication architecture optimization? To evaluate the speedups achievable on the machine for that application?
- 4.12 In scientific applications like the Barnes-Hut galaxy simulation, a key issue that affects scaling is error. These applications often simulate physical phenomena that occur in nature, using several approximations to represent a continuous phenomenon by a discrete model and solving it using numerical approximation techniques. Several application parameters represent distinct sources of approximation and, hence, of error in the simulation. For example, in Barnes-Hut the number of particles n represents the accuracy with which the galaxy is sampled (spatial discretization), the time-step interval Δt represents the approximation made in discretizing time, and the force calculation accuracy parameter θ determines the approximation in that calculation. The goal of an application scientist in running larger problems is usually to reduce the overall error in the simulation and have it more accurately reflect the phenomenon being simulated. Although there are no universal rules for how scientists scale different approximations, a principle that has both intuitive appeal and widespread practical applicability for physical simulations is the following: all sources of error should be scaled so that their error contributions are about equal.

For the Barnes-Hut galaxy simulation, studies in astrophysics (Hernquist 1987; Barnes and Hut 1989) show that while some error contributions are not completely independent, the following rules emerge as being valid in interesting parameter ranges:

- n : An increase in n by a factor of s leads to a decrease in simulation error by a factor of \sqrt{s} .
- Δt : The method used to integrate the particle orbits over time has a global error of the order of Δt^2 . Thus, reducing the error by a factor of \sqrt{s} (to match that due to an s -fold increase in n) requires a decrease in Δt by a factor of $\sqrt[4]{s}$. This means $\sqrt[4]{s}$ more time-steps to simulate a fixed amount of physical time, which we assume is held constant.
- θ : The force calculation error is proportional to θ^2 in the range of practical interest. Reducing the error by a factor of \sqrt{s} thus requires a decrease in θ by a factor of $\sqrt[4]{s}$.

Assume throughout this exercise that the execution time of a problem size on p processors is $1/p$ of the execution time on one processor; that is, perfect speedup is obtained for that problem size under problem-constrained scaling.

- How would you scale the other parameters θ and Δt if n is increased by a factor of s ? Call this rule *realistic* scaling, as opposed to *naive* scaling, which scales only the number of particles n .
- In a sequential program, the data set size is proportional to n and independent of the other parameters. Do the memory requirements grow differently under realistic and naive scaling in a shared address space (assuming that the important working set fits in the cache)? In message passing?
- The sequential execution time grows roughly as

$$\frac{1}{\Delta t} \cdot \frac{1}{\theta^2} \cdot n \log n$$

(assuming that a fixed amount of physical time is simulated). If n is scaled by a factor of s , how does the parallel execution time on p processors scale under realistic and under naive scaling, assuming perfect speedup?

- How does the parallel execution time grow under MC scaling, both naive and realistic, when the number of processors increases by a factor of k ? If the problem takes a day on the base machine (before it is scaled up), how long will it take in the scaled-up case on the bigger machine under both the naive and realistic models?
 - How does the number of particles that can be simulated in a shared address space grow under TC scaling, both realistic and naive, when the number of processors increases by a factor of k ?
 - Which scaling model appears more practical for this application: MC or TC?
- 4.13 For the Barnes-Hut example, how do the following execution characteristics scale under realistic and naive MC, TC, and PC scaling?
- The communication-to-computation ratio. Assume first that it depends only on n and the number of processors p , varying as \sqrt{p}/\sqrt{n} , and roughly plot the curves of growth rate of this ratio with the number of processors under the different models. Then comment on the likely effects of the other parameters under the different scaling models.
 - The sizes of the different working sets and, hence, the cache size you think is needed for good performance on a shared address space machine. Roughly plot the growth rate for the most important working set with number of processors under the different models. What major methodological conclusion in scaling does this reinforce? Comment on any differences between these trends and the trends for the amount of local replication needed in the locally essential trees version of message passing.
 - The frequency of synchronization (per unit computation, say), both locks and barriers. Describe qualitatively at least.

- d. The average frequency and size of input/output operations, assuming that every processor prints out the positions of all its assigned bodies (i) every ten time-steps; (ii) every fixed amount of physical time simulated (e.g., every year of simulated time in the galaxy's evolution).
 - e. The number of processors likely to share (access) a given piece of body data during force calculation in a coherent shared address space at a time. (As we will see in Chapter 8, this information is useful in the design of cache coherence protocols for scalable shared address space machines.)
 - f. The frequency and size of messages in an explicit message-passing implementation. Focus on the communication needed for force calculation and assume that each processor sends only one message to every other processor, communicating the data that the latter needs from the former to compute its forces in that time-step.
- 4.14 The Radix sorting application requires a parallel prefix computation to compute the global histogram from local histograms. A simplified version of the computation is as follows. Suppose each of the p processes has a local value it has computed (think of this as representing the number of keys for a given digit value in the local histogram of that process). The goal is to compute an array of p entries, in which entry i is the sum of all the local values from processors 0 through $i - 1$.
- a. Describe and implement the simplest linear method to compute this output array.
 - b. Now design a parallel method with a shorter critical path. (Hint: you can use a tree structure.) Analyze the time required for each method. Implement the two methods and compare their performance on a machine of your choice. You may use the simplified example here or the fuller example where the “local value” is in fact an array with one entry per radix digit and the output array is two-dimensional, indexed by process identifier and radix digit. That is, the fuller example does the computation for each radix digit rather than for just one.
 - c. Discuss the ways in which you could orchestrate synchronization in the latter method and the trade-offs among them.

Shared Memory Multiprocessors

The most prevalent form of parallel architecture is the multiprocessor of small to moderate scale that provides a global physical address space and symmetric access to all of main memory from any processor, often called a *symmetric multiprocessor* or SMP. Every processor has its own cache, and all the processors and memory modules attach to the same interconnect, which is usually a shared bus. SMPs dominate the server market and are becoming more common on the desktop. They are also important building blocks for larger-scale systems. The efficient sharing of resources, such as memory and processors, makes these machines attractive as “throughput engines” for multiple sequential jobs with varying memory and CPU requirements. The ability to access all shared data efficiently from any of the processors using ordinary loads and stores, together with the automatic movement and replication of shared data in the local caches, makes them attractive for parallel programming. These features are also very useful for the operating system, whose different processes share data structures and can easily run on different processors.

From the viewpoint of the layers of the communication architecture in Figure 5.1, the shared address space programming model is supported directly by hardware. User processes can read and write shared virtual addresses, and these operations are realized by individual loads and stores of shared physical addresses. In fact, the relationship between the programming model and the hardware operation is so close that they both are often referred to simply as “shared memory.” A message-passing programming model can be supported by an intervening software layer—typically a run-time library—that treats large portions of the shared address space as private to each process and manages some portions explicitly as per-process message buffers. A send/receive operation pair is realized by copying data between these buffers. The operating system need not be involved since address translation and protection on the shared buffers is provided by the hardware. For portability, most message-passing programming interfaces have indeed been implemented on popular SMPs. In fact, such implementations often deliver higher message-passing performance than traditional, distributed-memory message-passing systems—as long as contention for the shared bus and memory does not become a bottleneck—largely because of the lack of operating system involvement in communication. The operating system is still used for input/output and multiprogramming support.

Since all communication and local computation generates memory accesses in a shared address space, from a system architect’s perspective the key high-level design

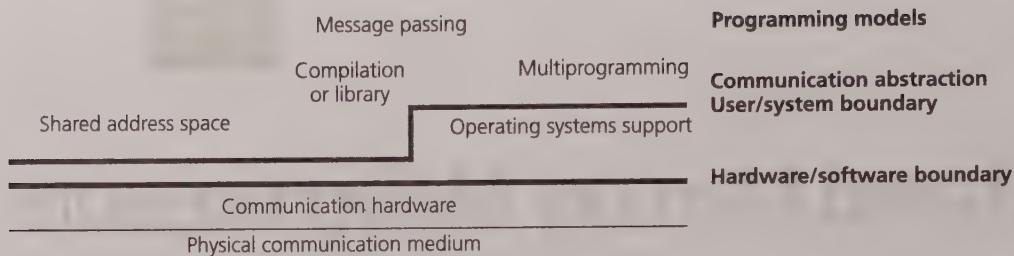


FIGURE 5.1 Layers of abstraction of the communication architecture for bus-based SMPs. A shared address space is supported directly in hardware, while message passing is supported in software.

issue is the organization of the extended memory hierarchy. In general, memory hierarchies in multiprocessors fall primarily into four categories, as shown in Figure 5.2, which correspond loosely to the scale of the multiprocessor being considered. The first three are symmetric multiprocessors (all of main memory is equally far away from all processors); while the fourth is not.

In the shared cache approach (Figure 5.2[a]), the interconnect is located between the processors and a shared first-level cache, which in turn connects to a shared main memory subsystem. Both the cache and the main memory system may be interleaved to increase available bandwidth. This approach has been used for connecting very small numbers of processors (2–8). In the mid-1980s, it was a common technique for connecting a couple of processors on a board; today, it is a possible strategy for a multiprocessor-on-a-chip, where a small number of processors on the same chip share an on-chip first-level cache. However, it applies only at a very small scale, both because the interconnect between the processors and the shared first-level cache is on the critical path that determines the latency of cache access and because the shared cache must deliver tremendous bandwidth to the multiple processors accessing it simultaneously.

In the bus-based shared memory approach (Figure 5.2[b]), the interconnect is a shared bus located between the processor's private caches (or cache hierarchies) and the shared main memory subsystem. This approach has been widely used for small-to medium-scale multiprocessors consisting of up to 20 or 30 processors. It is the dominant form of parallel machine sold today, and considerable design effort has been invested in essentially all modern microprocessors to support “cache-coherent” shared memory configurations. For example, the Intel Pentium Pro processor can attach to a coherent shared bus without any glue logic, and low-cost bus-based machines that use these processors have greatly increased the popularity of this approach. The scaling limit for these machines comes primarily due to bandwidth limitations of the shared bus and memory system.

The last two approaches are intended to be scalable to many processing nodes. The dancehall approach also places the interconnect between the caches and main memory, but the interconnect is now a scalable point-to-point network rather than a bus, and memory is divided into many logical modules that connect to logically dif-

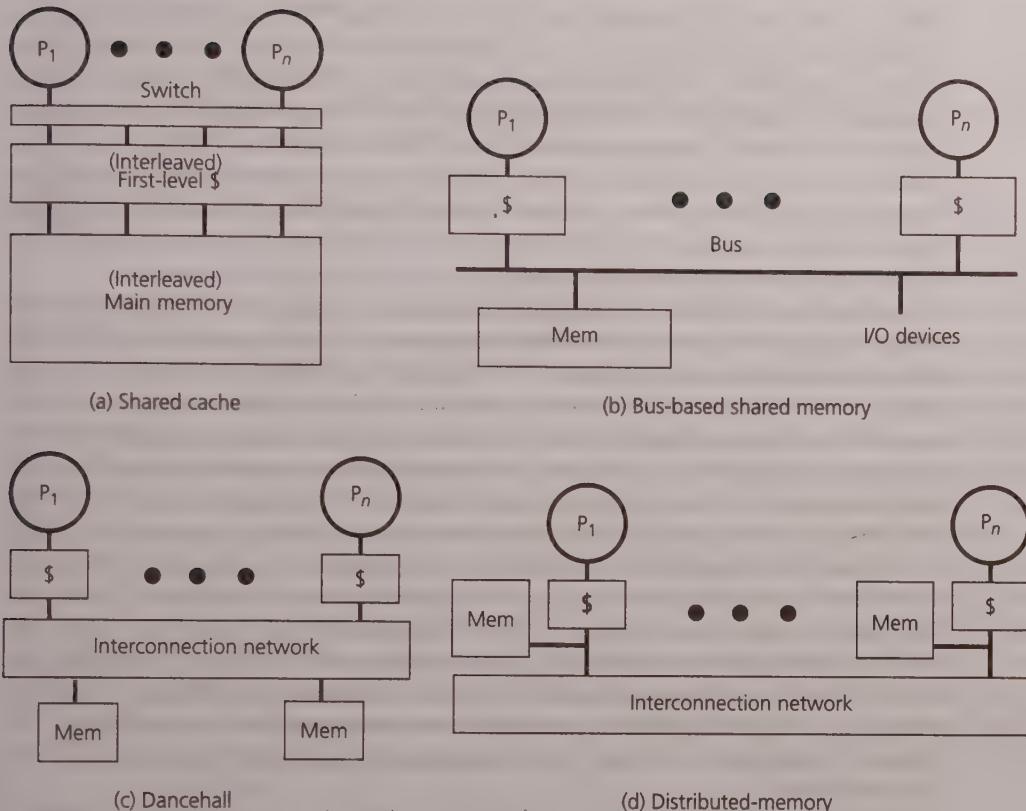


FIGURE 5.2 Common extended memory hierarchies found in multiprocessors

ferent points in the interconnect (Figure 5.2[c]). This approach is symmetric—all of main memory is uniformly far away from all processors—but its limitation is that all of memory is indeed *far* away from all processors. Especially in large systems, several “hops” or switches in the interconnect must be traversed to reach any memory module from any processor. The fourth approach, distributed-memory, is not symmetric. A scalable interconnect is located between processing nodes, but each node has its own local portion of the global main memory to which it has faster access (Figure 5.2[d]). By exploiting locality in the distribution of data, most cache misses may be satisfied in the local memory and may not have to traverse the network. This design is most attractive for scalable multiprocessors, and several chapters are devoted to the topic later in the book. Of course, it is also possible to combine multiple approaches into a single machine design—for example, a distributed-memory machine whose individual nodes are bus-based SMPs or a machine in which processors share a cache at a level of the hierarchy other than the first level.

In all cases, caches play an essential role in reducing the average data access time as seen by the processor and in reducing the bandwidth requirement each processor

places on the shared interconnect and memory system. The bandwidth requirement is reduced because the data accesses issued by a processor that are satisfied in the cache do not have to appear on the interconnect. In all but the shared cache approach, each processor has at least one level of its cache hierarchy that is private. This raises a critical challenge—namely, that of *cache coherence*. The problem arises when copies of the same memory block are present in the caches of one or more processors; if a processor writes to and hence modifies that memory block, then, unless special action is taken, the other processors will continue to access the old, stale copy of the block that is in their caches.

Currently, most small-scale multiprocessors use a shared bus interconnect with per-processor caches and a centralized main memory, whereas scalable systems use physically distributed main memory. The dancehall and shared cache approaches are employed in relatively specific settings. Specific organizations may change as technology evolves. However, besides being the most popular, the bus-based and distributed-memory organizations also illustrate the two fundamental approaches to solving the cache coherence problem, depending on the nature of the interconnect: one for the case where any transaction placed on the interconnect is visible to all processors (like a bus) and the other where the interconnect is decentralized and a point-to-point transaction is visible only to the processors at its endpoints. This chapter focuses on the logical design of protocols that exploit the fundamental properties of a bus to solve the cache coherence problem. The next chapter expands on the design issues associated with realizing these cache coherence techniques in hardware. The basic design of scalable distributed-memory multiprocessors will be addressed in Chapter 7, followed by coverage of the issues specific to scalable cache coherence in Chapters 8 and 9.

Section 5.1 describes the cache coherence problem for shared memory architectures in detail and describes the simplest example of what are called *snooping* cache coherence protocols. Coherence is not only a key hardware design concept but is a necessary part of our intuitive notion of the abstraction of memory. However, parallel software often makes stronger assumptions than coherence about how memory behaves. Section 5.2 extends the discussion of ordering begun in Chapter 1 and introduces the concept of memory consistency, which defines the semantics of shared address space. This issue has become increasingly important in computer architecture and compiler design; a large fraction of the reference manuals for most recent instruction set architectures is devoted to the memory consistency model. Once the abstractions and concepts are defined, Section 5.3 presents the design space for more realistic snooping protocols and shows how they satisfy the conditions for coherence as well as for a useful consistency model. It describes the operation of commonly used protocols at the logical state transition level. The techniques used for the quantitative evaluation of several design trade-offs at this level are illustrated in Section 5.4, using aspects of the methodology for workload-driven evaluation from Chapter 4.

The latter portions of the chapter examine the implications that cache-coherent shared memory architectures have for the software that runs on them. Section 5.5 examines how the low-level synchronization operations make use of the available

hardware primitives on cache-coherent multiprocessors and how algorithms for locks and barriers can be tailored to use the machine efficiently. Section 5.6 discusses the implications for parallel programming in general, and in particular, it discusses how temporal and spatial data locality may be exploited to reduce cache misses and traffic on the shared bus.

5.1 CACHE COHERENCE

Think for a moment about your intuitive model of what a memory should do. It should provide a set of locations that hold values, and when a location is read it should return the latest value written to that location. This is the fundamental property of the memory abstraction that we rely on in sequential programs, in which we use memory to communicate a value from a point in a program where it is computed to other points where it is used. We rely on the same property of a memory system when using a shared address space to communicate data between threads or processes running on one processor. A read returns the latest value written to the location regardless of which process wrote it. Caching does not interfere because all processes see the memory through the same cache hierarchy. We would like to rely on the same property when the two processes run on different processors that share a memory. That is, we would like the results of a program that uses multiple processes to be no different when the processes run on different physical processors than when they run (interleaved or multiprogrammed) on the same physical processor. However, when two processes see the shared memory through different caches, a danger exists that one may see the new value in its cache while the other still sees the old value.

5.1.1 The Cache Coherence Problem

The cache coherence problem in multiprocessors is both pervasive and performance critical. It is illustrated in Example 5.1.

EXAMPLE 5.1 Figure 5.3 shows three processors with caches connected via a bus to shared main memory. A sequence of accesses to location u is made by the processors. First, processor P_1 reads u from main memory, bringing a copy into its cache. Then processor P_3 reads u from main memory, bringing a copy into its cache. Then processor P_3 writes location u , changing its value from 5 to 7. With a write-through cache, this will cause the main memory location to be updated; however, when processor P_1 reads location u again (action 4), it will unfortunately read the stale value 5 from its own cache instead of the correct value 7 from main memory. This is a cache coherence problem. What happens if the caches are write back instead of write through?

Answer The situation is even worse with write-back caches. P_3 's write would merely set the dirty (or modified) bit associated with the cache block holding location u and would not update main memory right away. Only when this cache block is subsequently replaced from P_3 's cache would its contents be written back to main memory. Thus, not only will P_1 read the stale value, but when processor P_2 reads

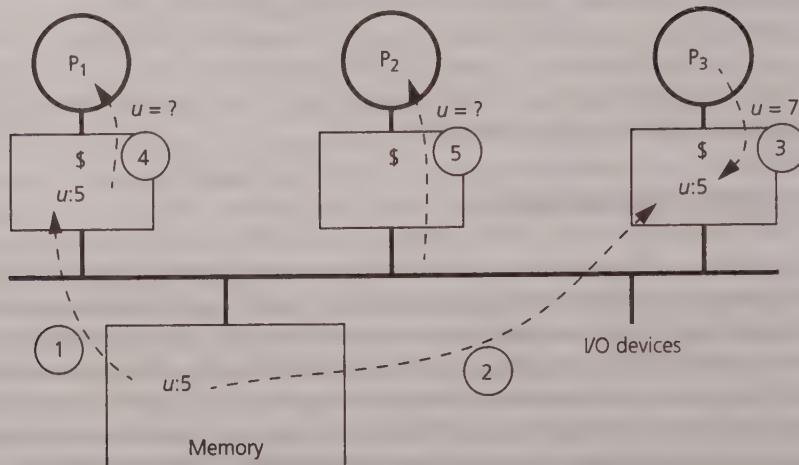


FIGURE 5.3 Example cache coherence problem. The figure shows three processors with caches connected by a bus to main memory. u is a location in memory whose contents are being read and written by the processors. The sequence in which reads and writes are done is indicated by the number listed inside the circles placed next to the arc. It is easy to see that unless special action is taken when P_3 updates the value of u to 7, P_1 will subsequently continue to read the stale value out of its cache, and P_2 will also read a stale value out of main memory.

location u (action 5), it will miss in its cache and read the stale value of 5 from main memory instead of 7. Finally, if multiple processors write distinct values to location u in their write-back caches, the final value that will reach main memory will be determined by the order in which the cache blocks containing u are replaced and will have nothing to do with the order in which the writes to u occur. ■

Clearly, the behavior described in Example 5.1 violates our intuitive notion of what a memory should do. In fact, cache coherence problems arise even in uniprocessors when I/O operations occur. Most I/O transfers are performed by direct memory access (DMA) devices that move data between memory and the peripheral component without involving the processor. When the DMA device writes to a location in main memory, unless special action is taken, the processor may continue to see the old value if that location was previously present in its cache. With write-back caches, a DMA device may read a stale value for a location from main memory because the latest value for that location is in the processor's cache. Since I/O operations are much less frequent than memory operations, several coarse solutions have been adopted in uniprocessors. For example, segments of memory space used for I/O may be marked as “uncacheable” (i.e., they do not enter the processor cache), or the processor may always use uncached load and store operations for locations used to communicate with I/O devices. For I/O devices that transfer large blocks of data at a time, such as disks, operating system support is often enlisted to ensure coherence. In many systems, the pages of memory from/to which the data is

to be transferred are flushed by the operating system from the processor's cache before the I/O is allowed to proceed. In still other systems, all I/O traffic is made to flow through the processor cache hierarchy, thus maintaining coherence. This, of course, pollutes the cache hierarchy with data that may not be of immediate interest to the processor. Fortunately, the techniques and support used to solve the multiprocessor cache coherence problem also solve the I/O coherence problem. Essentially all microprocessors today provide support for multiprocessor cache coherence.

In multiprocessors, reading and writing of shared variables by different processors is expected to be a frequent event since it is the way that multiple processes belonging to a parallel application communicate with each other. Therefore, we do not want to disallow caching of shared data or to invoke the operating system on all shared references. Rather, cache coherence needs to be addressed as a basic hardware design issue; for example, stale cached copies of a shared location (like the copy of u in P_1 's cache in Example 5.1) must be eliminated when the location is modified, either by invalidating them or updating them with the new value. In fact, the operating system itself benefits greatly from transparent, hardware-supported coherence of its data structures.

Before we explore techniques to provide coherence, it is useful to define the coherence property more precisely. Our intuitive notion that "each read should return the last value written to that location" is problematic for parallel architecture because "last" may not be well defined. Two different processors might write to the same location at the same instant, or one processor may read so soon after another writes that, due to the speed of light and other factors, there isn't time to propagate the invalidation or update to the reader. Even in the sequential case, "last" is not a chronological or physical notion but refers to latest in program order. For now, we can think of program order within a process as the order in which memory operations occur in the machine language program. The subtleties of program order are elaborated further in Section 5.2. The challenge in the parallel case is that, while program order is defined for the operations within each individual process, in order to define the semantics of a coherent memory system we need to make sense of the collection of program orders.

Let us first review the definitions of some terms in the context of uniprocessor memory systems so that we can extend the definitions for multiprocessors. By *memory operation*, we mean a single read (load), write (store), or read-modify-write access to a memory location. Instructions that perform multiple reads and writes, such as those that appear in many complex instruction sets, can be viewed as broken down into multiple memory operations, and the order in which these memory operations are executed is specified by the instruction. These memory operations within an instruction are assumed to execute atomically with respect to each other in the specified order; that is, all aspects of one appear to execute before any aspect of the next. A memory operation issues when it leaves the processor's internal environment and is presented to the memory system, which includes the caches, write buffers, bus, and memory modules. A very important point for ordering is that the only way the processor observes the state of the memory system is by issuing memory operations (e.g., reads); thus, for a memory operation to be performed with respect to the

processor means that it appears to have taken place, as far as the processor can tell from the memory operations it issues. In particular, a write operation is said to perform with respect to the processor when a subsequent read by the processor returns the value produced by either that write or a later write. A read operation is said to perform with respect to the processor when subsequent writes issued by the processor cannot affect the value returned by the read. Notice that in neither case do we specify that the physical location in the memory chip has been accessed or that specific bits of hardware have changed their values. Also, “subsequent” is well defined in the sequential case since reads and writes are ordered by the program order.

The same definitions for memory operations issuing and performing with respect to a processor apply in the parallel case; we can simply replace “the processor” with “a processor” in the definitions. The problem is that “subsequent” and “last” are not yet well defined since we do not have one program order; rather, we have separate program orders for every process, and these program orders interact when accessing the memory system. One way to sharpen our idea of a coherent memory system is to picture what would happen if there were a single shared memory and no caches. Every write and every read to a memory location would access the physical location at main memory. The operation would be performed with respect to all processors at this point and would therefore be said to *complete*. Thus, the memory would impose a serial order on all the read and write operations from all processors to the location. Moreover, the reads and writes to the location from any individual processor should be in program order within this overall serial order. In this case, then, the main memory location provides a natural point in the hardware to determine the order across processes of operations to that location. We have no reason to believe that the memory system should interleave accesses from different processors in a particular way, so any interleaving that preserves the individual program orders is reasonable. We do assume some basic fairness; eventually, the operations from each processor should be performed. Our intuitive notion of “last” can be viewed as most recent in a hypothetical serial order that maintains these properties, and “subsequent” can be defined similarly. Since this serial order must be consistent, it is important that all processors see the writes to a location in the same order (if they bother to look, i.e., to read the location).

The appearance of such a total, serial order on operations to a location is what we expect from any coherent memory system. Of course, the total order need not actually be constructed at any given point in the machine while executing the program. Particularly in a system with caches, we do not want main memory to see all the memory operations, and we want to avoid serialization whenever possible. We just need to make sure that the program behaves as if some serial order was enforced.

More formally, we say that a multiprocessor memory system is *coherent* if the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location (i.e., put all reads/writes issued by all processes into a total order) that is consistent with the results of the execution and in which

1. operations issued by any particular process occur in the order in which they were issued to the memory system by that process, and

2. the value returned by each read operation is the value written by the last write to that location in the serial order.

Two properties are implicit in the definition of coherence: *write propagation* means that writes become visible to other processes; *write serialization* means that all writes to a location (from the same or different processes) are seen in the same order by all processes. For example, write serialization means that if read operations by process P_1 to a location see the value produced by write w_1 (from P_2 , say) before the value produced by write w_2 (from P_3 , say), then reads by another process P_4 (or P_2 or P_3) also should not be able to see w_2 before w_1 . There is no need for an analogous concept of read serialization since the effects of reads are not visible to any process but the one issuing the read.

The results of a program can be viewed as the values returned by the read operations in it, perhaps augmented with an implicit set of reads to all locations at the end of the program. From the results, we cannot determine the order in which operations were actually executed by the machine or exactly when bits changed, only the order in which they appear to execute. Fortunately, this is all that matters since this is all that processors can detect. This concept will become even more important when we discuss memory consistency models.

5.1.2 Cache Coherence through Bus Snooping

Having defined the memory coherence property, let us examine techniques to solve the cache coherence problem. For instance, in Figure 5.3, how do we ensure that P_1 and P_2 see the value that P_3 wrote? In fact, a simple and elegant solution to cache coherence arises from the very nature of a bus. The bus is a single set of wires connecting several devices, each of which can observe every bus transaction, for example, every read or write on the shared bus. When a processor issues a request to its cache, the cache controller examines the state of the cache and takes suitable action, which may include generating bus transactions to access memory. Coherence is maintained by having all cache controllers “snoop” on the bus and monitor the transactions, as illustrated in Figure 5.4 (Goodman 1983). A snooping cache controller may take action if a bus transaction is *relevant* to it—that is, if it involves a memory block of which it has a copy in its cache. Thus, P_1 may take an action, such as invalidating or updating its copy of the location, if it sees the write from P_3 . In fact, since the allocation and replacement of data in caches is managed at the granularity of a cache block (usually several words long) and cache misses fetch a block of data, most often coherence is maintained at the granularity of a cache block as well. In other words, either an entire cache block is in valid state in the cache or none of it is. Thus, a cache block is the granularity of allocation in the cache, of data transfer between caches, and of coherence.

The key properties of a bus that support coherence are the following. First, all transactions that appear on the bus are visible to all cache controllers. Second, they are visible to all controllers in the same order (the order in which they appear on the bus). A coherence protocol must guarantee that all the “necessary” transactions in

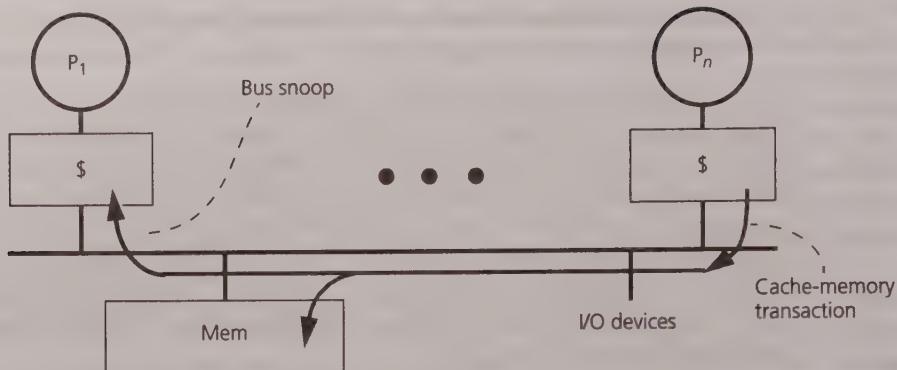


FIGURE 5.4 A snooping cache-coherent multiprocessor. Multiple processors with private caches are placed on a shared bus. Each processor's cache controller continuously "snoops" on the bus watching for relevant transaction and updates its state suitably to keep its local cache coherent. The gray arrows show the transaction being placed on the bus and accepted by main memory, as in a uniprocessor system. The black arrow indicates the snoop.

fact appear on the bus, in response to memory operations, and that the controllers take the appropriate actions when they see a relevant transaction.

The simplest illustration of maintaining coherence is a system that has single-level write-through caches. It is basically the approach followed by the first commercial bus-based SMPs in the mid-1980s. In this case, every write operation causes a write transaction to appear on the bus, so every cache controller observes every write (thus providing write propagation). If a snooping cache has a copy of the block, it either invalidates or updates its copy. Protocols that invalidate cached copies (other than the writer's copy) on a write are called *invalidation-based protocols*, whereas those that update other cached copies are called *update-based protocols*. In either case, the next time the processor with the copy accesses the block, it will see the most recent value, either through a miss or because the updated value is in its cache. Main memory always has valid data, so the cache need not take any action when it observes a read on the bus. Example 5.2 illustrates how the coherence problem in Figure 5.3 is solved with write-through caches.

EXAMPLE 5.2 Consider the scenario presented in Figure 5.3. Assuming write-through caches, show how the bus may be used to provide coherence using an invalidation-based protocol.

Answer When processor \$P_3\$ writes 7 to location \$u\$, \$P_3\$'s cache controller generates a bus transaction to update memory. Observing this bus transaction as relevant and as a write transaction, \$P_1\$'s cache controller invalidates its own copy of the block containing \$u\$. The main memory controller will update the value it has stored for location \$u\$ to 7. Subsequent reads to \$u\$ from processors \$P_1\$ and \$P_2\$ (actions 4 and 5) will both miss in their private caches and get the correct value of 7 from the main memory. ■

The check to determine if a bus transaction is relevant to a cache is essentially the same tag match that is performed for a request from the processor. The action taken may involve invalidating or updating the contents or state of that cache block and/or supplying the latest value for that block from the cache to the bus.

A snoopy cache coherence protocol ties together two basic facets of computer architecture that are also found in uniprocessors: bus transactions and the state transition diagram associated with a cache block. Recall that the first component—the bus transaction—consists of three phases: arbitration, command/address, and data. In the arbitration phase, devices that desire to initiate a transaction assert their bus request, and the bus arbiter selects one of these and responds by asserting its grant signal. Upon grant, the selected device places the command, for example, read or write, and the associated address on the bus command and address lines. All devices observe the address and, in a uniprocessor, one of them recognizes that it is responsible for the particular address. For a read transaction, the address phase is followed by data transfer. Write transactions vary from bus to bus according to whether the data is transferred during or after the address phase. For most buses, a responding device can assert a wait signal to hold off the data transfer until it is ready. This wait signal is different from the other bus signals because it is a wired-OR across all the processors; that is, it is a logical 1 if any device asserts it. The initiator does not need to know which responding device is participating in the transfer, only that there is one and whether it is ready.

The second basic facet of computer architecture leveraged by a cache coherence protocol is that each block in a uniprocessor cache has a state associated with it, along with the tag and data, which indicates the disposition of the block, (e.g., invalid, valid, dirty). The cache policy is defined by the *cache block state transition diagram*, which is a finite state machine specifying how the disposition of a block changes. Transitions for a cache block occur upon access to that block or to an address that maps to the same cache line as that block. (We refer to a cache block as the actual data, and a line as the fixed storage in the hardware cache, in exact analogy with a page and a page frame in main memory.) While only blocks that are actually in cache lines have hardware state information, logically, all blocks that are not resident in the cache can be viewed as being in either a special “not present” state or in the “invalid” state. In a uniprocessor system, for a write-through, write-no-allocate cache (Hennessy and Patterson 1996), only two states are required: valid and invalid. Initially, all the blocks are invalid. When a processor read operation misses, a bus transaction is generated to load the block from memory and the block is marked valid. Writes generate a bus transaction to update memory, and they also update the cache block if it is present in the valid state. Writes do not change the state of the block. If a block is replaced, it may be marked invalid until the memory provides the new block, whereupon it becomes valid. A write-back cache requires an additional state per cache line, indicating a “dirty” or modified block.

In a multiprocessor system, a block has a state in each cache, and these cache states change according to the state transition diagram. Thus, we can think of a block’s cache state as being a vector of p states instead of a single state, where p is the number of caches. The cache state is manipulated by a set of p distributed finite state

machines, implemented by the cache controllers. The state machine or state transition diagram that governs the state changes is the same for all blocks and all caches, but the current state of a block in different caches is different. As before, if a block is not present in a cache we can assume it to be in a special “not present” state or even in the invalid state.

In a snooping cache coherence scheme, each cache controller receives two sets of inputs: the processor issues memory requests, and the bus snooper informs about bus transactions from other caches. In response to either, the controller may update the state of the appropriate block in the cache according to the current state and the state transition diagram. It may also take an action. For example, it responds to the processor with the requested data, potentially generating new bus transactions to obtain the data. It responds to bus transactions by updating its state and sometimes intervenes in completing the transaction. Thus, a *snooping protocol* is a distributed algorithm represented by a collection of cooperating finite state machines. It is specified by the following components:

- the set of states associated with memory blocks in the local caches
- the state transition diagram, which takes as inputs the current state and the processor request or observed bus transaction and produces as output the next state for the cache block
- the actions associated with each state transition, which are determined in part by the set of feasible actions defined by the bus, the cache, and the processor design

The different state machines for a block are coordinated by bus transactions.

A simple invalidation-based protocol for a coherent write-through, write-no-allocate cache is described by the state transition diagram in Figure 5.5. As in the uniprocessor case, each cache block has only two states: invalid (I) and valid (V) (the “not present” state is assumed to be the same as invalid). The transitions are marked with the input that causes the transition and the output that is generated with the transition. For example, when a controller sees a read from its processor miss in the cache, a BusRd transaction is generated, and upon completion of this transaction the block transitions up to the valid state. Whenever the controller sees a processor write to a location, a bus transaction is generated that updates that location in main memory with no change of state. The key enhancement to the uniprocessor state diagram is that when the bus snooper sees a write transaction on the bus for a memory block that is cached locally, the controller sets the cache state for that block to invalid, thereby effectively discarding its copy. (Figure 5.5 shows this bus-induced transition with a dashed arc.) By extension, if any processor generates a write for a block that is cached by any of the others, all of the others will invalidate their copies. Thus, multiple simultaneous readers of a block may coexist without generating bus transactions or invalidations, but a write will eliminate all other cached copies.

To see how this simple write-through invalidation protocol provides coherence, we need to show that for any execution under the protocol a total order on the mem-

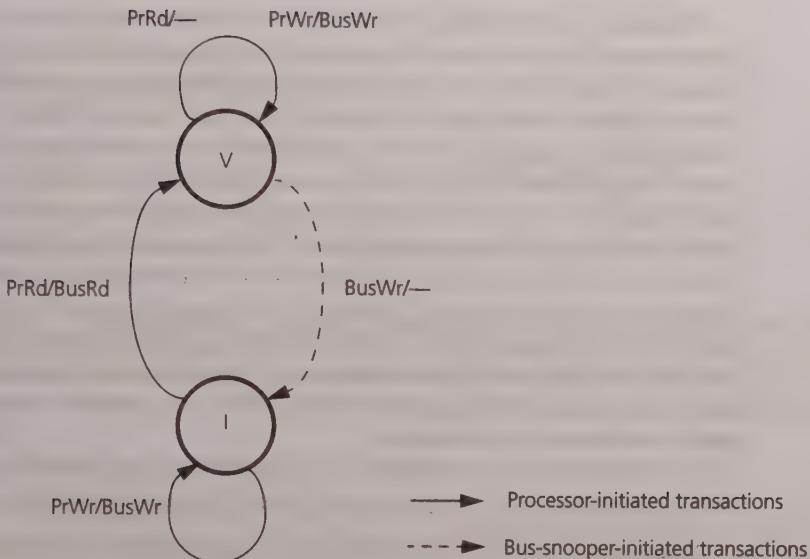


FIGURE 5.5 Snoopy coherence for a multiprocessor with write-through, write-no-allocate caches. There are two states, valid (V) and invalid (I), with intuitive semantics. The notation A/B (e.g., PrRd/BusRd) means if A is observed, then transaction B is generated. From the processor side, the requests can be read (PrRd) or write (PrWr). From the bus side, the cache controller may observe/generate transactions bus read (BusRd) or bus write (BusWr).

Memory operations for a location can be constructed that satisfies the program order and write serialization conditions. Let us assume for the present discussion that both bus transactions and the memory operations are atomic. That is, only one transaction is in progress on the bus at a time: once a request is placed on the bus, all phases of the transaction, including the data response, complete before any other request from any processor is allowed access to the bus (such a bus with atomic transactions is called an *atomic bus*). Also, a processor waits until its previous memory operation is complete before issuing another memory operation. With single-level caches, it is also natural to assume that invalidations are applied to the caches, and hence the write completes during the bus transaction itself. (These assumptions will be continued throughout this chapter and will be relaxed when we look at protocol implementations in more detail and study high-performance designs with greater concurrency in Chapter 6.) Finally, we may assume that the memory handles writes and reads in the order in which they are presented by the bus.

In the write-through protocol, all writes appear on the bus. Since only one bus transaction is in progress at a time, in any execution all writes to a location are serialized (consistently) by the order in which they appear on the shared bus, called the *bus order*. Since each snooping cache controller performs the invalidation during the bus transaction, invalidations are performed by all cache controllers in bus order.

Processors “see” writes through read operations, so for write serialization we must ensure that reads from all processors see the writes in the serialized bus order. However, reads to a location are not completely serialized since read hits may be performed independently and concurrently in their caches without generating bus transactions. To see how reads may be inserted in the serial order of writes, consider the following scenario. A read that goes on the bus (a read miss) is serialized by the bus along with the writes; it will therefore obtain the value written by the most recent write to the location in bus order. The only memory operations that do not go on the bus are read hits. In this case, the value read was placed in the cache by either the most recent write to that location by the same processor or by its most recent read miss (in program order). Since both these sources of the value appear on the bus, read hits also see the values produced in the consistent bus order. Thus, under this protocol, bus order together with program order provide enough constraints to satisfy the demands of coherence.

More generally, we can construct a (hypothetical) total order that satisfies coherence by observing the following partial orders imposed by the protocol:

- A memory operation M_2 is subsequent to a memory operation M_1 if the operations are issued by the same processor and M_2 follows M_1 in program order.
- A read operation is subsequent to a write operation W if the read generates a bus transaction that follows that for W .
- A write operation is subsequent to a read or write operation M if M generates a bus transaction and the bus transaction for the write follows that for M .
- A write operation is subsequent to a read operation if the read does not generate a bus transaction (is a hit) and is not already separated from the write by another bus transaction.

Any serial order that preserves the resulting partial order is coherent. The “subsequent” ordering relationship is transitive. An illustration of the resulting partial order is depicted in Figure 5.6, where the bus transactions associated with writes segment the individual program orders. The partial order does not constrain the ordering of read bus transactions from different processors that occur between two write transactions, though the bus will likely establish a particular order. In fact, any interleaving of read operations in the segment between two writes is a valid serial order, as long as it obeys program order.

Of course, the problem with this simple write-through approach is that every store instruction goes to memory, which is why most modern microprocessors use write-back caches (at least at the level closest to the bus). This problem is exacerbated in the multiprocessor setting, since every store from every processor consumes precious bandwidth on the shared bus, resulting in poor scalability, as illustrated by Example 5.3.

EXAMPLE 5.3 Consider a superscalar RISC processor issuing two instructions per cycle running at 200 MHz. Suppose the average CPI (clocks per instruction) for this processor is 1, 15% of all instructions are stores, and each store writes 8 bytes of data. How many processors will a 1-GB/s bus be able to support without becoming saturated?

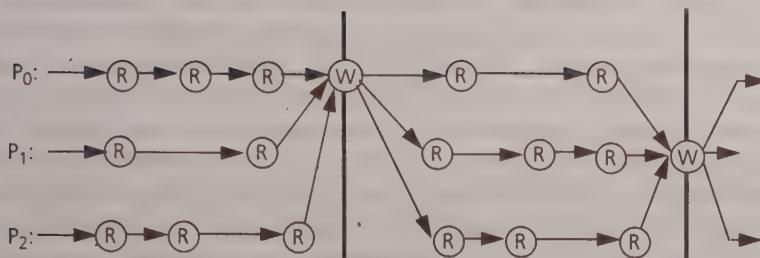


FIGURE 5.6 Partial order of memory operations for an execution with the write-through invalidation protocol. Write bus transactions define a global sequence of events between which individual processors read locations in program order. The execution is consistent with any total order obtained by interleaving the processor orders within each segment.

Answer A single processor will generate 30 million stores per second ($0.15 \text{ stores per instruction} \times 1 \text{ instruction per cycle} \times 1,000,000/200 \text{ cycles per second}$), so the total write-through bandwidth is 240 MB of data per second per processor. Even ignoring address and other information and ignoring read misses, a 1-GB/s bus will therefore support only about four processors. ■

For most applications, a write-back cache would absorb the vast majority of the writes. However, if writes do not go to memory, they do not generate bus transactions, and it is no longer clear how the other caches will observe these modifications and ensure write propagation. Also, when writes to different caches are allowed to occur concurrently, no obvious ordering mechanism exists to sequence the writes. We will need somewhat more sophisticated cache coherence protocols to make the “critical” events visible to the other caches and to ensure write serialization.

The space of protocols for write-back caches is quite large. Before we examine it, let us step back to the more general ordering issue alluded to in the introduction to this chapter and examine the semantics of a shared address space as determined by the memory consistency model.

5.2

MEMORY CONSISTENCY

Coherence, on which we have focused so far, is essential if information is to be transferred between processors by one writing to a location that the other reads. Eventually, the value written will become visible to the reader—indeed to all readers. However, coherence says nothing about when the write will become visible. Often in writing a parallel program, we want to ensure that a read returns the value of a particular write; that is, we want to establish an order between a write and a read. Typically, we use some form of event synchronization to convey this dependence, and we use more than one memory location.

Consider, for example, the code fragments executed by processors P_1 and P_2 in Figure 5.7, which we saw when discussing point-to-point event synchronization in a shared address space in Chapter 2. It is clear that the programmer intends for process P_2 to spin idly until the value of the shared variable `flag` changes to 1 and then to print the value of variable `A` as 1, since the value of `A` was updated before that of `flag` by process P_1 . In this case, we use accesses to another location (`flag`) to preserve a desired order of different processes' accesses to the same location (`A`). In particular, we assume that the write of `A` becomes visible to P_2 before the write to `flag` and that the read of `flag` by P_2 that breaks it out of its while loop completes before its read of `A` (a print operation is essentially a read). These program orders within P_1 and P_2 's accesses to different locations are not implied by coherence, which, for example, only requires that the new value for `A` eventually become visible to process P_2 , not necessarily before the new value of `flag` is observed.

The programmer might try to avoid this issue by using a barrier or other explicit event synchronization, as shown in Figure 5.8. We expect the value of `A` to be printed as 1 since `A` was set to 1 before the barrier. Even this approach has two potential problems, however. First, we are adding assumptions to the meaning of the barrier: not only do processes wait at the barrier until all of them have arrived, they also wait until all writes issued prior to the barrier have become visible to the other processors. Second, a barrier is often built using reads and writes to ordinary shared variables (e.g., `b1` in the figure) rather than with specialized hardware support. In this case, as far as the machine is concerned, it sees only accesses to different shared variables in the compiled code, not a special barrier operation. Coherence does not say anything at all about the order among these accesses.

Clearly, we expect more from a memory system than to "return the last value written" for each location. To establish order among accesses to the same location (say, `A`) by different processes, we sometimes expect a memory system to respect the order of reads and writes to different locations (`A` and `flag` or `A` and `b1`) issued by the same process. Coherence says nothing about the order in which writes to different locations become visible. Similarly, it says nothing about the order in which the reads issued to different locations by P_2 are performed with respect to P_1 . Thus, coherence does not in itself prevent an answer of 0 from being printed by either example, which is certainly not what the programmer had in mind.

In other situations, the programmer's intention may not be so clear. Consider the example in Figure 5.9. The accesses made by process P_1 are ordinary writes, and `A` and `B` are not used as flags or synchronization variables. Should we intuitively expect that if the value printed for `B` is 2, then the value printed for `A` is 1? Whatever the answer, the two print statements read different locations and coherence says nothing about the order in which the writes by P_1 become visible to P_2 . This example is in fact a fragment from Dekker's algorithm (Tanenbaum and Woodhull 1997) to determine which of two processes arrives first at a critical point as a step in ensuring mutual exclusion. The algorithm relies on writes to distinct locations by a process becoming visible to other processes in the order in which they appear in the

P_1	P_2
	/*Assume initial value of A and flag is 0*/
$A = 1;$	while (flag == 0); /*spin idly*/
flag = 1;	print A;

FIGURE 5.7 Requirements of event synchronization through flags. The figure shows two processors concurrently executing two distinct code fragments. For programmer intuition to be maintained, it must be the case that the printed value of A is 1. The intuition is that because of program order, if flag = 1 is visible to process P_2 , then it must also be the case that $A = 1$ is visible to P_2 .

P_1	P_2
	/*Assume initial value of A is 0*/
$A = 1;$...
----- BARRIER(b1) -----	----- BARRIER(b1) -----
	print A;

FIGURE 5.8 Maintaining order among accesses to a location using explicit synchronization through barriers. As in Figure 5.7, the programmer expects the value printed for A to be 1 since passing the barrier should imply that the write of A by P_1 has already completed and is therefore visible to P_2 .

P_1	P_2
	/*Assume initial values of A and B are 0*/
(1a) $A = 1;$	(2a) print B;
(1b) $B = 2;$	(2b) print A;

FIGURE 5.9 Order among accesses without synchronization. Here it is less clear what a programmer should expect since neither a flag nor any other explicit event synchronization is used.

program. Clearly, we need something more than coherence to give a shared address space a clear semantics, that is, an ordering model that programmers can use to reason about the possible results and hence the correctness of their programs.

A *memory consistency model* for a shared address space specifies constraints on the order in which memory operations must appear to be performed (i.e., to become visible to the processors) with respect to one another. This includes operations to the same locations or to different locations and by the same process or different processes, so in this sense memory consistency subsumes coherence.

5.2.1 Sequential Consistency

In the discussion in Chapter 1 of fundamental design issues for a communication architecture, Section 1.4 described informally a desirable ordering model for a shared address space: the reasoning that allows a multithreaded program to work under any possible interleaving on a uniprocessor should hold when some of the threads run in parallel on different processors. The ordering of data accesses within a process was therefore the program order, and that across processes was some interleaving of the program orders. That is, the multiprocessor case should not be able to cause values to become visible to processes in the shared address space in a manner that no sequential interleaving of accesses from different processes can generate. This intuitive model was formalized by Lamport as *sequential consistency* (SC), which is defined as follows (Lamport 1979):¹

A multiprocessor is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.

Figure 5.10 depicts the abstraction of memory provided to programmers by a sequentially consistent system (Adve and Gharachorloo 1996). It is similar to the machine model we used to introduce coherence, though now it applies to multiple memory locations. Multiple processes appear to share a single logical memory, even though in the real machine main memory may be distributed across multiple processors, each with their own private caches and buffers. Every process appears to issue and complete memory operations one at a time and atomically in program order; that is, a memory operation does not appear to be issued until the previous one from that process has completed. In addition, the common memory appears to service these requests one at a time in an interleaved manner according to an arbitrary (but hopefully fair) schedule. Memory operations appear *atomic* in this interleaved order; that is, it should appear globally (to all processes) as if one operation in the consistent interleaved order executes and completes before the next one begins.

As with coherence, it is not important in what order memory operations actually issue or even complete. What matters for sequential consistency is that they appear to complete in a manner that satisfies the constraints just described. In the example in Figure 5.9, under SC the result $(0, 2)$ for (A, B) would not be allowed—preserving our intuition—since it would then appear that the writes of A and B by process P_1 executed out of program order. However, the memory operations may actually execute and complete in the order $1b, 1a, 2b, 2a$. It does not matter that they actually complete out of program order since the results of the execution $(1, 2)$ are the same as if the operations were executed and completed in program order. On the other hand, the actual execution order $1b, 2a, 2b, 1a$ would not be sequentially consistent since it would produce the result $(0, 2)$, which is not allowed under SC. Other examples illustrating the intuitiveness of sequential consistency can be found

1. Two closely related concepts in software systems are serializability (Papadimitriou 1979) for concurrent updates to a database and linearizability (Herlihy and Wing 1987) for concurrent objects.

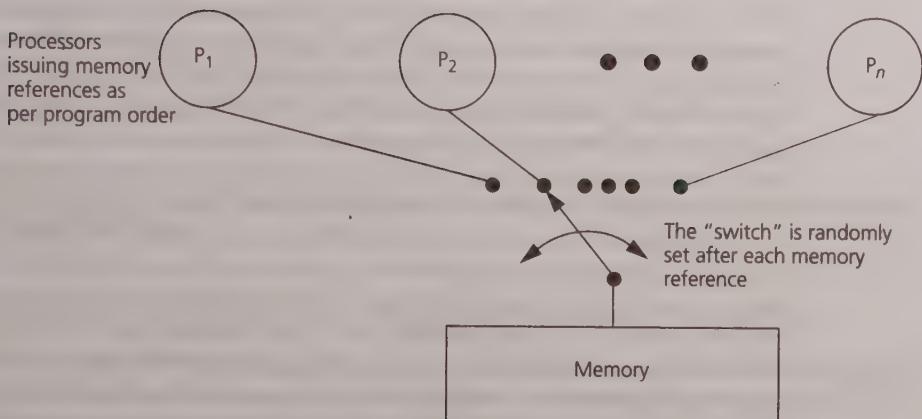


FIGURE 5.10 Programmer’s abstraction of the memory subsystem under the sequential consistency model. The model completely hides the underlying concurrency in the memory system hardware (e.g., the possible existence of distributed main memory, the presence of caches and write buffers) from the programmer.

in Exercise 5.6. Note that SC does not obviate the need for synchronization. The reason is that SC allows operations from different processes to be interleaved arbitrarily and does so at the granularity of individual instructions. Synchronization is needed if we want to preserve atomicity (mutual exclusion) across multiple memory operations from a process or if we want to enforce constraints on the interleaving across processes.

The term “program order” also bears some elaboration. Intuitively, *program order* for a process is simply the order in which statements appear according to the source code that the process executes; more specifically, it is the order in which memory operations occur in the assembly code that results from a straightforward translation of source statements one by one to machine instructions. This is not necessarily the order in which an optimizing compiler presents memory operations to the hardware since the compiler may reorder memory operations (within certain constraints, such as preserving dependences to the same location). The programmer has in mind the order of statements in the source program, but the processor sees only the order of the machine instructions. In fact, there is a “program order” at each of the interfaces in the parallel computer architecture—particularly the programming model interface seen by the programmer and the hardware/software interface—and ordering models may be defined at each. Since the programmer reasons with the source program, it makes sense to use this to define program order when discussing memory consistency models; that is, we will be concerned with the consistency model presented by the language and the underlying system to the programmer.

Implementing SC requires that the system (software and hardware) preserve the intuitive constraints defined previously. There are really two constraints. The first is the program order requirement: memory operations of a process must appear to

become visible—to itself and others—in program order. The second constraint guarantees that the total order or the interleaving across processes is consistent for all processes by requiring that the operations appear atomic. That is, it should appear that one operation is completed with respect to all processes before the next one in the total order is issued (regardless of which process issues it). The tricky part of this second requirement is making writes appear atomic, especially in a system with multiple copies of a memory word that need to be informed on a write. The *write atomicity* requirement, included in the preceding definition of sequential consistency, implies that the position in the total order at which a write appears to perform should be the same with respect to all processors. It ensures that nothing a processor does after it has seen the new value produced by a write (e.g., another write that it issues) becomes visible to other processes before they too have seen the new value for that write. In effect, the write atomicity required by SC extends the write serialization required by coherence: while write serialization says that writes to the same location should appear to all processors to have occurred in the same order, write atomicity says that all writes (to any location) should appear to all processors to have occurred in the same order. Example 5.4 shows why write atomicity is important.

EXAMPLE 5.4 Consider the three processes in Figure 5.11. Show how not preserving write atomicity violates sequential consistency.

Answer Since P_2 waits until A becomes 1 and then sets B to 1, and since P_3 waits until B becomes 1 and only then reads the value of A , from transitivity we would infer that P_3 should find the value of A to be 1. If P_2 is allowed to go on past the read of A and write B before it is guaranteed that P_3 has seen the new value of A , then P_3 may read the new value of B but read the old value of A (e.g., from its cache), violating our sequentially consistent intuition. ■

More formally, each process's program order imposes a partial order on the set of all operations; that is, it imposes an ordering on the subset of the operations that are issued by that process. An interleaving of the operations from different processes defines a total order on the set of all operations. Since the exact interleaving is not defined by SC, interleaving the partial (program) orders for different processes may yield a large number of possible total orders. The following definitions therefore apply:

- *Sequentially consistent execution.* An execution of a program is said to be sequentially consistent if the results it produces are the same as those produced by any one of the possible total orders (interleavings) as defined earlier. That is, a total order or interleaving of program orders from processes should exist that yields the same result as the actual execution.
- *Sequentially consistent system.* A system is sequentially consistent if any possible execution on that system is sequentially consistent.

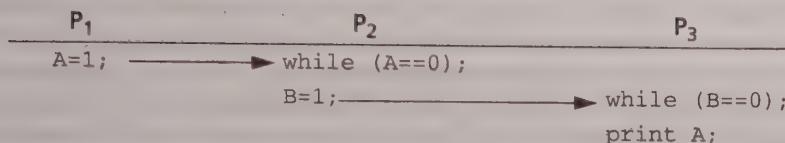


FIGURE 5.11 Example illustrating the importance of write atomicity for sequential consistency

5.2.2

Sufficient Conditions for Preserving Sequential Consistency

Having discussed the definitions and high-level requirements, let us see how a multiprocessor implementation can be made to satisfy SC. It is possible to define a set of sufficient conditions that will guarantee sequential consistency in a multiprocessor—whether bus-based or distributed, cache-coherent or not. The following set, adapted from its original form (Dubois, Scheurich, and Briggs 1986; Scheurich and Dubois 1987), is relatively simple:

1. Every process issues memory operations in program order.
2. After a write operation is issued, the issuing process waits for the write to complete before issuing its next operation.
3. After a read operation is issued, the issuing process waits for the read to complete, and for the write whose value is being returned by the read to complete, before issuing its next operation. That is, if the write whose value is being returned has performed with respect to this processor (as it must have if its value is being returned), then the processor should wait until the write has performed with respect to all processors.

The third condition is what ensures write atomicity and is quite demanding. It is not a simple local constraint because the read must wait until the logically preceding write has become globally visible. Note that these are sufficient, rather than necessary, conditions. Sequential consistency can be preserved with less serialization in many situations, as we shall see.

With program order defined in terms of the source program, it is important that the compiler should not change the order of memory operations that it presents to the hardware (processor). Otherwise, sequential consistency from the programmer's perspective may be compromised even before the hardware gets involved. Unfortunately, many of the optimizations that are commonly employed in both compilers and processors violate these sufficient conditions. For example, compilers routinely reorder accesses to different locations within a process, so a processor may in fact issue accesses out of the program order seen by the programmer. Explicitly parallel programs use uniprocessor compilers, which are concerned only about preserving dependences to the same location. Advanced compiler optimizations that greatly improve performance—such as common subexpression elimination, constant

propagation, register allocation, and loop transformations like loop splitting, loop reversal, and blocking (Wolfe 1989)—can change the order in which different locations are accessed or can even eliminate memory operations.² In practice, to constrain these compiler optimizations, multithreaded and parallel programs annotate variables or memory references that are used to preserve orders. A particularly stringent example is the use of the `volatile` qualifier in a variable declaration, which prevents the variable from being register allocated or any memory operation on the variable from being reordered with respect to operations before or after it in program order. Example 5.5 illustrates these issues.

EXAMPLE 5.5 How would reordering the memory operations in Figure 5.7 affect semantics in a sequential program (only one of the processes running), in a parallel program running on a multiprocessor, and in a threaded program in which the two processes are interleaved on the same processor? How would you solve the problem?

Answer The compiler may reorder the writes to `A` and `flag` with no impact on a sequential program. However, this can violate our intuition for both parallel programs and concurrent (or multithreaded) uniprocessor programs. In the latter case, a context switch can happen between the two reordered writes, so the process switched in may see the update to `flag` without seeing the update to `A`. Similar violations of intuition occur if the compiler reorders the reads of `flag` and `A`. For many compilers, we can avoid these reorderings by declaring the variable `flag` to be of type `volatile integer` instead of just `integer`. Other solutions are also possible and are discussed in Chapter 9. ■

Even if the compiler preserves program order, modern processors use sophisticated mechanisms like write buffers, interleaved memory, pipelining, and out-of-order execution techniques (Hennessy and Patterson 1996). These allow memory operations from a process to issue, execute, and/or complete out of program order. Like compiler optimizations, these architectural optimizations work for sequential programs because the appearance of program order in these programs requires that dependences be preserved only among accesses to the same memory location, as shown in Figure 5.12. The problem in parallel programs is that the out-of-order processing of operations to different shared variables by a process can be detected by other processes.

Preserving the sufficient conditions for SC in multiprocessors is quite a strong requirement since it limits compiler reordering and out-of-order processing techniques. Several weaker consistency models have been proposed and techniques have been developed to satisfy SC while relaxing the sufficient conditions. We will examine these approaches in the context of scalable shared address space machines in Chapter 9. For the purposes of this chapter, we assume the compiler does not reorder memory operations, so the program order that the processor sees is the same as

2. Note that register allocation, performed by modern compilers to eliminate memory operations, can affect coherence itself, not just memory consistency. For the flag synchronization example in Figure 5.7, if the compiler were to register-allocate the `flag` variable for process P_2 , the process could end up spinning forever: the cache coherence hardware updates or invalidates only the memory and the caches, not the registers of the machine, so the write propagation property of coherence is violated.

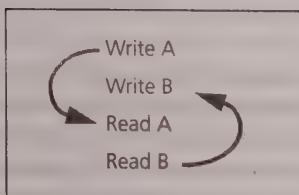


FIGURE 5.12 Preserving the orders in a sequential program running on a uniprocessor. Only the orders corresponding to the two dependence arcs must be preserved. The first two operations can be reordered without a problem, as can the last two or the middle two.

that seen by the programmer. On the hardware side, we assume that the sufficient conditions must be satisfied. To do this, we need mechanisms for a processor to detect completion of its writes so it may proceed past them (completion of reads is easy; a read completes when the data returns to the processor) and mechanisms to satisfy the condition that preserves write atomicity. For all the protocols and systems considered in this chapter, we see how they satisfy coherence (including write serialization), how they can satisfy sequential consistency (in particular, how write completion is detected and write atomicity is guaranteed), and what shortcuts can be taken while still satisfying the sufficient conditions.

For bus-based machines, the serialization imposed by transactions appearing on the shared bus is very useful in ordering memory operations. It is easy to verify that the two-state write-through invalidation protocol discussed previously actually provides sequential consistency—not just coherence—quite easily. The key observation to extend the arguments made for coherence in that system is that writes and read misses to all locations, not just to individual locations, are serialized in bus order. When a read obtains the value of a write, the write is guaranteed to have completed since it caused a previous bus transaction, thus ensuring write atomicity. When a write is performed with respect to any processor, all previous writes in bus order have completed.

5.3 DESIGN SPACE FOR SNOOPING PROTOCOLS

The beauty of snooping-based cache coherence is that the entire machinery for solving a difficult problem boils down to applying a small amount of extra interpretation to events that naturally occur in the system. The processor is completely unchanged. No explicit coherence operations must be inserted in the program. By extending the requirements on the cache controller and exploiting the properties of the bus, the reads and writes that are inherent to the program are used implicitly to keep the caches coherent, and the serialization provided by the bus maintains consistency. Each cache controller observes and interprets the bus transactions generated by others to maintain its internal state. Our initial design point with write-through caches is not very efficient, but we are now ready to study the design space for snooping protocols that make efficient use of the limited bandwidth of the shared bus. All of these use write-back caches, allowing processors to write to different blocks in their local caches concurrently without any bus transactions. Thus,

extra care is required to ensure that enough information is transmitted over the bus to maintain coherence.

Recall that with a write-back cache on a uniprocessor, a processor write miss causes the cache to read the entire block from memory, update a word, and retain the block in *modified* (or *dirty*) state so it may be written back to memory on replacement. In a multiprocessor, this modified state is also used by the protocols to indicate exclusive ownership of the block by a cache. In general, a cache is said to be the *owner* of a block if it must supply the data upon a request for that block (Sweazey and Smith 1986). A cache is said to have an *exclusive* copy of a block if it is the only cache with a valid copy of the block (main memory may or may not have a valid copy). Exclusivity implies that the cache may modify the block without notifying anyone else. If a cache does not have exclusivity, then it cannot write a new value into the block before first putting a transaction on the bus to communicate with others. The writer may have the block in its cache in a valid state, but since a transaction must be generated, it is called a write miss just like a write to a block that is not present or is invalid in the cache. If a cache has the block in modified state, then clearly it is the owner and it has exclusivity. (The need to distinguish ownership from exclusivity will become clear soon.)

On a write miss in an invalidation protocol, a special form of transaction called a *read exclusive* is used to tell other caches about the impending write and to acquire a copy of the block with exclusive ownership. This places the block in the cache in modified state, where it may now be written. Multiple processors cannot write the same block concurrently since this would lead to inconsistent values. The read-exclusive bus transactions generated by their writes will be serialized by the bus, so only one of them can have exclusive ownership of the block at a time. The cache coherence actions are driven by these two types of transactions: read and read exclusive. Eventually, when a modified block is replaced from the cache, the data is written back to memory, but this event is not caused by a memory operation to that block and is almost incidental to the protocol. A block that is not in modified state need not be written back upon replacement and can simply be dropped since memory has the latest copy. Many protocols have been devised for write-back caches, and we examine the basic alternatives.

We also consider update-based protocols. Recall that in update-based protocols, whenever a shared location is written to by a processor, its value is updated in the caches of all other processors holding that memory block.³ Thus, when these processors subsequently access that block, they can do so from their caches with low latency. The caches of all other processors are updated with a single bus transaction, thus conserving bandwidth when there are multiple sharers. In contrast, with invalidation-based protocols, on a write operation the cache state of that memory block in all other processors' caches is set to invalid, so those processors will have to obtain the block through a miss and hence a bus transaction on their next read.

3. This is a write-broadcast scenario. Read-broadcast designs have also been investigated, in which the cache containing the modified copy flushes it to the bus when it sees a read on the bus, at which point all other copies are updated too.

However, subsequent writes to that block by the same processor do not create further traffic on the bus (as they do with an update protocol) until the block is accessed by another processor. This is attractive when a single processor performs multiple writes to the same memory block before other processors access the contents of that memory block. The detailed trade-offs are more complex, and they depend on the workload offered to the machine; they will be illustrated quantitatively in Section 5.4. In general, invalidation-based strategies have been found to be more robust and are therefore provided as the default protocol by most vendors. Some vendors provide an update protocol as an option to be used for blocks corresponding to selected data structures or pages.

The choices made for the protocol (update versus invalidate) and the caching strategies directly affect the choice of states, the state transition diagram, and the associated actions. Substantial flexibility is available to the computer architect in the design task at this level. Instead of listing all possible choices, let us consider three common coherence protocols that will illustrate the design options.

5.3.1

A Three-State (MSI) Write-Back Invalidiation Protocol

The first protocol we consider is a basic invalidation-based protocol for write-back caches. It is very similar to the protocol that was used in the Silicon Graphics 4D series multiprocessor machines (Baskett, Jermoluk, and Solomon 1988). The protocol uses the three states required for any write-back cache in order to distinguish valid blocks that are unmodified (clean) from those that are modified (dirty). Specifically, the states are *modified* (M), *shared* (S), and *invalid* (I). Invalid has the obvious meaning. Shared means the block is present in an unmodified state in this cache, main memory is up-to-date, and zero or more other caches may also have an up-to-date (shared) copy. Modified, also called dirty, means that only this cache has a valid copy of the block, and the copy in main memory is stale. Before a shared or invalid block can be written and placed in the modified state, all the other potential copies must be invalidated via a read-exclusive bus transaction. This transaction serves to order the write as well as cause the invalidations and hence ensure that the write becomes visible to others (write propagation).

The processor issues two types of requests: reads (PrRd) and writes (PrWr). The read or write could be to a memory block that exists in the cache or to one that does not. In the latter case, a block currently in the cache will have to be replaced by the newly requested block, and if the existing block is in the modified state, its contents will have to be written back to main memory.

We assume that the bus allows the following transactions:

- *Bus Read* (BusRd): This transaction is generated by a PrRd that misses in the cache, and the processor expects a data response as a result. The cache controller puts the address on the bus and asks for a copy that it does not intend to modify. The memory system (possibly another cache) supplies the data.
- *Bus Read Exclusive* (BusRdX): This transaction is generated by a PrWr to a block that is either not in the cache or is in the cache but not in the modified

state. The cache controller puts the address on the bus and asks for an exclusive copy that it intends to modify. The memory system (possibly another cache) supplies the data. All other caches are invalidated. Once the cache obtains the exclusive copy, the write can be performed in the cache. The processor may require an acknowledgment as a result of this transaction.

- **Bus Write Back (BusWB):** This transaction is generated by a cache controller on a write back; the processor does not know about it and does not expect a response. The cache controller puts the address and the contents for the memory block on the bus. The main memory is updated with the latest contents.

The bus read exclusive (sometimes called *read-to-own*) is the only new transaction that would not exist except for cache coherence. The new action needed to support write-back protocols is that, in addition to changing the state of cached blocks, a cache controller can intervene in an observed bus transaction and flush the contents of the referenced block from its cache onto the bus rather than allowing the memory to supply the data. Of course, the cache controller can also initiate bus transactions as described above, supply data for write backs, or pick up data supplied by the memory system.

State Transitions

The state transition diagram that governs a block in each cache in this snooping protocol is as shown in Figure 5.13. The states are organized so that the closer the state is to the top, the more tightly the block is bound to that processor. A processor read to a block that is invalid (or not present) causes a BusRd transaction to service the miss. The newly loaded block is *promoted*, moved up in the state diagram, from invalid to the shared state in the requesting cache, whether or not any other cache holds a copy. Any other caches with the block in the shared state observe the BusRd but take no special action, allowing main memory to respond with the data. However, if a cache has the block in the modified state (there can only be one) and it observes a BusRd transaction on the bus, then it must get involved in the transaction since the copy in main memory is stale. This cache flushes the data onto the bus, in lieu of memory, and *demotes* its copy of the block to the shared state (see Figure 5.13). The memory and the requesting cache both pick up the block. This can be accomplished either by a direct cache-to-cache transfer across the bus during this BusRd transaction or by signaling an error on the BusRd transaction and generating a write transaction to update memory. In the latter case, the original cache will eventually retry its request and obtain the block from memory. (It is also possible to have the flushed data picked up only by the requesting cache but not by memory, leaving memory still out-of-date, but this requires more states [Sweazey and Smith 1986].)

Writing into an invalid block is a write miss, which is serviced by first loading the entire block and then modifying the desired bytes within it. The write miss generates a read-exclusive bus transaction, which causes all other cached copies of the block to be invalidated, thereby granting the requesting cache exclusive ownership of the

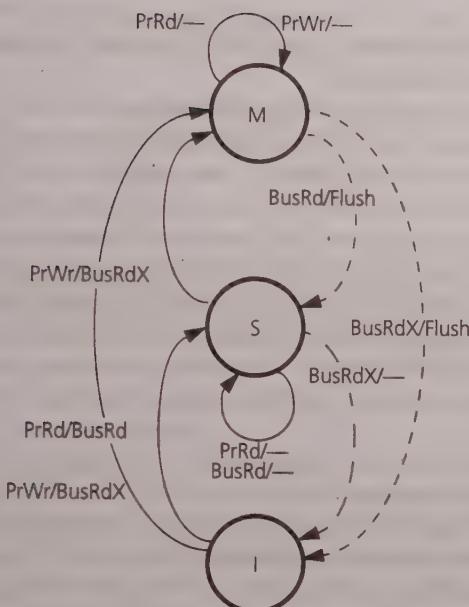


FIGURE 5.13 Basic three-state invalidation protocol. M, S, and I stand for modified, shared, and invalid states, respectively. The notation A/B means that if the controller observes the event A from the processor side or the bus side, then in addition to the state change, it generates the bus transaction or action B . “—” means null action. Transitions due to observed bus transactions are shown in dashed arcs, while those due to local processor actions are shown in bold arcs. If multiple A/B pairs are associated with an arc, it simply means that multiple inputs can cause the same state transition. For completeness, we should specify actions from each state corresponding to each observable event. If such transitions are not shown, it means that they are uninteresting and no action needs to be taken. Replacements and the write backs they may cause are not shown in the diagram for simplicity.

block. The block of data returned by the read exclusive is promoted to the modified state, and the desired bytes are then written into it. If another cache later requests exclusive access, then in response to its BusRdX transaction this block will be invalidated (demoted to the invalid state) after flushing the exclusive copy to the bus.

The most interesting transition occurs when writing into a shared block. As discussed earlier, this is treated essentially like a write miss, using a read-exclusive bus transaction to acquire exclusive ownership; we refer to it as a write miss throughout the book. The data that comes back in the read exclusive can be ignored in this case, unlike when writing to an invalid or not present block, since it is already in the cache. In fact, a common optimization to reduce data traffic in bus protocols is to introduce a new transaction, called a *bus upgrade* or BusUpgr, for this situation. A BusUpgr obtains exclusive ownership just like a BusRdX, by causing other copies to be invalidated, but it does not cause main memory or any other device to respond with the data for the block. Regardless of whether a BusUpgr or a BusRdX is used

(let us continue to assume BusRdX), the block in the requesting cache transitions to the modified state. Additional writes to the block while it is in the modified state generate no additional bus transactions.

A replacement of a block from a cache logically demotes the block to invalid (not present) by removing it from the cache. A replacement therefore causes the state machines for two blocks to change states in that cache: the one being replaced changes from its current state to invalid, and the one being brought in changes from invalid (not present) to its new state. The latter state change cannot take place before the former, which requires some care in implementation. If the block being replaced was in modified state, the replacement transition from M to I generates a write-back transaction. No special action is taken by the other caches on this transaction. If the block being replaced was in shared or invalid state, then it itself does not cause any transaction on the bus. Replacements are not shown in the state diagram for simplicity.

Note that to specify the protocol completely, for each state we must have outgoing arcs with labels corresponding to all observable events (the inputs from the processor and bus sides) and must show the actions corresponding to them. Of course, the actions and state transitions can be null sometimes, and in that case we may either explicitly specify null actions (see states S and M in Figure 5.13), or we may simply omit those arcs from the diagram (see state I). Also, since we treat the not-present state as invalid, when a new block is brought into the cache on a miss, the state transitions are performed as if the previous state of the block was invalid. Example 5.6 illustrates how the state transition diagram is interpreted.

EXAMPLE 5.6 Using the MSI protocol, show the state transitions and bus transactions for the scenario depicted in Figure 5.3.

Answer The results are shown in Figure 5.14. ■

With write-back protocols, a block can be written many times before the memory is actually updated. A read may obtain data not from memory but rather from a writer's cache, and in fact it may be this read rather than a replacement that causes memory to be updated. In addition, write hits do not appear on the bus, so the concept of a write being performed with respect to other processors is a little different. In fact, to say that a write is being performed means that the write is being "made visible." A write to a shared or invalid block is made visible by the bus read-exclusive transaction it triggers. The writer will "observe" the data in its cache after this transaction. The write will be made visible to other processors by the invalidations that the read exclusive generates, and those processors will experience a cache miss before actually observing the value written. Write hits to a modified block are visible to other processors but again are observed by them only after a miss through a bus transaction. Thus, in the MSI protocol, the write to a nonmodified block is performed or made visible when the BusRdX transaction occurs, and the write to a modified block is made visible when the block is updated in the writer's cache.

Processor Action	State in P ₁	State in P ₂	State in P ₃	Bus Action	Data Supplied By
1. P ₁ reads u	S	-	-	BusRd	Memory
2. P ₃ reads u	S	-	S	BusRd	Memory
3. P ₃ writes u	I	-	M	BusRDX	Memory
4. P ₁ reads u	S	-	S	BusRd	P ₃ cache
5. P ₂ reads u	S	S	S	BusRd	Memory

FIGURE 5.14 The three-state invalidation protocol in action for processor transactions shown in Figure 5.3. The figure shows the state of the relevant memory block at the end of each processor action, the bus transaction generated (if any), and the entity supplying the data.

Satisfying Coherence

Since both reads and writes can take place without generating bus transactions in a write-back protocol, it is not obvious that it satisfies the conditions for coherence, much less sequential consistency. Let's examine coherence first. Write propagation is clear from the preceding discussion, so let us focus on write serialization. The read-exclusive transaction ensures that the writing cache has the only valid copy when the block is actually written in the cache, just like a write transaction in the write-through protocol. It is followed immediately by the corresponding write being performed in the cache before any other bus transactions are handled by that cache controller, so it is ordered in the same way for all processors (including the writer) with respect to other bus transactions. The only difference from a write-through protocol, with regard to ordering operations to a location, is that not all writes generate bus transactions. However, the key here is that between two transactions for that block that do appear on the bus, only one processor can perform such write hits; this is the processor (say, P) that performed the most recent read-exclusive bus transaction w for the block. In the serialization, this sequence of write hits therefore appears (in program order) between w and the next bus transaction for that block. Reads by processor P will clearly see them in this order with respect to other writes. For a read by another processor, there is at least one bus transaction for that block that separates the completion of that read from the completion of these write hits. That bus transaction ensures that that read also sees the writes in the consistent serial order. Thus, reads by all processors see all writes in the same order.

Satisfying Sequential Consistency

To see how SC is satisfied, let us first appeal to the definition itself and see how a consistent global interleaving of all memory operations may be constructed. As with write-through caches, the serial arbitration for the bus in fact defines a total order on bus transactions for all blocks, not just those for a single block. All cache controllers observe read and read-exclusive bus transactions in the same order and perform invalidations in this order. Between consecutive bus transactions, each processor

performs a sequence of memory operations (read and write hits) in program order. Thus, any execution of a program defines a natural partial order:

A memory operation M_j is subsequent to operation M_i if (1) the operations are issued by the same processor and M_j follows M_i in program order, or (2) M_j generates a bus transaction that follows the memory operation for M_i .

This partial order looks graphically like that of Figure 5.6, except the local sequence within a segment has writes as well as reads and both read-exclusive and read bus transactions play important roles in establishing the orders. Between bus transactions, any interleaving of the sequences of local operations (hits) from different processors leads to a consistent total order. For writes that occur in the same segment between bus transactions, a processor will observe the writes by other processors ordered by bus transactions that it generates, and its own writes ordered by program order.

We can also see how SC is satisfied in terms of the sufficient conditions. Write completion is detected when the read-exclusive bus transaction occurs on the bus and the write is performed in the cache. The read completion condition, which provides write atomicity, is met because a read either (1) causes a bus transaction that follows that of the write whose value is being returned, in which case the write must have completed globally before the read; (2) follows such a read by the same processor in program order; or (3) follows in program order on the same processor that performed the write, in which case the processor has already waited for the write to complete (become visible) globally. Thus, all the sufficient conditions are easily guaranteed. We return to this topic when we discuss implementing protocols in Chapter 6.

Lower-Level Design Choices

To illustrate some of the implicit design choices that have been made in the protocol, let us examine more closely the transition from the M state when a BusRd for that block is observed. In Figure 5.13, we transition to state S and flush the contents of the memory block to the bus. Although it is imperative that the contents are placed on the bus, we could instead have transitioned to state I, thus giving up the block entirely. The choice of going to S versus I reflects the designer's assertion that the original processor is more likely to continue reading the block than the new processor is to write to the memory block. Intuitively, this assertion holds for mostly read data, which is common in many programs. However, a common case where it does not hold is for a flag or buffer that is used to transfer information back and forth between processes: one processor writes it, the other reads it and modifies it, then the first reads it and modifies it, and so on. Accumulations into a shared counter exhibit similar *migratory* behavior across multiple processors. The problem with betting on read sharing in these cases is that every write has to first generate an invalidation, thereby increasing its latency. Indeed, the coherence protocol used in the early Synapse multiprocessor made the alternate choice of going directly from M to I state on a BusRd, thus betting the migratory pattern would be more frequent.

Some machines (Sequent Symmetry model B and the MIT Alewife) attempt to adapt the protocol when such a migratory access pattern is observed (Cox and Fowler 1993; Dahlgren, Dubois, and Stenstrom 1994). These choices can affect the performance of the memory system, as we see later in the chapter.

5.3.2 A Four-State (MESI) Write-Back Invalidation Protocol

A concern arises with our MSI protocol if we consider a sequential application running on a multiprocessor. Such multiprogrammed use in fact constitutes the most common workload on small-scale multiprocessors. When the process reads in and modifies a data item, in the MSI protocol two bus transactions are generated even though there are never any sharers. The first is a BusRd that gets the memory block in S state, and the second is a BusRdX (or BusUpgr) that converts the block from S to M state. By adding a state that indicates that the block is the only (exclusive) copy but is not modified and by loading the block in this state, we can save the latter transaction since the state indicates that no other processor is caching the block. This new state, called *exclusive-clean* or *exclusive-unowned* (or even simply “*exclusive*”), indicates an intermediate level of binding between shared and modified. It is exclusive, so unlike the shared state, the cache can perform a write and move to the modified state without further bus transactions; but it does not imply ownership (memory has a valid copy), so unlike the modified state, the cache need not reply upon observing a request for the block. Variants of this MESI protocol are used in many modern microprocessors, including the Intel Pentium, PowerPC 601, and the MIPS R4400 used in the Silicon Graphics Challenge multiprocessors. It was first published by researchers at the University of Illinois at Urbana-Champaign (Papamarcos and Patel 1984) and is often referred to as the Illinois protocol (Archibald and Baer 1986).

The MESI protocol thus consists of four states: modified (M) or dirty, exclusive-clean (E), shared (S), and invalid (I). M and I have the same semantics as before. E, the exclusive-clean or exclusive state, means that only one cache (this cache) has a copy of the block and it has not been modified (i.e., the main memory is up-to-date). S means that potentially two or more processors have this block in their cache in an unmodified state. The bus transactions and actions needed are very similar to those for the MSI protocol.

State Transitions

When the block is first read by a processor, if a valid copy exists in another cache, then it enters the processor’s cache in the S state, as usual. However, if no other cache has a copy at the time (for example, in a sequential application), it enters the cache in the E state. When that block is written by the same processor, it can directly transition from E to M state without generating another bus transaction since no other cache has a copy. If another cache had obtained a copy in the meantime, the state of the block would have been demoted from E to S by the snooping protocol.

This protocol places a new requirement on the physical interconnect of the bus. An additional signal, called the shared signal (S), must be available to the controllers in order to determine on a BusRd if any other cache currently holds the data. During the address phase of the bus transaction, all caches determine if they contain the requested block and, if so, assert the shared signal. This signal is a wired-OR line, so the controller making the request can observe whether any other processors are caching the referenced memory block and can thereby decide whether to load a requested block in the E state or the S state.

Figure 5.15 shows a state transition diagram for a MESI protocol, still assuming that the BusUpgr transaction is not used. The notation BusRd(S) means that the bus read transaction caused the shared signal S to be asserted; BusRd(\bar{S}) means S was unasserted. A plain BusRd means that we don't care about the value of S for that transition. A write to a block in any state will promote the block to the M state, but if it was in the E state, then no bus transaction is required. Observing a BusRd will demote a block from E to S since now another cached copy exists. As usual, observing a BusRd will demote a block from M to S state and will also cause the block to be flushed onto the bus; here too, the block may be picked up only by the requesting cache and not by main memory, but this may require additional states beyond MESI. (A fifth, *owned* state may be added, which indicates that even though other shared copies of the block may exist, this cache [instead of main memory] is responsible for supplying the data when it observes a relevant bus transaction. This leads to a five-state MOESI protocol [Sweazey and Smith 1986].) Notice that it is possible for a block to be in the S state even if no other copies exist since copies may be replaced ($S \rightarrow I$) without notifying other caches. The arguments for satisfying coherence and sequential consistency are the same as in the MSI protocol.

Lower-Level Design Choices

An interesting question for bus-based protocols is who should supply the block for a BusRd transaction when both the memory and another cache have a copy of it. In the original (Illinois) version of the MESI protocol, the cache rather than main memory supplied the data—a technique called *cache-to-cache sharing*. The argument for this approach was that caches, being constructed out of SRAM rather than DRAM, could supply the data more quickly. However, this advantage is not necessarily present in modern bus-based machines, in which intervening in another processor's cache to obtain data may be more expensive than obtaining the data from main memory. Cache-to-cache sharing also adds complexity to a bus-based protocol: main memory must wait until it is certain that no cache will supply the data before driving the bus, and if the data resides in multiple caches, then a selection algorithm is needed to determine which one will provide the data. On the other hand, this technique is useful for multiprocessors with physically distributed memory (as we see in Chapter 8) because the latency to obtain the data from a nearby cache may be much smaller than that for a faraway memory unit. This effect can be especially important for machines constructed as a network of SMP nodes because caches

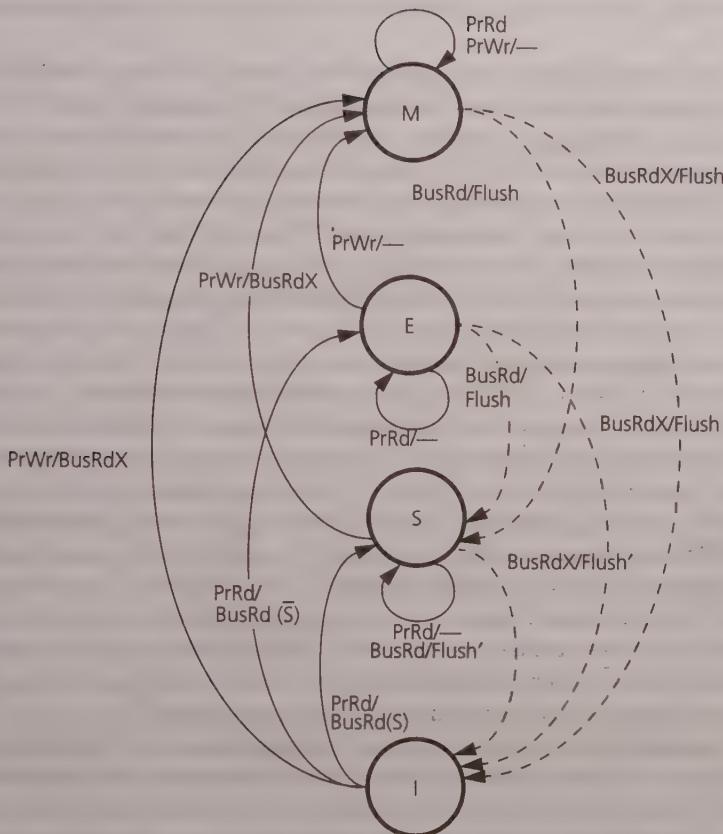


FIGURE 5.15 State transition diagram for the Illinois MESI protocol. MESI stands for the modified (dirty), exclusive, shared, and invalid states, respectively. The notation is the same as that in Figure 5.13. The E state helps reduce bus traffic for sequential programs where data is not shared. Whenever feasible, the Illinois version of the MESI protocol makes caches, rather than main memory, supply data for BusRd and BusRdX transactions. Since multiple processors may have a copy of the memory block in their cache, we need to select only one to supply the data on the bus. Flush' is true only for that processor; the remaining processors take their usual action (invalidation or no action). In general, Flush' in a state diagram indicates that the block is flushed only if cache-to-cache sharing is in use and then only by the cache that is responsible for supplying the data.

within the requestor's SMP node may supply the data. The Stanford DASH multiprocessor (Lenoski et al. 1993) used such cache-to-cache transfers for this reason.

5.3.3 A Four-State (Dragon) Write-Back Update Protocol

Let us now examine a basic update-based protocol for write-back caches. This protocol was first proposed by researchers at Xerox PARC for their Dragon multiprocessor system (McCreight 1984; Thacker, Stewart, and Satterthwaite 1988), and an

enhanced version of it is used in the Sun SparcServer multiprocessors (Catanzaro 1997).

The Dragon protocol consists of four states: exclusive-clean (E), shared-clean (Sc), shared-modified (Sm), and modified (M). Exclusive-clean (or exclusive) has the same meaning and the same motivation as before: only one cache (this cache) has a copy of the block, and it has not been modified (i.e., the main memory is up-to-date). *Shared-clean* means that potentially two or more caches (including this one) have this block, and main memory may or may not be up-to-date. *Shared-modified* means that potentially two or more caches have this block, main memory is not up-to-date, and it is this cache's responsibility to update the main memory at the time this block is replaced from the cache (i.e., this cache is the owner). A block may be in Sm state in only one cache at a time. However, it is quite possible that one cache has the block in Sm state, while others have it in Sc state. Or it may be that no cache has it in Sm state, but some have it in Sc state. This is why, when a cache has the block in Sc state, memory may or may not be up-to-date; it depends on whether some other cache has it in Sm state. M signifies exclusive ownership as before: the block is modified (dirty) and present in this cache alone, main memory is stale, and it is this cache's responsibility to supply the data and to update main memory on replacement. Note that there is no explicit invalid (I) state as in the previous protocols. This is because Dragon is an update-based protocol; the protocol always keeps the blocks in the cache up-to-date, so it is always okay to use the data present in the cache if the tag match succeeds. However, if a block is not present in a cache at all, it can be imagined in a special invalid or not-present state.⁴

The processor requests, bus transactions, and actions for the Dragon protocol are similar to the Illinois MESI protocol. The processor is still assumed to issue only read (PrRd) and write (PrWr) requests. However, since we do not have an invalid state, to specify actions on a tag mismatch we add two more request types: processor read miss (PrRdMiss) and write miss (PrWrMiss). As for bus transactions, we have bus read (BusRd), bus write back (BusWB), and a new transaction called bus update (BusUpd). The BusRd and BusWB transactions have the usual semantics. The BusUpd transaction takes the specific word (or bytes) written by the processor and broadcasts it on the bus so that all other processors' caches can update themselves. By broadcasting only the contents of the specific modified word rather than the whole cache block, it is hoped that the bus bandwidth is more efficiently utilized. (See Exercise 5.4 for reasons why this may not always be the case.) As in the MESI protocol, to support the E state, a shared signal (S) is available to the cache controller. Finally, the only new capability needed is for the cache controller to update a locally cached memory block (labeled an Update action) with the contents that are being broadcast on the bus by a relevant BusUpd transaction.

4. Logically, there is another state as well, but it is rather crude and is used to bootstrap the protocol. A "miss mode" bit is provided with each cache line to force a miss when that block is accessed. Initialization software reads data into every line in the cache with the miss mode bit turned on to ensure that the processor will miss the first time it references a block that maps to that line. After this first miss, the miss mode bit is turned off and the cache operates normally.

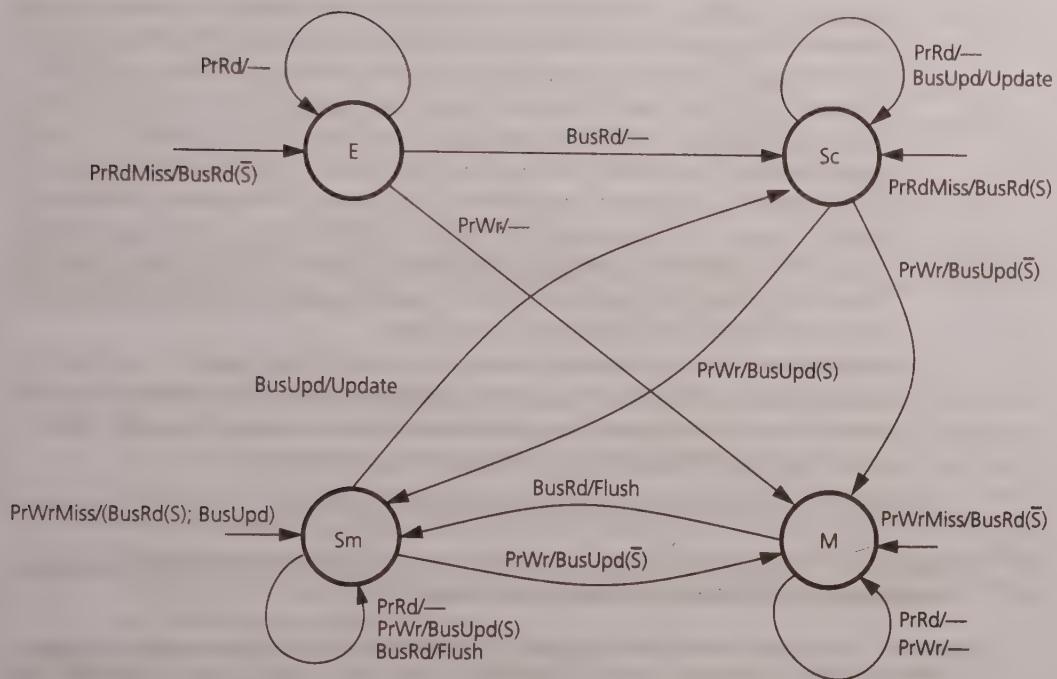


FIGURE 5.16 State transition diagram for the Dragon update protocol. The four states are exclusive (E), shared-clean (Sc), shared-modified (Sm), and modified (M). There is no invalid (I) state because the update protocol always keeps blocks in the cache up-to-date.

State Transitions

Figure 5.16 shows the state transition diagram for the Dragon update protocol. To take a processor-centric view, we can explain the diagram in terms of actions taken when a cache incurs a read miss, a write (hit or miss), or a replacement (no action is ever taken on a read hit).

- **Read miss:** A BusRd transaction is generated. Depending on the status of the shared signal (S), the block is loaded in the E or Sc state in the local cache. If the block is in M or Sm states in one of the other caches, that cache asserts the shared signal and supplies the latest data for that block on the bus, and the block is loaded in the local cache in Sc state. If the other cache had it in state M, it changes its state to Sm. If the block is in Sc state in other caches, memory supplies the data, and it is loaded in Sc state. If no other cache has a copy, then the shared line remains unasserted, the data is supplied by the main memory, and the block is loaded in the local cache in E state.
- **Write:** If the block is in the M state in the local cache, then no action needs to be taken. If the block is in the E state in the local cache, then it changes to M state and again no further action is needed. If the block is in Sc or Sm state,

however, a BusUpd transaction is generated. If any other caches have a copy of the data, they assert the shared signal, update the corresponding bytes in their cached copies, and change their state to Sc if necessary. The local cache also updates its copy of the block and changes its state to Sm if necessary. Main memory is not updated. If no other cache has a copy of the data, the shared signal remains unasserted, the local copy is updated, and the state is changed to M. Finally, if on a write the block is not present in the cache, the write is treated simply as a read-miss transaction followed by a write transaction. Thus, first a BusRd is generated. If the block is also found in other caches, a BusUpd is generated, and the block is loaded locally in the Sm state; otherwise, the block is loaded locally in the M state.

- **Replacement:** On a replacement (arcs not shown in the figure), the block is written back to memory using a bus transaction only if it is in the M or Sm state. If it is in the Sc state, then either some other cache has it in Sm state or none does, in which case it is already valid in main memory.

Example 5.7 illustrates the transitions for a familiar scenario.

EXAMPLE 5.7 Using the Dragon update protocol, show the state transitions and bus transactions for the scenario depicted in Figure 5.3.

Answer The results are shown in Figure 5.17. We can see that, whereas for processor actions 3 and 4 only one word is transferred on the bus in the update protocol, the whole memory block is transferred twice in the invalidation-based protocol. Of course, it is easy to construct scenarios in which the invalidation protocol does much better than the update protocol, and we discuss the detailed trade-offs in Section 5.4. ■

Lower-Level Design Choices

Again, many implicit design choices have been made in this protocol. For example, it is feasible to eliminate the shared-modified state. In fact, the update protocol used in the DEC Firefly multiprocessor does exactly that. The rationale is that every time the BusUpd transaction occurs, main memory can also update its contents along with the other caches holding that block; therefore, shared clean suffices, and a shared-modified state is not needed. The Dragon protocol is instead based on the assumption that the SRAM caches are much quicker to update than the DRAM main memory, so it is inappropriate to wait for main memory to be updated on all BusUpd transactions. Another subtle choice relates to the action taken on cache replacements. When a shared-clean block is replaced, should other caches be informed of that replacement via a bus transaction so that if only one cache remains with a copy of the memory block, it can change its state to exclusive or modified? The advantage of doing this would be that the bus transaction upon the replacement might not be in the critical path of a memory operation, whereas the later bus transaction that it saves might be.

Since all writes appear on the bus in an update protocol, write serialization, write completion detection, and write atomicity are all quite straightforward with a simple

Processor Action	State in P ₁	State in P ₂	State in P ₃	Bus Action	Data Supplied By
1. P ₁ reads u	E	—	—	BusRd	Memory
2. P ₃ reads u	Sc	—	Sc	BusRd	Memory
3. P ₃ writes u	Sc	—	Sm	BusUpd	P ₃ cache
4. P ₁ reads u	Sc	—	Sm	null	—
5. P ₂ reads u	Sc	Sc	Sm	BusRd	P ₃ cache

FIGURE 5.17 The Dragon update protocol in action for the processor actions shown in Figure 5.3. The figure shows the state of the relevant memory block at the end of each processor action, the bus transaction generated (if any), and the entity supplying the data.

atomic bus, a lot like they were in the write-through case. However, with both invalidation- and update-based protocols, we must address many subtle implementation issues and race conditions, even with an atomic bus and a single-level cache. We discuss this next level of protocol and hardware design in Chapter 6, as well as more realistic scenarios with pipelined buses, multilevel cache hierarchies, and hardware techniques that can reorder the completion of memory operations. Nonetheless, we can quantify many protocol trade-offs even at the state diagram level that we have been considering so far.

5.4 ASSESSING PROTOCOL DESIGN TRADE-OFFS

Like any other complex system, the design of a multiprocessor requires many inter-related decisions to be made. Even when a processor has been picked, we must decide on the maximum number of processors to be supported by the system, various parameters of the cache hierarchy (e.g., number of levels in the hierarchy, and for each level the cache size, associativity, block size, and whether the cache is write through or write back), the design of the bus (e.g., width of the data and address buses, the bus protocol), the design of the memory system (e.g., interleaved memory banks or not, width of memory banks, size of internal buffers), and the design of the I/O subsystem. Many of the issues are similar to those in uniprocessors (Smith 1982) but accentuated. For example, a write-through cache standing before the bus may be a poor choice for multiprocessors because the bus bandwidth is shared by many processors, and memory may need to be more greatly interleaved because it services cache misses from multiple processors. Greater cache associativity may also be useful in reducing conflict misses that generate bus traffic.

The cache coherence protocol is a crucial new design issue for a multiprocessor. It includes protocol class (invalidation or update), protocol states and actions, and lower-level implementation trade-offs. Protocol decisions interact with all the other design issues. On the one hand, the protocol influences the extent to which the latency and bandwidth characteristics of system components are stressed; on the other, the performance characteristics as well as the organization of the memory and communication architecture influence the choice of protocols. As discussed in

Chapter 4, these design decisions need to be evaluated relative to the behavior of real programs. Such evaluation was very common in the late 1980s, albeit using an immature set of parallel programs as workloads (Archibald and Baer 1986; Agarwal and Gupta 1988; Eggers and Katz 1988, 1989a, 1989b).

Making design decisions in real systems is part art and part science. The art draws on the past experience, intuition, and aesthetics of the designers, and the science is based in workload-driven evaluation. The goals are usually to meet a cost-performance target and to have a balanced system, so that no individual resource is a performance bottleneck yet each resource has only minimal excess capacity. This section illustrates some key protocol trade-offs by putting the workload-driven evaluation methodology from Chapter 4 into action.

5.4.1 Methodology

The basic strategy is as follows. The workload is executed on a simulator of a multiprocessor architecture, as described in Chapter 4. By observing the state transitions encountered in the simulator, we can determine the frequency of various events such as cache misses and bus transactions. We can then evaluate the effect of protocol choices in terms of other design parameters such as latency and bandwidth requirements.

Choosing parameters according to the methodology of Chapter 4, this section first establishes the basic state transition characteristics generated by the set of applications for the four-state Illinois MESI protocol. It then illustrates how to use these frequency measurements to obtain a preliminary quantitative analysis of the design trade-offs raised by the example protocols above, such as the use of the exclusive state in the MESI protocol and the use of BusUpgr rather than BusRdX transactions for the S → M transition. This section also illustrates more traditional design issues, such as how the cache block size—the granularity of both coherence and communication—impacts the latency and bandwidth needs of the applications. To understand this effect, we classify cache misses into categories such as cold, capacity, and sharing misses, examine the effect of block size on each category, and explain the results in light of application characteristics. Finally, this understanding of the applications is used to illustrate the trade-offs between invalidation-based and update-based protocols, again in light of latency and bandwidth implications.

The analysis in this section is based on the frequency of various important events, not on the absolute times taken or, therefore, the performance. This approach is common in studies of cache architecture because the results transcend particular system implementations and technology assumptions. However, it should be viewed as only a preliminary analysis since many detailed factors that might affect the performance trade-offs in real systems are abstracted away. For example, measuring state transitions provides a means of calculating miss rates and bus traffic, but realistic values for latency, overhead, and occupancy are needed to translate the rates into the actual bandwidth requirements imposed on the system. To obtain an estimate of bandwidth requirements, we may artificially assume that every reference takes a fixed number of cycles to complete. However, the bandwidth requirements them-

selves do not translate into performance directly but only indirectly by increasing the cost of misses due to contention. Contention is very difficult to estimate because it depends on the timing parameters used and on the burstiness of the traffic, which is not captured by the frequency measurements. Contention, timing, and hence performance are also affected by lower-level interactions with hardware structures (like queues and buffers) and policies.

The simulations used in this section do not model contention. Instead, they use a simple PRAM cost model: all memory operations are assumed to complete in the same amount of time (here a single cycle) regardless of whether they hit or miss in the cache. There are three main reasons for this. First, the focus is on understanding inherent protocol behavior and trade-offs in terms of event frequencies, not so much on performance. Second, since we are experimenting with different cache block sizes and organizations, we would like the interleaving of references from application processes on the simulator to be the same regardless of these choices; that is, all protocols and block sizes should see the same trace of references. With the execution-driven rather than trace-driven simulation we use, this is only possible if we make the cost of every memory operation the same in the simulations. Otherwise, if a reference misses with a small cache block but hits with a larger one, for example, then it will be delayed by different amounts in the interleaving in the two cases. It would therefore be difficult to determine which effects are inherently due to the protocol and which are due to the particular parameter values chosen. Third, realistic simulations that model contention take much more time. The disadvantage of using this simple model even to measure frequencies is that the timing model may affect some of the frequencies we observe; however, this effect is small for the applications we study.

The illustrative workloads we use are the six parallel programs (from the SPLASH-2 suite) and one multiprogrammed workload described in Chapters 3 and 4. The parallel programs run in batch mode with exclusive access to the machine and do not include operating system activity in the simulations, whereas the multiprogrammed workload includes operating system activity. The number of applications used is relatively small, but the applications are primarily for illustration as discussed in Chapter 4; the emphasis here is on choosing programs that represent important classes of computation and with widely varying characteristics. The frequencies of basic operations for the applications appear in Table 4.1. We now study them in more detail to assess design trade-offs in cache coherency protocols.

5.4.2 Bandwidth Requirement under the MESI Protocol

We begin by using the default 1-MB, single-level caches per processor, as discussed in Chapter 4. These are large enough to hold the important working sets for the default problem sizes, which is a realistic scenario for all applications. We use four-way set associativity (with LRU replacement) to reduce conflict misses and a 64-byte cache block size for realism. Driving the workloads through a cache simulator that models the Illinois MESI protocol generates the state transition frequencies shown in Table 5.1. The data is presented as the number of state transitions of a particular type per 1,000 references issued by the processors. Note in the table that a new state,

Table 5.1 State Transitions per 1,000 Data Memory References Issued by the Applications

Application	From	To				
		NP	I	E	S	M
Barnes-Hut	NP	0	0	0.0011	0.0362	0.0035
	I	0.0201	0	0.0001	0.1856	0.0010
	E	0.0000	0.0000	0.0153	0.0002	0.0010
	S	0.0029	0.2130	0	97.1712	0.1253
	M	0.0013	0.0010	0	0.1277	902.782
LU	NP	0	0	0.0000	0.6593	0.0011
	I	0.0000	0	0	0.0002	0.0003
	E	0.0000	0	0.4454	0.0004	0.2164
	S	0.0339	0.0001	0	302.702	0.0000
	M	0.0001	0.0007	0	0.2164	697.129
Ocean	NP	0	0	1.2484	0.9565	1.6787
	I	0.6362	0	0	1.8676	0.0015
	E	0.2040	0	14.0040	0.0240	0.9955
	S	0.4175	2.4994	0	134.716	2.2392
	M	2.6259	0.0015	0	2.2996	843.565
Radiosity	NP	0	0	0.0068	0.2581	0.0354
	I	0.0262	0	0	0.5766	0.0324
	E	0	0.0003	0.0241	0.0001	0.0060
	S	0.0092	0.7264	0	162.569	0.2768
	M	0.0219	0.0305	0	0.3125	839.507
Radix	NP	0	0	0.0030	1.3153	5.4051
	I	0.0485	0	0	0.4119	1.7050
	E	0.0006	0.0008	0.0284	0.0001	0
	S	0.0109	0.4156	0	84.6671	0.3051
	M	0.0173	4.2886	0	1.4982	906.945

continued

Table 5.1 State Transitions per 1,000 Data Memory References Issued by the Applications

Application		To					
		NP	I	E	S	M	
Raytrace	From	NP	0	0	1.3358	0.15486	0.0026
		I	0.0242	0	0.0000	0.3403	0.0000
		E	0.8663	0	29.0187	0.3639	0.0175
		S	1.1181	0.3740	0	310.949	0.2898
		M	0.0559	0.0001	0	0.2970	661.011
Multiprog User Data References	From	NP	0	0	0.1675	0.5253	0.1843
		I	0.2619	0	0.0007	0.0072	0.0013
		E	0.0729	0.0008	11.6629	0.0221	0.0680
		S	0.3062	0.2787	0	214.6523	0.2570
		M	0.2134	0.1196	0	0.3732	772.7819
Multiprog User Instruction References	From	NP	0	0	3.2709	15.7722	0
		I	0	0	0	0	0
		E	1.3029	0	46.7898	1.8961	0
		S	16.9032	0	0	981.2618	0
		M	0	0	0	0	0
Multiprog Kernel Data References	From	NP	0	0	1.0241	1.7209	4.0793
		I	1.3950	0	0.0079	1.1495	0.1153
		E	0.5511	0.0063	55.7680	0.0999	0.3352
		S	1.2740	2.0514	0	393.5066	1.7800
		M	3.1827	0.3551	0	2.0732	542.4318
Multiprog Kernel Instruction References	From	NP	0	0	2.1799	26.5124	0
		I	0	0	0	0	0
		E	0.8829	0	5.2156	1.2223	0
		S	24.6963	0	0	1,075.2158	0
		M	0	0	0	0	0

The data assumes 16 processors, 1-MB four-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

NP (not present), is introduced. This addition helps clarify transitions where, on a cache miss, one block is replaced (creating a transition from one of I, E, S, or M to NP) and a new block is brought in (creating a transition from NP to one of I, E, S, or M). The sum of state transitions can be greater than 1,000 even though we are presenting averages per 1,000 references because some references cause multiple state transitions. For example, a write miss can cause two transitions in the local processor's cache (e.g., S → NP for the old block and NP → M for the incoming block), in addition to transitions in other caches due to invalidations (I/E/S/M → I).⁵ This state transition frequency data is very useful for answering "what if" questions. Example 5.8 shows how we can determine the bandwidth requirement these workloads would place on the memory system.

EXAMPLE 5.8 Suppose that the integer-intensive applications run at a sustained 200 MIPS per processor and the floating-point-intensive applications at 200 MFLOPS per processor. Assuming that cache block transfers move 64 bytes on the data bus lines and that each bus transaction involves 6 bytes of command and address on the address lines, what is the traffic generated per processor?

Answer The first step is to calculate the amount of traffic per instruction. We determine what bus action is taken for each of the possible state transitions and therefore how much traffic is associated with each transaction. For example, an M → NP transition indicates that, due to a miss, a modified cache block needs to be written back. Similarly, an S → M transition indicates that an upgrade request must be issued on the bus. Flushing a modified block response to a bus transaction (e.g., the M → S or M → I transition) leads to a BusWB transaction as well. The bus transactions for all possible transitions are shown in Table 5.2. All transactions generate 6 bytes of address bus traffic and 64 bytes of data traffic, except BusUpgr, which only generates address traffic. ■

We can now compute the traffic generated. Using Table 5.2, we can convert the state transitions per 1,000 memory references in Table 5.1 to bus transactions per 1,000 memory references and convert this to address and data traffic by multiplying by the traffic per transaction. Then, using the frequency of memory accesses in Table 4.1, we can convert this to traffic per instruction or per FLOP. Finally, multiplying by the assumed processing rate, we get the address and data bandwidth requirement for each application. The result of this calculation is shown by the left-most bar for each application in Figure 5.18.

5. For the Multiprog workload, to speed up the simulations, a 32-KB instruction cache is used as a filter before passing the instruction references to the 1-MB unified instruction and data cache. The state transition frequencies for the instruction references are computed based only on those references that missed in the L₁ instruction cache. This filtering does not affect the bus traffic data that we will compute using these numbers. In addition, for Multiprog we present data separately for kernel instructions, kernel data references, user instructions, and user data references. A given reference may produce transitions of multiple types for user and kernel data. For example, if a kernel instruction miss causes a modified user data block to be written back, then we will have one transition for kernel instructions from NP → E/S and another transition for the user data reference category from M → NP.

Table 5.2 Bus Actions Corresponding to State Transitions in Illinois MESI Protocol

		To				
		NP	I	E	S	M
From	NP	—	—	BusRd	BusRd	BusRdX
	I	—	—	BusRd	BusRd	BusRdX
	E	—	—	—	—	—
	S	—	—	Not possible	—	BusUpgr
	M	BusWB	BusWB	Not possible	BusWB	—

The calculation in the preceding example gives the average bandwidth requirement under the assumption that the bus bandwidth is enough to allow the processors to execute at full speed. (In practice, bandwidth limitations may slow processors and events down, which in turn would lead to lower traffic per unit time.) This calculation provides a useful basis for sizing the number of processors that a system can support without saturating the bus. For example, on a machine such as the SGI Challenge with 1.2 GB/s of data bandwidth, the bus provides sufficient average bandwidth to support 16 processors on all the applications other than Radix for these problem sizes. A typical rule of thumb might be to leave 50% “headroom” to allow for burstiness of data transfers. If the Ocean and Multiprog workloads were also excluded, the bus could support up to 32 processors. If the bandwidth is not sufficient to support the application, the application will slow down. Thus, we would expect the speedup curve for Radix to flatten out quite quickly as the number of processors grows. In general, a multiprocessor is used for a variety of workloads, many with low per-processor bandwidth requirements, so the designer will choose to support configurations of a size that would overcommit the bus on the most demanding applications.

5.4.3 Impact of Protocol Optimizations

Given this base design point, we can evaluate protocol trade-offs under common machine parameter assumptions, as illustrated in Example 5.9.

EXAMPLE 5.9 We have described two invalidation protocols in this chapter—the basic three-state MSI protocol and the Illinois MESI protocol. The key difference is that the MESI protocol includes the existence of the exclusive state. How large is the bandwidth savings due to the E state?

Answer The main advantage of the E state is that no traffic need be generated when going from E → M. A three-state protocol would have to generate a BusUpgr transaction to acquire exclusive ownership for the memory block. To compute bandwidth savings, all we have to do is put a BusUpgr for the E → M transition in Table 5.2 and recompute the traffic as before. The middle bar in Figure 5.18 shows the resulting bandwidth requirements. ■

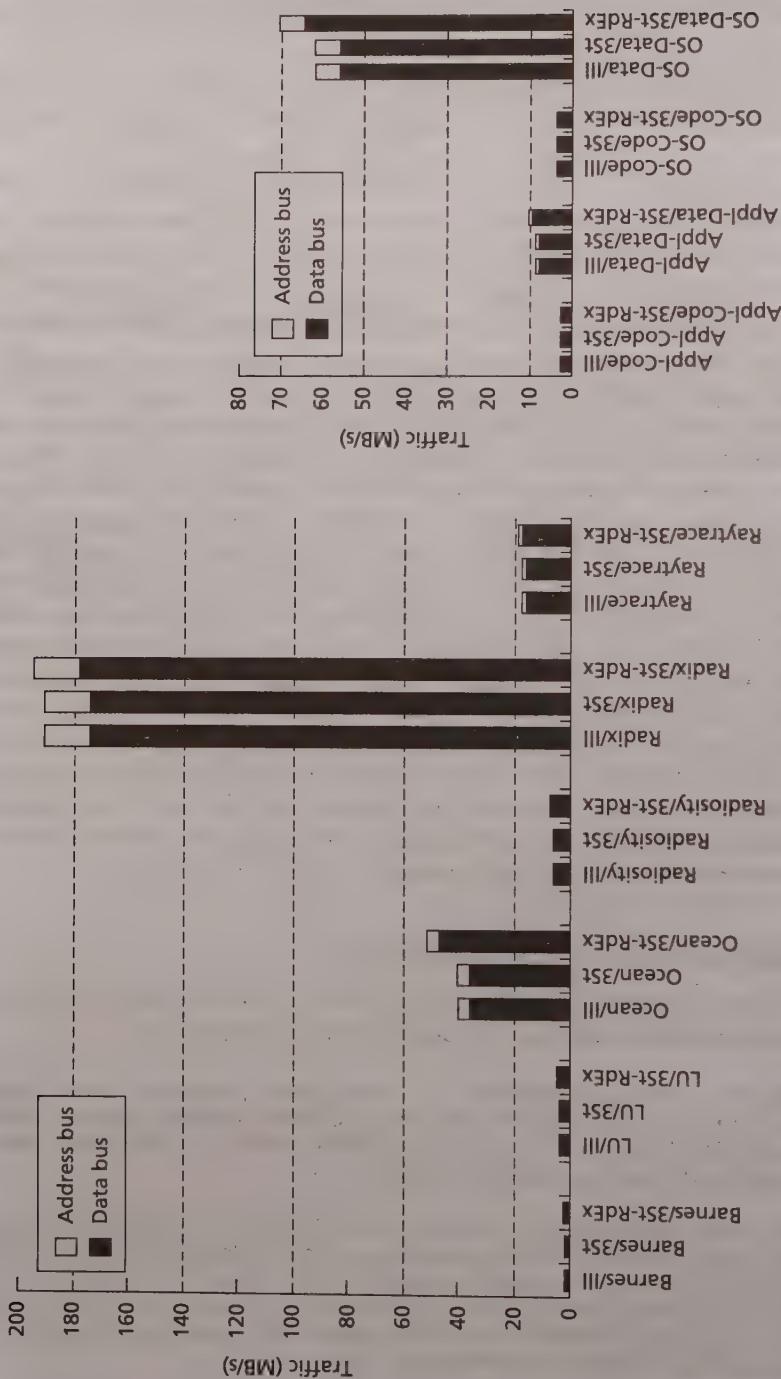


FIGURE 5.18 Per-processor bandwidth requirements for the various applications, assuming 200-MIPS/MFLOPS processors and 1-MB caches per processor. The left bar chart shows data for the parallel programs, and the right chart shows data for the Multiprog workload. The traffic is split into data traffic and address (including command) bus traffic. The leftmost bar shows traffic for the Illinois MESI protocol (Illi), the middle bar for the case where we use the basic three-state invalidation protocol without the E state (3St), and the rightmost bar for the three-state protocol when we use BusRdX instead of BusUpgr for S → M transitions (3St-RdEx).

Example 5.9 illustrates how an intuitive rationale for a more complex design may not stand up to quantitative measurement of workloads. Contrary to expectations, the E state offers negligible savings in traffic. This is true even for the Multiprog workload, which consists primarily of sequential jobs and should have benefited most. The primary reason for this negligible gain is that the fraction of $E \rightarrow M$ transitions in Table 5.1 is quite small (i.e., blocks loaded in exclusive state by a read miss are not often written while still in that state). In addition, the BusUpgr transaction that would have been needed for the $S \rightarrow M$ transition in a three-state protocol takes only 6 bytes of address traffic and no data traffic. Example 5.10 examines the advantage of the BusUpgr transaction.

EXAMPLE 5.10 Recall that even in the three-state MSI protocol, a write that finds the memory block in shared state in the cache generates a BusUpgr request on the bus rather than a BusRdX. This saves bandwidth, as no data need be transferred for a BusUpgr, but it complicates the implementation, as we shall see. The question is, how much bandwidth are we saving for taking on the extra complexity?

Answer To compute the bandwidth for the less complex implementation and a three-state protocol, all we have to do is put in BusRdX in the $E \rightarrow M$ and $S \rightarrow M$ transitions in Table 5.2 (these would all be $S \rightarrow M$ transitions in the three-state MSI protocol) and then recompute the bandwidth numbers. The results for all applications are shown in the rightmost bar in Figure 5.18. While for most applications the difference in bandwidth is small, Ocean and Multiprog kernel data references show that it can be as large as 10–20% for some applications. ■

The performance impact of these differences in bandwidth requirement depend on how the bus transactions are actually implemented. However, this high-level analysis indicates where more detailed evaluation is required.

Finally, as we discussed in Chapter 4, for the input data set sizes we are using it is important that we run the Ocean, Raytrace, and Radix applications for smaller, 64-KB cache sizes as well, to model the situation where an important working set does not fit in the cache hierarchy. The raw state transition data for this case is presented in Table 5.3, and the per-processor bandwidth requirements are shown in Figure 5.19. As we can see, not having one of the critical working sets fit in the processor cache can dramatically increase the bus bandwidth required due to capacity misses. A 1.2-GB/s bus can now barely support 4 processors for Ocean and Radix and 16 processors for Raytrace.

5.4.4 Trade-Offs in Cache Block Size

The cache organization is a critical performance factor in all modern computers, but it is especially so in multiprocessors. In the uniprocessor context, cache misses are typically categorized into the “three Cs”: compulsory, capacity, and conflict misses (Hill and Smith 1989; Hennessy and Patterson 1996). *Compulsory misses*, or *cold misses*, occur on the first reference to a memory block by a processor. *Capacity misses* occur when all the blocks that are referenced by a processor during the execution of a program do not fit in the cache (even with full associativity), so some

Table 5.3 State Transitions per 1,000 Memory References Issued by the Applications with Smaller Caches

Application		To					
		NP	I	E	S	M	
Ocean	From	NP	0	0	26.2491	2.6030	15.1459
		I	1.3305	0	0	0.3012	0.0008
		E	21.1804	0.2976	452.580	0.4489	4.3216
		S	2.4632	1.3333	0	113.257	1.1112
		M	19.0240	0.0015	0	1.5543	387.780
Radix	From	NP	0	0	3.5130	0.9580	11.3543
		I	1.6323	0	0.0001	0.0584	0.5556
		E	3.0299	0.0005	52.4198	0.0041	0.0481
		S	1.4251	0.1797	0	56.5313	0.1812
		M	8.5830	2.1011	0	0.7695	875.227
Raytrace	From	NP	0	0	7.2642	3.9742	0.1305
		I	0.0526	0	0.0003	0.2799	0.0000
		E	6.4119	0	131.944	0.7973	0.0496
		S	4.6768	0.3329	0	205.994	0.2835
		M	0.1812	0.0001	0	0.2837	660.753

The data assumes 16 processors, 64-KB four-way set-associative caches, 64-byte cache blocks, and the Illinois MESI coherence protocol.

blocks are replaced and later accessed again. *Conflict* or *collision* misses occur in caches with less than full associativity when the collection of blocks referenced by a program that maps to a single cache set does not fit in the set. They are misses that would not have occurred in a fully associative cache. Many studies have examined how cache size, associativity, and block size affect each category of miss.

Architecturally, capacity misses are reduced by enlarging the cache. Conflict misses are reduced by increasing the associativity or increasing the number of lines to map to in the cache (by increasing cache size or reducing block size). Cold misses can be reduced only by increasing the block size so that a single cold miss will bring in more data that may be accessed thereafter as well. What makes cache design challenging in uniprocessors is that these factors trade off against one another. For example, increasing the block size for a fixed cache capacity will reduce the number of blocks, so the reduced cold misses may come at the cost of increased conflict misses. Also, variations in cache organization can affect the miss penalty or the hit time and, therefore, perhaps the processor cycle time.

Cache-coherent multiprocessors introduce a fourth category of misses: *coherence* misses. These occur when blocks of data are shared among multiple caches. There

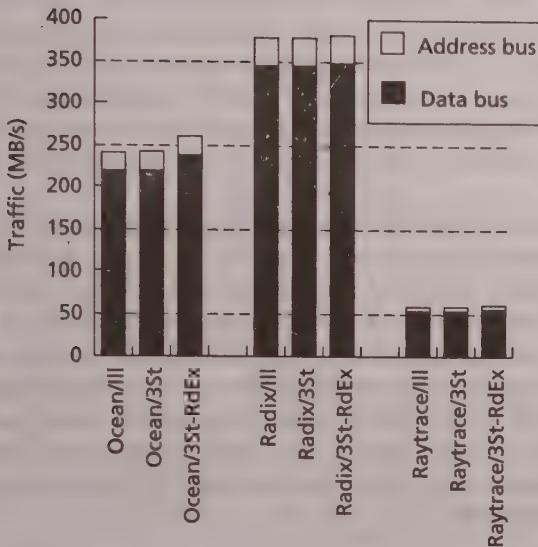


FIGURE 5.19 Per-processor bandwidth requirements for the various applications, assuming 200-MIPS/MFLOPS processors and 64-KB caches. The traffic is split into data traffic and address (including command) bus traffic. The leftmost bar shows traffic for the Illinois MESI protocol, the middle bar for the case where we use the basic three-state invalidation protocol without the E state (as described in Section 5.3.1), and the rightmost bar for the three-state protocol when we use BusRdX instead of BusUpgr for S → M transitions.

are two types: true sharing and false sharing misses. True sharing occurs when a data word produced (written) by one processor is used (read or written) by another. False sharing occurs when independent data words accessed by different processors happen to be placed in the same memory (cache) block, and at least one of the accesses is a write. The cache block size is not only the granularity (or unit) of the data fetched from the main memory, it is also typically used as the granularity of coherence. That is, on a write by a processor, the whole cache block is invalidated in other processors' caches, not just the word that is written.

More precisely, a *true sharing miss* occurs when one processor writes some words in a cache block, invalidating that block in another processor's cache, after which the second processor reads one of the modified words. It is called a “true” sharing miss because the miss truly communicates newly defined data values that are used by the second processor; such misses are essential to the correctness of the program, regardless of interactions with the machine organization or granularities. On the other hand, when one processor writes a word in a cache block and then another processor reads (or writes) a different word in the same cache block, the invalidation of the block and subsequent cache miss occurs as well, even though no useful values are being communicated between the processors. These misses are thus called *false sharing misses* (Dubois et al. 1993). As cache block size is increased, the probability of distinct variables being accessed by different processors but residing on the same

cache block increases. If at least some of these variables are written, the likelihood of false sharing misses increases as well. False sharing misses would not occur with a one-word cache block size, while true sharing misses would. Technology pushes in the direction of large cache block sizes (e.g., DRAM organization and access modes and the need to obtain high-bandwidth data transfers by amortizing overhead), so it is important to understand the potential impact of false sharing misses and how they may be avoided.

True sharing misses are inherent to a given parallel decomposition and assignment, so, like cold misses, the only way to reduce them is by increasing the block size and increasing spatial locality of communicated data. False sharing misses, on the other hand, are an example of the artifactual communication discussed in Chapter 3 since they are caused by interactions with the architecture. In contrast to true sharing and cold misses, false sharing misses can be decreased by reducing the cache block size, as well as by a host of other optimizations in software (orchestration) and hardware that we shall discuss later. Thus, a fundamental tension exists in determining the best cache block size, which can only be resolved by evaluating the options against real programs.

A Classification of Cache Misses

The flowchart in Figure 5.20 gives a detailed algorithm for classifying cache misses in cache-coherent multiprocessors.⁶ Understanding the details is not critical for now—it is enough for the rest of the chapter to understand only the preceding definitions—but it adds insight and is a useful exercise. In the algorithm, the *lifetime* of a block in a cache is defined as the time interval during which the block remains valid in the cache, that is, the time from the occurrence of the miss that loads the block in the cache until its invalidation, replacement, or the end of the program. We cannot classify a cache miss when it occurs but only when the fetched memory block is replaced or invalidated in the cache, because it is only then that we know whether true sharing or only false sharing occurred during that lifetime. Let us consider the simple cases first. Cases 1 and 2 are straightforward cold misses occurring on previously unwritten blocks. Cases 7 and 8 reflect false and true sharing on a block that was previously invalidated in the cache but yet replaced by another. The type of sharing is determined by whether the specific word or words modified since the invalidation are actually used during the current lifetime. Case 9 is a straightforward capacity (or conflict) miss since the block was previously replaced from the cache and the words in the block have not been modified since last accessed. All of the other cases refer to misses that occur due to a combination of factors. For example, cases 4 and 5 are cold misses because this processor has never accessed the block before; however, some other processor had written the block, so there is also

6. In this classification, we do not distinguish conflict from capacity misses since both are a result of the available resources (set or entire cache) becoming full and the difference between them does not shed additional light on multiprocessor issues.

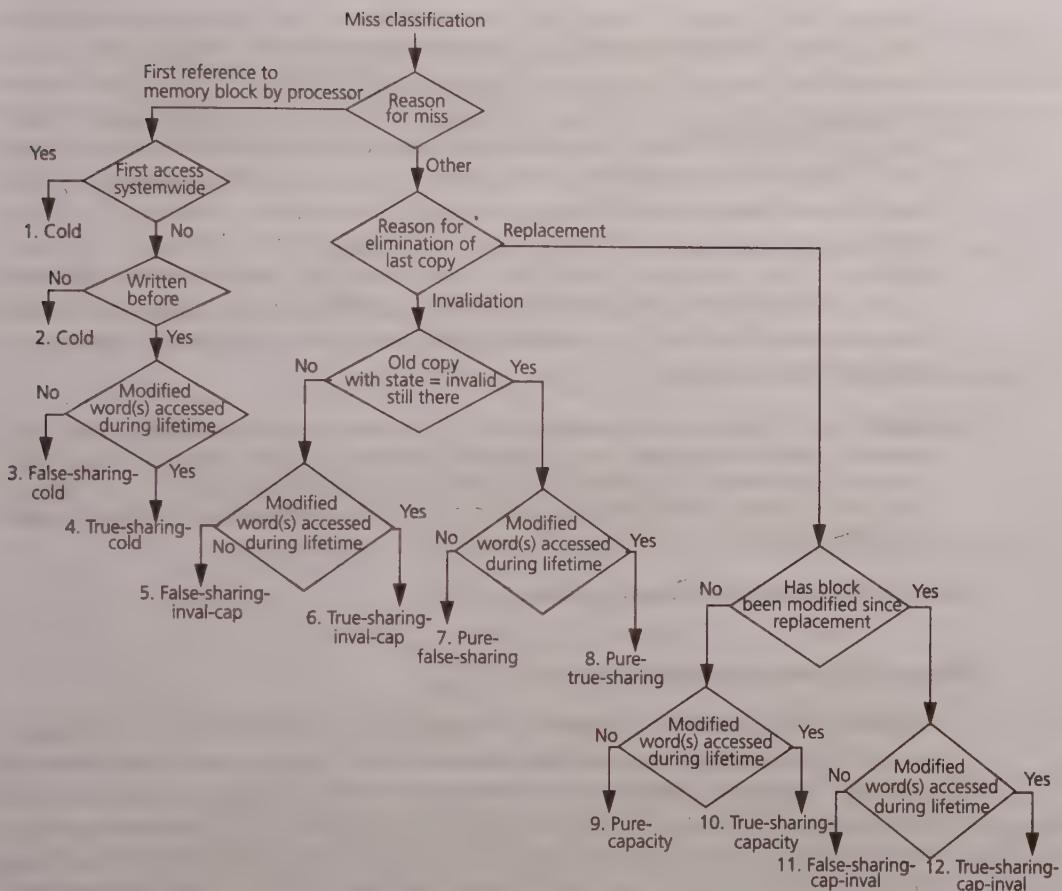


FIGURE 5.20 A classification of cache misses for shared memory multiprocessors. The four basic categories of cache misses in this classification are cold, capacity, true sharing, and false sharing misses (conflict misses are considered to be capacity misses for this purpose). Many mixed categories arise because there may be multiple causes for a miss. For example, a block may be first replaced from processor A's cache, then written to by processor B, and then read back by processor A, making it a capacity-cum-validation false/true sharing miss. This would be labeled "false/true sharing cap-inval" in the classification since sharing takes priority and since the replacement happened before the invalidation (cases 11 and 12 in the figure). If the block were first invalidated in A's cache, then the invalid block replaced, and then read again by A, it would be labeled "false/true sharing inval-cap" (cases 6 and 7). In terms of the four major categories, these misses all fall into true or false sharing misses, as appropriate. Note: the question "modified word(s) accessed during lifetime?" asks whether accesses are made by this processor in the current lifetime to word(s) within the cache block that have been modified since the last "essential coherence" miss to this block by this processor, where essential coherence misses correspond to categories 4, 6, 8, 10, and 12. This can only be determined when the current lifetime of the block ends.

sharing (false or true). Similarly, we can have false or true sharing on blocks that were previously replaced due to capacity or conflicts. Solving only one of the problems in these cases may not necessarily eliminate such misses. For example, if a miss occurs due to both false sharing and capacity problems, then eliminating the false sharing problem by reducing block size will likely not eliminate that miss. On the other hand, sharing misses are in a sense more fundamental than capacity misses since they will remain even if the size of cache is increased to infinity, so we give them priority in the classification of multiple-cause misses. All misses with true sharing in their names in the resulting classification are called *essential coherence misses*. They would occur even with infinite caches, single-word blocks, and all data preloaded into all caches (i.e., no cold misses). Example 5.11 illustrates these definitions of miss categories.

EXAMPLE 5.11 Suppose three processors, P_1 , P_2 , and P_3 , issue the memory operations shown in the first few columns of Table 5.4 (the first column indicates virtual time or steps). Use the miss classification algorithm to classify the misses in the last column. Assume that each processor's cache consists of only a single four-word cache block and that all the caches are initially empty.

Answer The results are shown in Table 5.4. ■

Impact of Block Size on Miss Rate

Applying the classification algorithm of Figure 5.20 to simulated runs of a workload, we can determine how frequently the various kinds of misses occur in programs and how the frequencies change with variations in cache organization, such as block size. Figure 5.21 shows the decomposition of the misses for the example applications running on 16 processors, with 1-MB four-way set-associative caches each, as the cache block size is varied from 8 bytes to 256 bytes. The bars show the four basic types of misses: cold misses (cases 1 and 2), capacity—including conflict—misses (case 11), true sharing misses, (cases 4, 6, 8, 10, 12), and false sharing misses (cases 3, 5, 7, and 11). In addition, they show the frequency of *upgrades*—writes that find the block in the cache but in the shared state. Upgrades are different from the other types of misses since the cache already has the valid data and only needs exclusive ownership. While they are not included in the classification scheme of Figure 5.20, they are still usually considered to be misses since they generate traffic on the interconnect and can stall the processor.

For each individual application, the miss characteristics change with block size much as we would expect from our understanding of the program and the miss categories. Cold, capacity, and true sharing misses tend to decrease with increasing block size because the additional data brought in with each miss is accessed before the block is replaced, due to spatial locality. However, false sharing misses tend to increase with block size. In all cases, true sharing is a significant fraction of the misses, so even with ideal, infinite caches, the miss rate and bus bandwidth will not go to zero. However, the overall characteristics differ widely across programs. For example, the size of the true sharing component varies significantly. Some applica-

Table 5.4 Classifying Misses in an Example Reference Stream from Three Processors

Time	P ₁	P ₂	P ₃	Miss Classification
1	ld w0		ld w2	P ₁ and P ₃ miss; but we will classify later on replace/invalid
2			st w2	P _{1,1} : pure cold miss; P _{3,2} : upgrade
3		ld w1		P ₂ misses, but we will classify later on replace/invalid
4		ld w2	ld w7	P ₂ hits; P ₃ misses; P _{3,1} : cold miss
5	ld w5			P ₁ misses
6		ld w6		P ₂ misses; P _{2,3} : cold true sharing miss (w2 accessed)
7		st w6		P _{1,5} : cold miss; P _{2,7} : upgrade; P _{3,4} : pure cold miss
8	ld w5			P ₁ misses
9	ld w6		ld w2	P ₁ hits; P ₃ misses
10	ld w2	ld w1		P _{1,10} : pure true share miss; P _{2,6} : cold miss
11	st w5			P ₁ misses; P _{1,10} : pure true sharing miss
12			st w2	P _{2,10} : capacity miss; P _{3,11} : upgrade
13			ld w7	P ₃ misses; P _{3,9} : capacity miss
14			ld w2	P ₃ misses; P _{3,13} : inval cap false sharing miss
15	ld w0			P ₁ misses; P _{1,11} : capacity miss

If multiple references are listed in the same row, we assume that P₁ issues before P₂ and P₂ issues before P₃. The notation ld/st wi refers to load/store of word i. W1 through w4 are on the same cache block, and so on. The notation P_{i,j} points to the memory reference issued by processor i at row j.

tions show a substantial increase in false sharing with block size, whereas others show almost none. Furthermore, the figure shows data only for the default data sets. In practice it is very important to examine the results as the input data set size and number of processors are scaled before drawing conclusions about the false sharing or spatial locality of an application (see Chapter 4). Let us investigate the properties of the applications that give rise to differences in miss characteristics observed at the machine level and that allow us to understand scaling qualitatively.

Relation to Application Structure

Multiword cache blocks exploit spatial locality by prefetching data surrounding the accessed address. Of course, beyond a point, larger cache blocks can hurt performance by (1) prefetching unneeded data, (2) causing increased conflict misses as the number of distinct blocks that can be stored in a finite cache decreases with increasing block size, and (3) causing increased false sharing misses. Spatial locality in parallel programs tends to be lower than in sequential programs because, when a

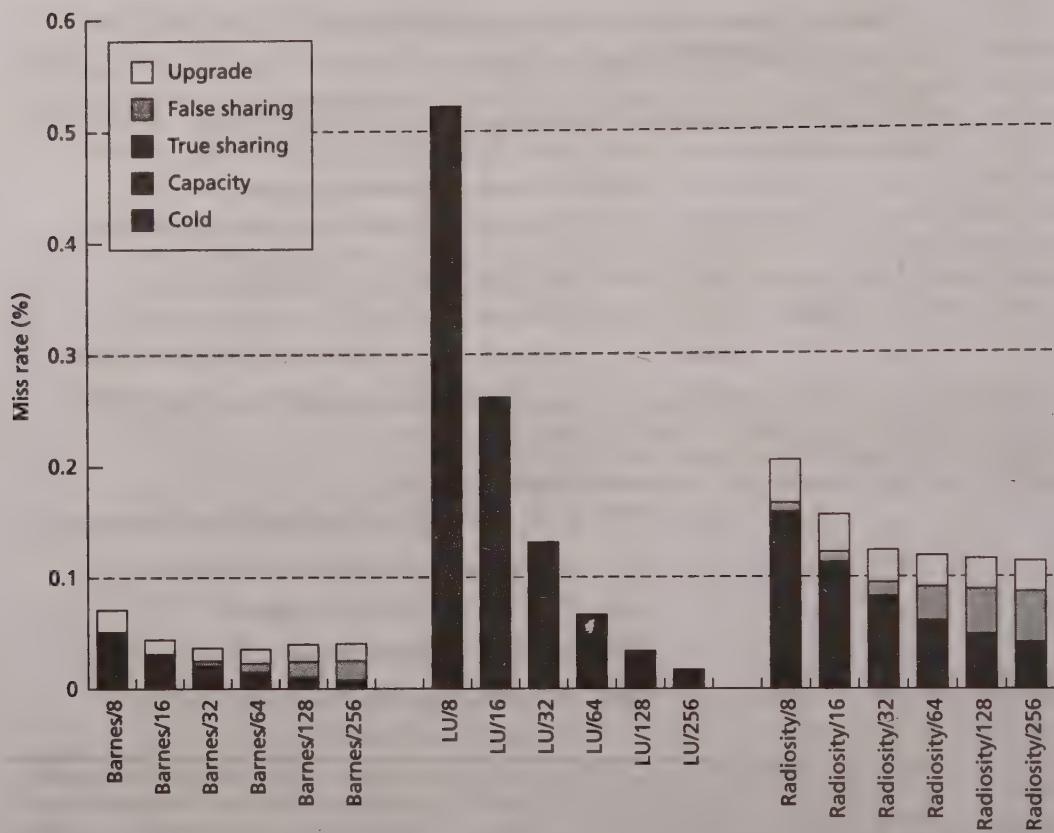


FIGURE 5.21(a) Breakdown of application miss rates as a function of cache block size for 1-MB caches per processor for Barnes-Hut, LU, and Radiosity applications. Conflict misses are included in capacity misses. The breakdown and behavior of misses vary greatly across applications, but we can observe some common trends. Cold misses and capacity misses tend to decrease quite quickly with block size as a result of spatial locality. True sharing misses also tend to decrease, whereas false sharing misses increase. While the false sharing component is usually small for small block sizes, it sometimes remains small and sometimes increases very quickly. Upgrades are shown at the top of the bars and without shading, so they can be ignored if desired.

memory block is brought into the cache, some of the data therein may belong to another processor and will not be used by the processor performing the miss. As an extreme example, some parallel programs assign adjacent elements of an array to different processors in order to ensure good load balance and in the process substantially decrease the spatial locality of the program.

The data in Figure 5.21 shows that LU and Ocean have good spatial locality and little false sharing even in the parallel case. The miss rates for many components drop proportionately to increases in cache block size, and false sharing misses are essentially nonexistent. This is in large part because these array-based codes use

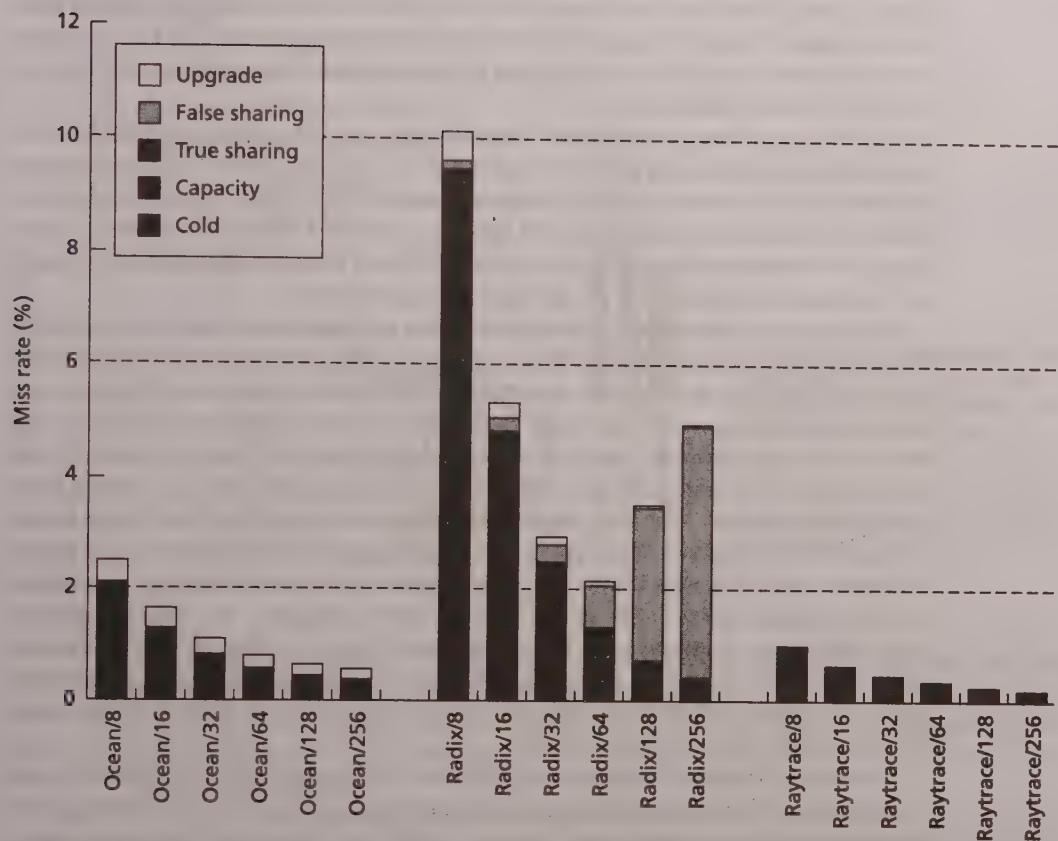


FIGURE 5.21(b) Breakdown of application miss rates as a function of cache block size for 1-MB caches per processor for Ocean, Radix, and Raytrace applications.

architecturally aware data structures, as discussed in Chapters 3 and 4. For example, a grid in Ocean is not represented as a single 2D array (which can introduce substantial false sharing at column-oriented partition boundaries) but as a 4D array: a 2D array of blocks, each of which is itself a 2D array. Such structuring, by programmers or compilers, ensures that most accesses are unit stride and over substantial, contiguous blocks of data, thus the nice behavior.

In Ocean, capacity misses are significant, but they are to the interior elements of a process's partition, so they have very good spatial locality. One difference with LU is that true sharing misses in Ocean do not exhibit such good spatial locality. Most of the true sharing misses are to elements at the borders of neighboring partitions. These exhibit good spatial locality at row-oriented borders where the data to be fetched is contiguous in the address space. However, when a processor accesses an element at a column-oriented border, it fetches an entire cache block of interior elements of its neighbor's partition, which it will not use and therefore wastes. Since

capacity misses are not very large with this problem and machine configuration, overall spatial locality is limited by that of true communication. In LU, even true communication is of B -by- B contiguous blocks at a time, so spatial locality is excellent even on true sharing misses.

As for scaling, the spatial locality for these two applications is expected to remain good with no false sharing as both the problem size and the number of processors are increased (at least until partitions become unrealistically small). This should be true even for cache blocks larger than 256 bytes, at least for LU. In Ocean, how capacity versus true communication misses (and hence spatial locality) scale depends strongly on the relative scaling of data set size and processor count.

The graphics application Raytrace also shows negligible false sharing but displays somewhat worse spatial locality. False sharing is small because the main data structure (the collection of polygons constituting the scene) is read-only. The only read-write sharing happens on the image plane data structure and the task queues, but that is well controlled and small for large enough problems. This true sharing miss rate is reduced by increasing cache block size. The reason for the poor spatial locality of capacity misses (although the overall magnitude is small in this configuration) is that the access pattern to the collection of polygons is quite arbitrary since the set of objects that a ray will bounce off of is unpredictable. As for scaling, as problem size is increased (most likely in the form of more polygons), the primary effect is likely to be larger capacity miss rates; the spatial locality within individual components should not change. A larger number of processors is in many ways similar to having a smaller problem size, except that we may see more sharing in the image plane and task queue data structures.

The Barnes-Hut and Radiosity applications show moderate spatial locality and false sharing. These applications employ complex data structures, including trees encoding spatial information and arrays in which the records assigned to each processor are not contiguous in memory. For example, Barnes-Hut operates on particle records stored in an array. As the application proceeds and particles move in physical space, particle records get reassigned to different processors, with the result that after some time adjacent particles in the array most likely belong to different processors. Spatial locality is exploited well within a particle record but not very well across records. False sharing becomes a problem at large block sizes for different reasons. First, different processors may write to different records that share a cache block. Second, a particle data structure (record) contains both fields that are being modified by the owner of that particle in a phase (e.g., the current force on this particle in the force calculation phase) and fields that are read by other processors and are not being modified in this phase (e.g., the current position of the particle). Since these two fields may fall in the same cache block for large block sizes, false sharing results. It is possible to eliminate such false sharing by splitting the particle data structure according to the access patterns of the fields, but that is not done in this program since the absolute magnitude of the miss rate is small. As problem size and the number of processors are scaled, the miss rate behavior of Barnes-Hut is expected to change little. This is because the working set size changes very slowly (as the log of the number of particles, unlike Ocean and Raytrace), spatial locality is

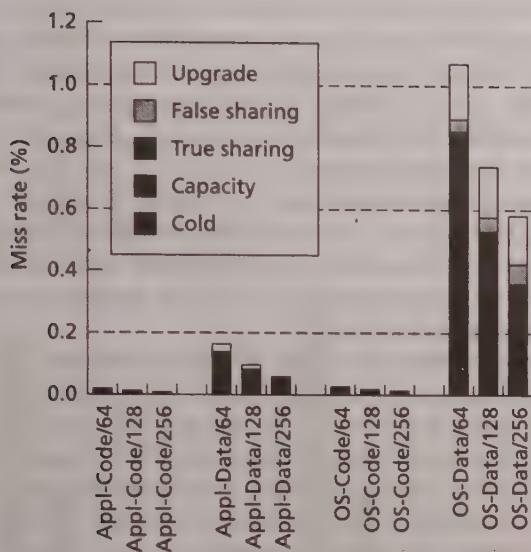


FIGURE 5.22 Breakdown of miss rates for Multiprog as a function of cache block size. The results are for 1-MB caches. Spatial locality for true sharing misses is much better for the applications than for the operating system.

determined by the size of one particle record and thus remains the same, and the sources of false sharing are not very sensitive to the number of processors. Radiosity is a much more complex application whose behavior is difficult to reason about with larger data sets or more processors; the only option is to gather empirical data showing the growth trends.

The poorest sharing behavior is exhibited by Radix, which not only has a very high miss rate even with 1-MB caches (due to cold and true sharing misses) but which gets significantly worse due to false sharing misses for block sizes of 128 bytes or more. The effect of false sharing in Radix was illustrated in Chapter 4. Let us now examine how it is governed. Consider sorting 256-K keys, using a radix of 1,024 and 16 processors. On average, this results in 16 keys per radix per processor (64 bytes of data), which are then written to a contiguous portion of a global array at an unpredictable starting point. Adjacent 64-byte chunks in this array are written by different processors. If the cache block size is larger than 64 bytes, the high potential for false sharing is clear. As the problem size is increased we will clearly see much less false sharing. The effect of increasing the number of processors is exactly the opposite. Radix illustrates quite dramatically that it is not sufficient to look at a given problem size and number of processors and, based on that, draw conclusions of whether or not false sharing or spatial locality is a problem. It is very important to understand how the results are dependent on the key parameters chosen in the experiment and how these parameters may vary in reality.

Data for the Multiprog workload for 1-MB caches is shown in Figure 5.22. The data is shown separately for user code, user data, kernel code, and kernel data. For code, there are only cold and capacity misses. Furthermore, we see that the spatial locality in operating system data references is not very good. This is true, to a some-

what lesser extent, for the application data misses as well, because `gcc` (the main application causing misses in `Multiprog`) uses a large number of linked lists, which do not offer good spatial locality. It is interesting that we have an observable fraction of application true sharing misses, although we are running only sequential applications. These misses arise due to process migration and are incurred when a sequential process migrates from one processor to another (a decision made by the operating system for resource management) and then references memory blocks that it wrote while it was executing on the other processor. While the spatial locality in cold and capacity misses is quite reasonable, the true sharing misses do not decrease at all for kernel data. One reason for this may be that the operating system has not been well structured as a parallel program.

Finally, let us examine the behavior of `Ocean`, `Radix`, and `Raytrace` for smaller 64-KB caches. The miss rate results are shown in Figure 5.23. As expected, the overall miss rates are higher, and capacity misses have increased substantially. The effects of cache block size for true sharing and false sharing misses are not significantly different from the results for 1-MB caches because these properties are quite fundamental to the assignment and orchestration used by a program and are not too sensitive to cache size. However, the behavior of capacity misses has a much larger effect on the behavior of the overall miss rate. For example, in `Ocean`, capacity misses now dominate sharing misses; since they have much better spatial locality, the overall miss rate decreases much more quickly with increasing block size than it did with 1-MB caches. (Very large blocks in a small cache can have the problem that blocks may be replaced from the cache due to conflicts before the processor has had a chance to reference all of the words in them.) In `Raytrace`, capacity misses have somewhat worse spatial locality than true sharing misses, so the overall benefits of large blocks look worse with smaller caches. Results for false sharing and spatial locality for other applications can be found in the literature (Torrellas, Lam, and Hennessy 1994; Jermiassen and Eggers 1991; Woo et al. 1995).

While larger cache blocks reduce the miss rate for most of our applications, within the range of block sizes we consider they have two important potential disadvantages. First, they can increase the cost of each miss since more data has to be transferred across the bus (although techniques like only waiting for the referenced word to arrive before allowing the processor to proceed, called a *critical word restart* approach, can alleviate this). Second, they increase traffic, and hence contention, if the whole block is not useful.

Impact of Block Size on Bus Traffic

Let us briefly examine the impact of cache block size on bus traffic rather than miss rate. While the number of misses and total traffic generated are clearly related, their impact on observed performance can be quite different. Misses have a cost that may contribute directly to performance, even though modern microprocessors try hard to hide the latency of misses by overlapping it with other activities. Traffic, on the

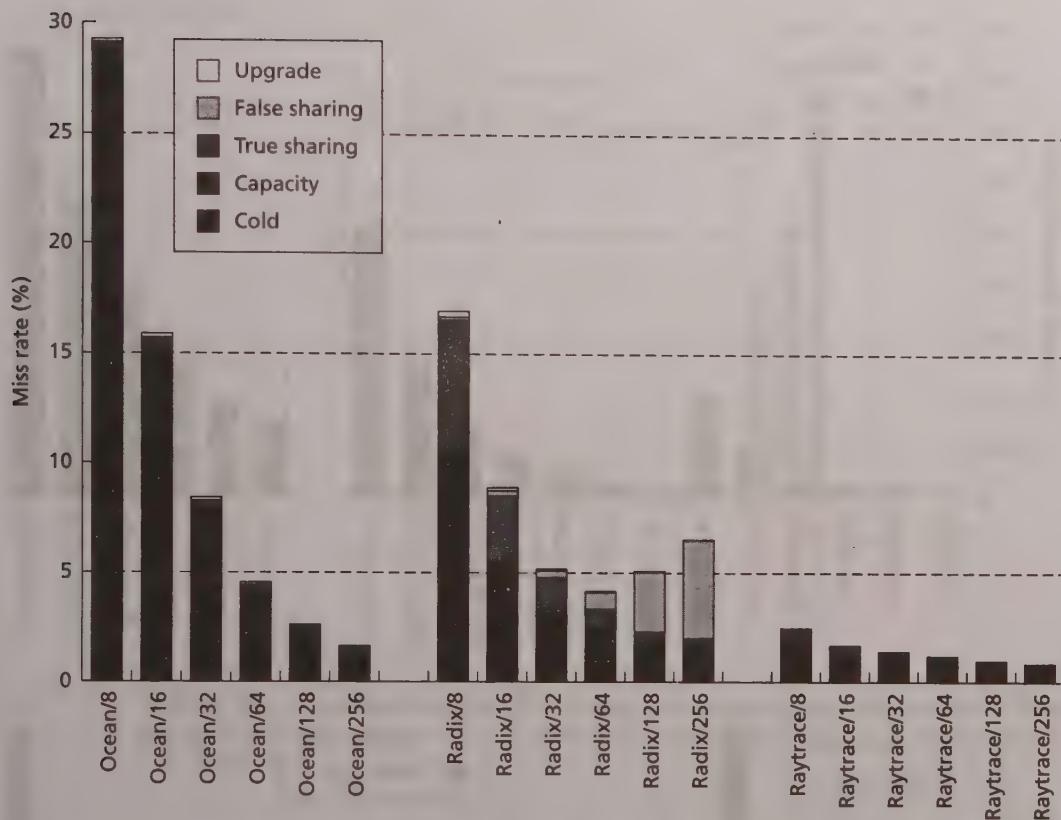


FIGURE 5.23 Breakdown of application miss rates as a function of cache block size for 64-KB caches. Capacity misses are now a much larger fraction of the overall miss rate. Capacity miss rates decrease differently with block size for different applications.

other hand, affects performance indirectly by causing contention and hence increasing the cost of other misses. For example, if an application program's misses are reduced significantly by increasing the cache block size, but the bus traffic is increased by 50%, this might be a reasonable trade-off if the application was originally using only 10% of the available bus and memory bandwidth. Increasing the bus and memory utilization to 15% is unlikely to increase the miss latencies significantly. However, if the application was originally using 75% of the bus and memory bandwidth, then increasing the block size is probably a bad idea.

Figure 5.24 shows the total bus traffic for our applications in bytes/instruction or bytes/FLOP as the cache block size is varied. Three key points can be observed from this graph. First, traffic behaves very differently than miss rate. Only LU shows monotonically decreasing total traffic for the block sizes used. Most other applications see a doubling or tripling of traffic as block size becomes large. Second, the

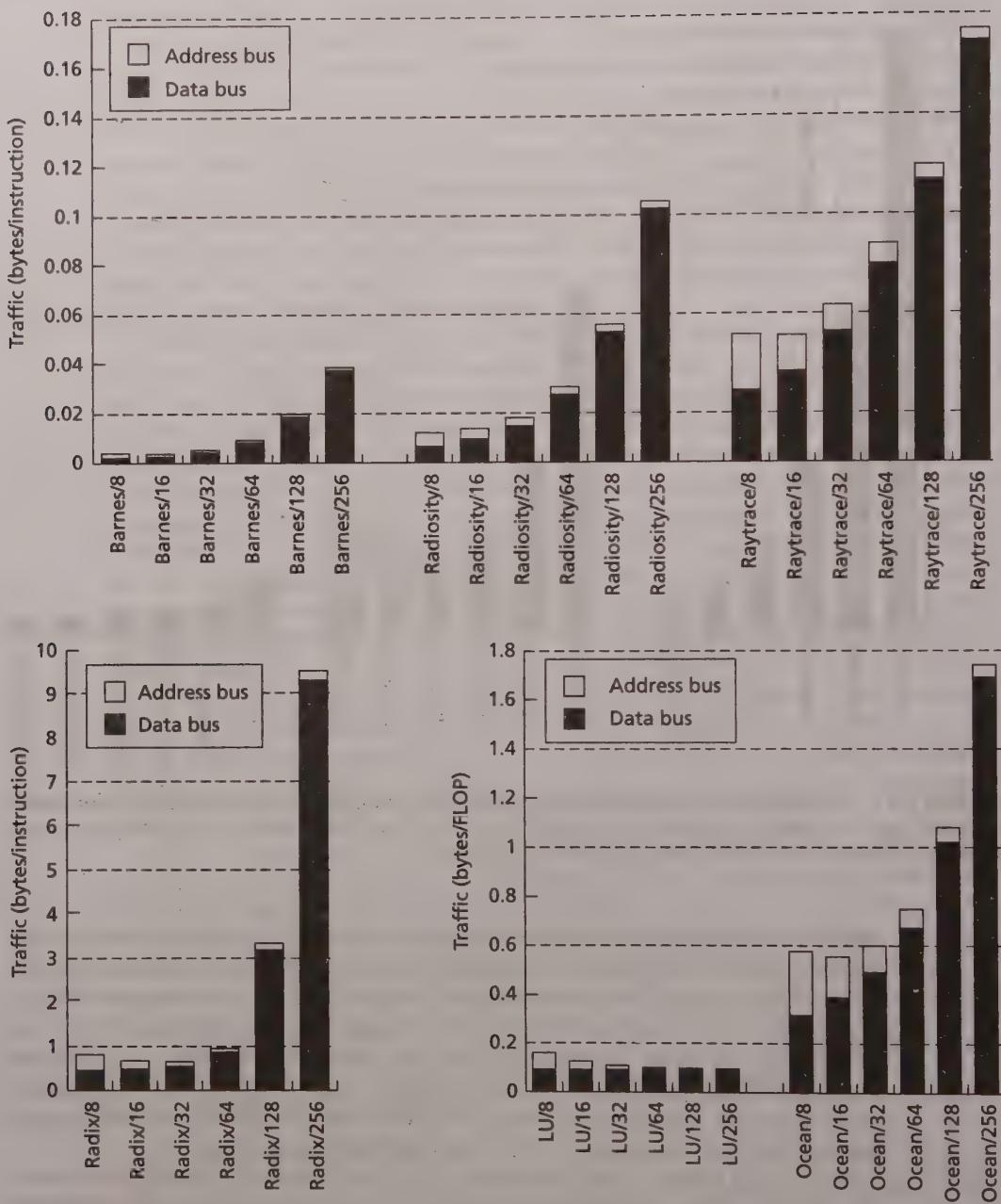


FIGURE 5.24 Traffic (in bytes/instruction or bytes/FLOP) as a function of cache block size with 1-MB caches per processor. Data traffic increases quite quickly with block size when communication misses dominate, except for applications like LU that have excellent spatial locality on all types of misses. Address (including command) bus traffic tends to decrease with block size since the miss rate and, hence, number of blocks transferred decrease.

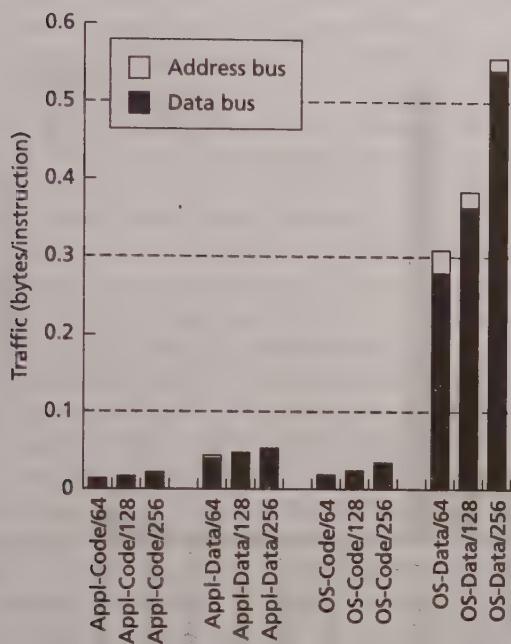


FIGURE 5.25 Traffic in bytes/instruction as a function of cache block size for Multiprog with 1-MB caches. Traffic increases quickly with block size for data references from the OS kernel.

overall traffic requirements for the applications are still small, even for 256-byte block sizes, with the exception of Radix. Radix's large bandwidth requirements (approximately 650 MB/s per processor for 128-byte cache blocks, assuming a sustained 200-MIPS processor) reflect its false sharing problems at large block sizes. Third, the constant address and command traffic overhead for each bus transaction or miss comprises a significant fraction of total traffic for small block sizes. Hence, although actual application data traffic usually increases as we increase the block size due to poor spatial locality, the total traffic is often minimized at 16–32 bytes rather than 8 bytes due to the amortization of the overhead with improved miss rates.

Figure 5.25 shows the traffic data for Multiprog. While the increase in traffic from 64-byte cache blocks to 128-byte blocks is small, the jump at 256-byte blocks is much more substantial (primarily due to kernel data references). Finally, Figure 5.26 shows the traffic results for 64-KB caches for the three relevant applications. For Ocean, even 64- and 128-byte cache blocks don't look so bad, due to the dominance of capacity misses that have good spatial locality.

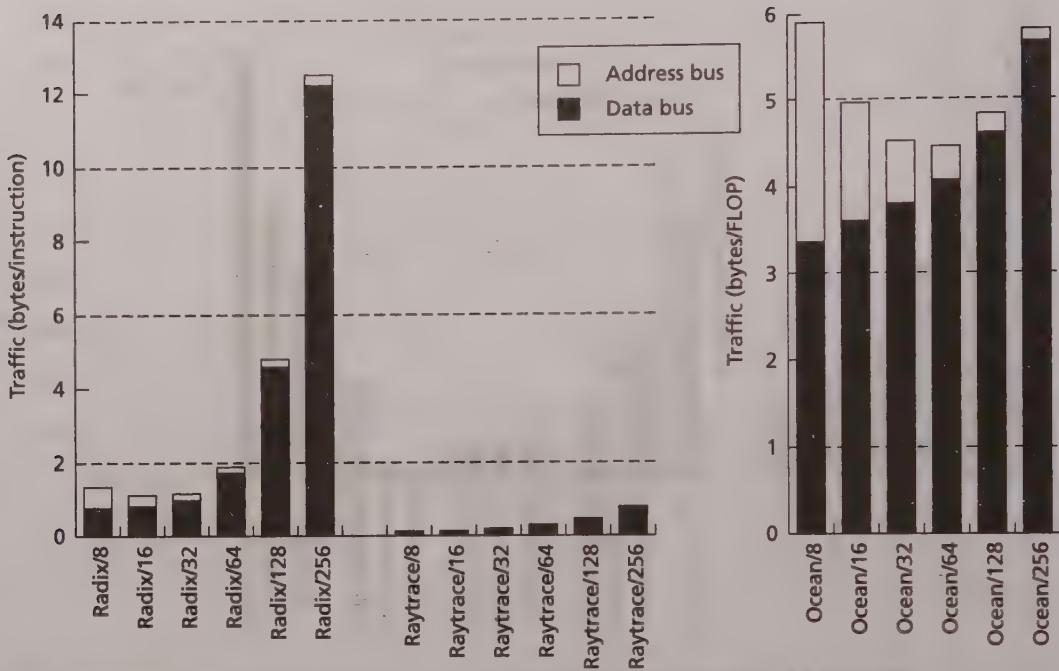


FIGURE 5.26 Traffic (in bytes/instruction or bytes/FLOP) as a function of cache block size with 64-KB caches per processor. Traffic increases more slowly now for Ocean than with 1-MB caches since the capacity misses that now dominate exhibit excellent spatial locality (traversal of a process's assigned subgrid). However, traffic in Radix increases quickly once the threshold block size that causes false sharing is exceeded.

Alleviating the Drawbacks of Large Cache Blocks

The trend toward larger cache block sizes is driven by the increasing gap between processor performance and memory access time. The larger block size amortizes the cost of the bus transaction and memory access across a greater amount of data. The increasing density of processor and memory chips makes it possible to employ large first-level and second-level caches so that the prefetching of data obtained through a larger block size dominates the small increase in conflict misses. However, this trend may bode poorly for multiprocessor designs because false sharing becomes a larger problem. Fortunately, hardware and software mechanisms can be employed to counter the effects of large block size.

Software techniques to reduce false sharing and improve locality on coherence misses are discussed in detail later in the chapter. They essentially involve organizing data structures or work assignments so that data accessed by different processes is not interleaved finely in the shared address space. One example is the use of higher-dimensional arrays so blocks or partitions are wholly contiguous. Compiler

techniques have also been developed to automate some methods of laying out data to reduce false sharing (Jeremiassen and Eggers 1991).

Since false sharing is caused by a large granularity of coherence, the way to reduce it while still exploiting spatial locality is to use large blocks for data transfer but a smaller unit of coherence. A natural hardware mechanism is the use of sub-blocks. Each cache block has a single address tag but distinct state bits for each of several subblocks. One subblock may be valid while others are invalid or dirty. This technique is used in many uniprocessor systems to reduce the amount of data that is copied back to memory on a replacement or to reduce the memory access time on a read miss by resuming the processor when the accessed subblock is present (critical word restart). To avoid false sharing, a write by one processor may invalidate the subblock in another processor's cache while leaving the other subblocks valid. Alternatively, small cache blocks can be used, but on a miss the system can prefetch blocks beyond the accessed block. Proposals have also been made for caches with adjustable block sizes (Dubnicki and LeBlanc 1992). The disadvantage of these approaches is increased state and complexity beyond a commodity cache design.

A more subtle hardware technique is to delay propagating or applying invalidations from a processor until it has issued multiple writes. Delaying invalidations and performing them all at once reduces the occurrence of intervening read misses to those blocks. However, this sort of technique can change the memory consistency model in subtle ways, so further discussion is deferred until Chapter 9 where we consider weaker consistency models in the context of scalable machines. Another hardware technique to reduce false sharing is the use of update- rather than invalidation-based protocols.

5.4.5 Update-Based versus Invalidiation-Based Protocols

Whether writes should cause other cached copies to be updated or invalidated has been the subject of considerable debate. Various vendors have taken different stands and, in fact, have changed their position from one design to the next. The controversy arises because the relative performance of update-based versus invalidation-based protocols depends strongly on the sharing patterns exhibited by the workload and on the cost of various underlying operations. Intuitively, if the processors that were using the data before it was updated (written) are likely to want to see the new values in the future, updates should perform better than invalidations. However, if the processors holding the old data are never going to use it again, the update traffic is useless and just consumes interconnect and controller resources. Invalidations would clean out the old copies and eliminate the apparent sharing. This "pack rat" phenomenon with update protocols is especially irritating under multiprogrammed use of a machine, when sequential processes migrate from processor to processor under OS control so that useless updates are performed in caches of processors that are no longer running that process. It is easy to construct cases in which either scheme does substantially better than the other, as illustrated by Example 5.12.

EXAMPLE 5.12 Consider the following two program reference patterns:

- *Pattern 1:* Repeat k times; processor 1 writes a new value into variable V and processors 2 through P read the value of V . This represents a one-producer-many-consumer scenario that may arise, for example, when processors are accessing a highly contended flag for one-to-many event synchronization.
- *Pattern 2:* Repeat k times; processor 1 writes M times to variable V and then processor 2 reads the value of V . This represents a sharing pattern that may occur between pairs of processors, where the first successfully computes and accumulates values into a variable and then when the accumulation is complete, another processor reads the value.

What is the relative cost for update- and invalidation-based protocols in terms of the number of cache misses and bus traffic? Assume that an invalidation/upgrade transaction consumes 6 bytes (5 bytes for address plus 1 byte for command), an update takes 14 bytes (6 bytes for address and command and 8 bytes of data for the updated word), and a regular cache miss takes 70 bytes (6 bytes for address and command plus 64 bytes of data corresponding to cache block size). Also assume that $P = 16$, $M = 10$, $k = 10$, and that all caches initially are empty.

Answer With an update scheme in pattern 1, the first iteration on all P processors will incur a regular cache miss (including processor 1 when it writes) plus an update due to the write. In subsequent $k - 1$ iterations, no more misses will occur and only one update per iteration will be generated. Thus, overall we will see misses = $P = 16$; traffic = $P \times \text{RdMiss} + (k - 1) \times \text{Update} = 16 \times 70 + 10 \times 14 = 1,260$ bytes.

With an invalidate scheme, all P processors will incur a regular cache miss in the first iteration. In subsequent $k - 1$ iterations, processor 1 will generate an upgrade, but all others will experience a read miss. Thus, counting upgrades as misses, overall we will see misses = $P + (k - 1) \times P = 16 + 9 \times 16 = 160$, of which 151 are read misses and 9 are upgrades; traffic = read misses \times RdMiss + $(k - 1) \times$ Upgrade = $151 \times 70 + 9 \times 6 = 10,624$ bytes.

With an update scheme on pattern 2, the first iteration will incur two regular cache misses, one for processor 1 and the other for processor 2. In subsequent $k - 1$ iterations, no more misses will be generated, but M updates will be generated in each iteration. Thus, overall we will see misses = 2; traffic = $2 \times \text{RdMiss} + M \times (k - 1) \times \text{Update} = 2 \times 70 + 10 \times 9 \times 14 = 1,400$ bytes.

With an invalidate scheme, two regular cache misses will occur in the first iteration. In subsequent $k - 1$ iterations, one upgrade (for the first write only) plus one regular read miss will be generated in each iteration. Thus, counting upgrades as misses, overall we will see misses = $2 + (k - 1) \times 2 = 2 + 9 = 11$; traffic = misses \times RdMiss + $(k - 1) \times$ Upgrade = $11 \times 70 + 9 \times 6 = 824$ bytes. ■

These example patterns suggest that it might be possible to design schemes that capture the advantages of both update and invalidate protocols. The success of such schemes will depend on their costs and on the sharing patterns for real parallel programs and workloads. Let us briefly explore the design options and then employ workload-driven evaluation.

Combining Update- and Invalidation-Based Protocols

One way to take advantage of both update and invalidate protocols is to support both in hardware and to decide dynamically at page granularity whether coherence

for a given page is to be maintained using an update or an invalidate protocol. The decision about the choice of protocol can be indicated by making a system call. The main advantage of such schemes is that they are relatively easy to support; they utilize the TLB to indicate to the rest of the coherence subsystem which of the two protocols to use. The main disadvantage of such schemes is the burden they put on the programmer to choose protocols for pages or data structures. The decision task is also made difficult because of the coarse granularity at which control is made available; data structures that desire different protocols may fall on the same page.

An alternative is to choose the protocol at a cache block granularity, by observing the sharing behavior at run time. Ideally, for each write, we would like to be able to peer into the future references that will be made to that cache block by all processors and then decide whether to invalidate other copies or to do an update. Since this information is obviously not available, and since there are substantial perturbations due to cache replacements and false sharing, a more practical scheme is needed.

So-called competitive schemes change the protocol for a block between invalidate and update in hardware based on observed patterns at run time. The key attribute of such schemes is that if a wrong decision is made once for a cache block, the losses due to that wrong decision should be kept bounded and small (Karlin et al. 1986). For instance, if a block is currently using update mode, it should not remain in that mode if one processor is continuously writing to it but none of the other processors are reading values from it.

One class of schemes that has been proposed to bound the losses of update protocols works as follows (Grahn, Stenstrom, and Dubois 1995). Starting with the base Dragon update protocol described in Section 5.3.3, associate a countdown counter with each block. Whenever a cache block is accessed by the local processor, the counter value for that block is reset to a threshold value k . Every time an update is received for a block, the counter is decremented. If the counter goes to zero, the block is locally invalidated. The consequence of the local invalidations is that the next time an update is generated on the bus, it may find that no other cache has a valid copy; in that case, that block will switch to the modified state (as per the Dragon protocol) and will stop generating updates. If some other processor now accesses that block, the block will again switch to shared state and this mixed protocol will again start generating updates.

A related approach implemented in the Sun SparcCenter 2000 is to selectively invalidate rather than update with some probability that is a parameter set when configuring the machine (Catanzaro 1997). Other mixed approaches may also be used. For example, one approach uses an invalidation-based protocol for first-level caches and, by default, an update-based protocol for second-level caches. However, if the L_2 cache receives a second update for the block while the block in the L_1 cache is still invalid, then the block is invalidated in the L_2 cache as well. When the block is thus invalidated in all other L_2 caches, writes to the block no longer cause updates.

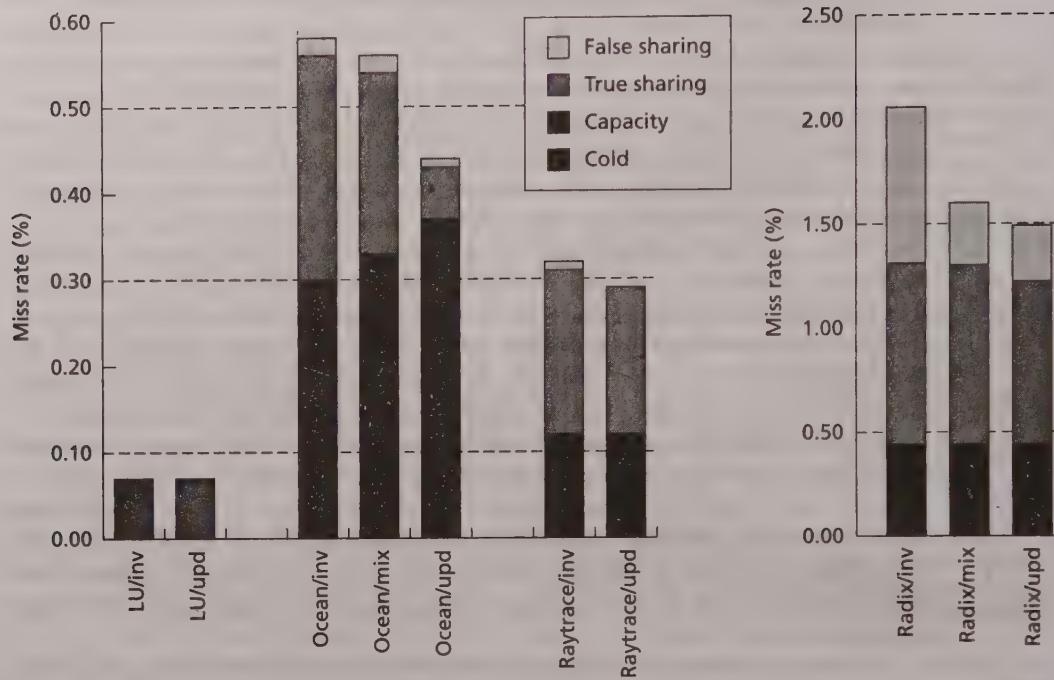


FIGURE 5.27 Miss rates and their decomposition for invalidate, update, and hybrid protocols. The data assumes 1-MB caches, 64-byte cache blocks, four-way set associativity, and threshold $k = 4$ for hybrid protocol.

Workload-Driven Evaluation

To assess the trade-offs among invalidate, update, and the mixed protocols just described, Figure 5.27 shows the miss rates by category for four applications using the default 1-MB four-way set-associative caches with a 64-byte block size. The mixed protocol used is the threshold-based scheme just described. We see that for applications with significant capacity miss rates, the misses sometimes increase with an update protocol. This makes sense because the protocol (with LRU replacement in a set) keeps data in processor caches that would have been removed by an invalidation protocol. For applications with significant true sharing or false sharing miss rates, these categories decrease with an update protocol: after a write update, the other caches holding the blocks can access them without a miss. Overall, the update protocol appears to be advantageous for the sum of these three categories and the mixed protocol falls in between. The category that is not shown in this figure, however, is the upgrade or update operations for these protocols. This data is presented in Figure 5.28. Note that the scale of the graphs has changed because update operations are roughly four times more prevalent than misses. It is useful to separate these operations from other misses because the way they are handled in the machine is

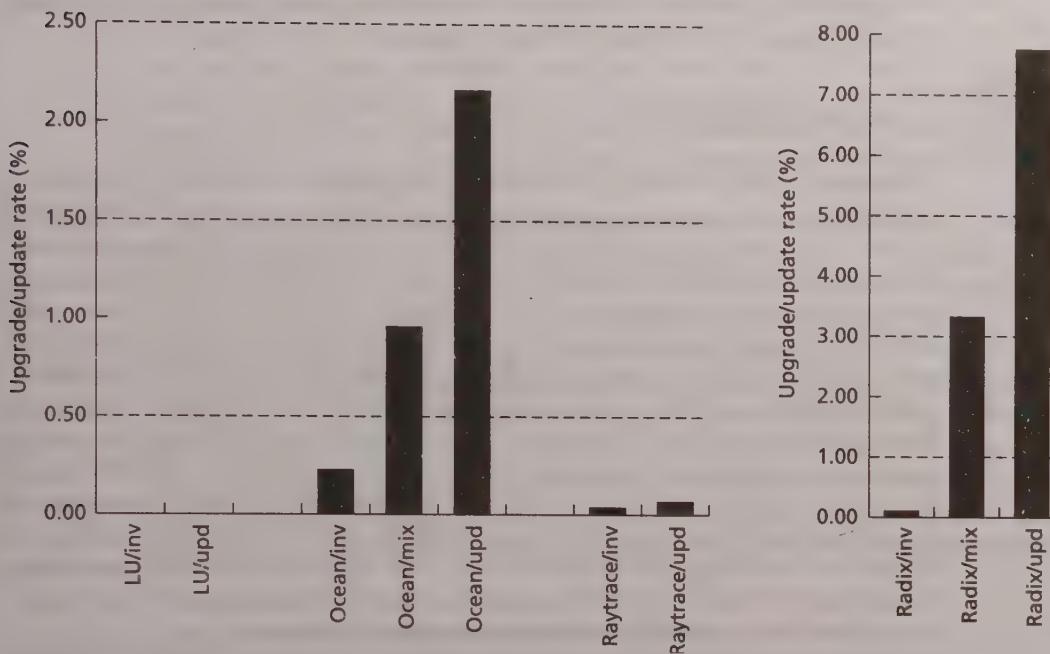


FIGURE 5.28 Upgrade and update rates for invalidate, update, and mixed protocols. The data assumes 1-MB caches, 64-byte cache blocks, four-way set associativity, and threshold $k = 4$ for hybrid protocol. Rates are measured relative to total memory references.

likely to be different. Updates are a single-word write rather than a full cache block transfer. Because the data is being pushed from where it is being produced, it may arrive at the consumer before it is needed. Even for the producer, the latency of update and upgrade operations may be less critical than that of misses since it is quite easily hidden from the processor's critical path (see Chapter 11).

Unfortunately, the traffic associated with updates is quite substantial. In large part, this occurs because multiple writes are made by a processor to the same block before a read, all generating updates. With the invalidate protocol, the first of these writes may cause an invalidation, but the rest can simply accumulate locally in the block and be transferred in one bus transaction on a flush or a write back (see Example 5.12). The increased traffic causes contention and can greatly increase the cost of misses. Sophisticated update schemes might attempt to delay the update to achieve a similar effect (by merging writes in the write buffer) or use other techniques to reduce traffic and improve performance (Dahlgren 1995). However, the increased bandwidth demand, the complexity of supporting updates, the trend toward larger cache blocks, and the pack rat phenomenon with the important case of multiprogrammed sequential workloads underlie the trend away from update-based protocols in the industry. We see in Chapter 8 that update protocols also have some other problems for scalable cache-coherent architectures, making it less attractive for microprocessors to support these protocols.

Having discussed how to keep data coherent, let us now consider how synchronization is managed in bus-based multiprocessors.

5.5 SYNCHRONIZATION

A critical interplay of hardware and software in multiprocessors arises in supporting synchronization operations: mutual exclusion, point-to-point events, and global events. There has been considerable debate over the years about how much hardware support and exactly what hardware primitives should be provided to support these synchronization operations. The conclusions have changed from time to time with changes in technology and design style. Hardware support has the advantage of speed, but moving functionality to software has the advantages of cost, flexibility, and adaptability to different situations. The classic works of Dijkstra (1965) and Knuth (1966) show that it is possible to provide mutual exclusion with only atomic read and write operations (assuming a sequentially consistent memory). However, all practical synchronization methods rely on hardware support for some sort of *atomic read-modify-write* operation, in which the value of a memory location is ensured to be read, modified, and written back atomically without intervening accesses to the location by other processors. Simple or sophisticated synchronization algorithms can be built in software using these primitives.

The history of instruction sets offers a glimpse into the evolving hardware support for synchronization. One of the key instruction set enhancements in the IBM 370 was the inclusion of a sophisticated atomic instruction, the *compare&swap* instruction, to support synchronization in concurrent programming on uniprocessor or multiprocessor systems. The *compare&swap* compares the value in a memory location with the value in a specified register and, if they are equal, swaps the value in the memory location with the value in a second specified register. The Intel x86 allows any instruction to be prefixed with a lock modifier to make it atomic; since the source and destination operands are memory locations, much of the instruction set can be used to implement various atomic operations involving even more than one memory location. Advocates of high-level language architecture have proposed that the user-level synchronization operations, such as locks and barriers, should be supported directly at the machine level, not just atomic read-modify-write primitives; that is, the synchronization “algorithm” itself should be implemented in hardware. This issue became very active during the reduced instruction set debates since the operations that access memory were scaled back to simple loads and stores with only one memory operand. The Sparc approach was to provide atomic operations involving a register or registers and a memory location using a simple swap (atomically swapping the contents of the specified register and memory location) and a *compare&swap*. MIPS left off atomic primitives in the early instruction sets, as did the IBM Power architecture used in the RS6000. The primitive that was eventually incorporated in MIPS was a novel combination of a special load and a conditional store, described later in this section, which allows a variety of higher-level read-modify-write operations to be constructed without requiring the design to implement them all. In essence, the pair of instructions can be used instead of a sin-

gle instruction to implement atomic exchange or more complex atomic operations. This approach was later incorporated into the PowerPC and DEC Alpha architectures and is now quite popular. As we will see, synchronization brings to light a rich family of trade-offs across the layers of communication architecture. Not only can a spectrum of high-level operations and low-level primitives be supported by hardware, but the synchronization requirements of applications vary substantially as well.

The focus of this section is on how synchronization operations can be implemented on a bus-based cache-coherent multiprocessor through a combination of software algorithms and hardware primitives. In particular, it describes the implementation of mutual exclusion through lock-unlock pairs, point-to-point event synchronization through flags, and global event synchronization through barriers. Let us begin by considering the components of a synchronization event. This will make it clear why supporting the high-level mutual exclusion and event operations directly in hardware is difficult and is likely to make the implementation too rigid. Then, given that the hardware supports only the basic atomic operations, we can examine the role of the user software and system software in synchronization operations and then consider the hardware and software design trade-offs in greater detail.

5.5.1 Components of a Synchronization Event

There are three major components of a synchronization event:

1. *Acquire method*: a method by which a process tries to acquire the right to the synchronization (to enter the critical section or proceed past the event synchronization).
2. *Waiting algorithm*: a method by which a process waits for a synchronization to become available; for example, if a process tries to acquire a lock but the lock is not free, or to proceed past an event but the event has not yet occurred.
3. *Release method*: a method for a process to enable other processes to proceed past a synchronization event; for example, an implementation of the *Unlock* operation, a method for the last process arriving at a barrier to release the waiting processes, or a method for notifying a process waiting at a point-to-point event that the event has occurred.

The choice of waiting algorithm is quite independent of the type of synchronization. There are two main choices: busy-waiting and blocking. *Busy-waiting* means that the process spins in a loop that repeatedly tests for a variable to change its value. A release of the synchronization event by another processor changes the value of the variable, allowing the waiting process to proceed. Under *blocking*, the process does not spin but simply blocks (suspends) itself and releases the processor if it finds that it needs to wait. It will be awakened and made ready to run again when the release it was waiting for occurs. The trade-offs between busy-waiting and blocking are clear. Blocking has higher overhead since suspending and resuming a process involves the operating system (and suspending and resuming a thread involves the run-time system of a threads package), but it makes the processor available to other

threads or processes that have useful work to do. Busy-waiting avoids the cost of suspension but consumes the processor and cache bandwidth while waiting. Blocking is strictly more powerful than busy-waiting because, if the process or thread that is being waited upon is not allowed to run, the busy-wait will never end.⁷ Busy-waiting is likely to be better when the waiting period is short, whereas blocking is likely to be a better choice if the waiting period is long and if there are other processes to run. Hybrid waiting methods can be used in which the process busy-waits for a while in case the waiting period is short, and if the waiting period exceeds a certain threshold, the process blocks, allowing other processes to run (a two-phase waiting algorithm).

The difficulty in implementing high-level synchronization operations in hardware is not the acquire or the release component but the waiting algorithm. Thus, it makes sense to provide hardware support for the critical aspects of the acquire and release methods and allow the three components to be glued together in software. However, subtle but very important hardware/software interactions remain in how the spinning operation in the busy-wait component is realized.

5.5.2 Role of the User and System

Who should be responsible for implementing the internals of high-level synchronization operations such as locks and barriers? Typically, a programmer wants to use locks, events, or even higher-level operations without having to worry about their internal implementation. The implementation is left to the system, which must decide how much hardware support to provide and how much of the functionality to implement in software. Software synchronization algorithms using simple atomic exchange primitives have been developed that approach the speed of full hardware implementations, and the flexibility and hardware simplification they afford are very attractive. As with other aspects of system design, the utility of faster operations with more hardware support depends on the frequency of the use of those operations in the applications. So, once again, the best answer will be determined by a better understanding of application behavior.

Software implementations of synchronization constructs are usually included in system libraries. Good synchronization library design can be quite challenging. One potential complication is that the same type of synchronization (lock, barrier), and even the same synchronization variable, may be used at different times under very different run-time conditions. For example, a lock may be accessed with low contention (a small number of processors, maybe only one, trying to acquire the lock at a time) or with high contention (many processors trying to acquire the lock at the same time). The different scenarios impose different performance requirements.

7. This problem of denying resources to the critical process or thread is one that is actually made simpler with more processors. When the processes are time-shared on a single processor, strict busy-waiting without preemption is sure to be a problem. If each process or thread has its own processor, it is guaranteed not to be a problem. Multiprogramming environments on a limited set of processors may fall somewhere in between.

Under high contention, most processes will spend time waiting, and the key requirement of a lock algorithm is that it provide high lock-unlock transfer bandwidth; under low contention, the key goal is to provide low latency for lock acquisition. Different algorithms may satisfy different requirements better, so we must either find a good compromise algorithm or provide different algorithms for each type of synchronization from which a user can choose. If we are lucky, a flexible library can at run time choose the best implementation for the situation at hand. Different synchronization algorithms may also rely on different basic hardware primitives, so some may be better suited to a particular machine than others. Under multiprogramming, process scheduling and other resource interactions can change the synchronization behavior of the processes in a parallel program. A more sophisticated algorithm that addresses multiprogramming effects may provide better performance in practice than a simple algorithm that has lower latency and higher bandwidth in the dedicated case. All of these factors make synchronization a critical point of hardware/software interaction.

5.5.3 Mutual Exclusion

Mutual exclusion (lock-unlock) operations are implemented using a wide range of algorithms. The simple algorithms tend to be fast when there is little contention for the lock but inefficient under high contention, whereas sophisticated algorithms that deal well with contention have a higher cost in the low-contention case. After a brief discussion of hardware locks, this section describes the simplest software algorithms for memory-based locks using atomic exchange instructions. Following this is a discussion of how these simple algorithms can be implemented by using the special load-locked and store-conditional instruction pairs to synthesize atomic exchange, in place of atomic exchange instructions themselves, and what the trade-offs are. Next, we will look at more sophisticated algorithms that can be built using either method of implementing atomic operations.

Hardware Locks

Lock operations can be supported entirely in hardware, although this is not popular on modern bus-based machines. One option that was used on some older machines was to have a set of lock lines on the bus, each used for one lock at a time. The processor holding the lock asserts the line, and processors waiting for the lock wait for it to be released. A priority circuit determines which processor gets the lock next when there are multiple requestors. However, this approach was quite inflexible since only a limited number of locks can be in use at a time and the waiting algorithm is fixed (typically a form of busy-wait with abort after time-out). Usually, these hardware locks were used only by the operating system for specific purposes, one of which was to implement a larger set of software locks in memory. The CRAY Xmp provided an interesting variant of this approach. A set of registers was shared among

the processors, including a fixed collection of lock registers. Although the architecture made it possible to assign lock registers to user processes, with only a small set of such registers it was awkward to do so in a general-purpose setting, and in practice the lock registers too were used primarily to implement higher-level locks in memory.

Simple Software Lock Algorithms

Consider a lock operation used to provide atomicity for a critical section of code. For the acquire method, a process trying to obtain a lock must check that the lock is free and, if it is, then claim ownership of the lock. The state of the lock can be stored in a binary variable, with 0 representing free and 1 representing busy. A simple way of thinking about the lock acquire operation is that a process trying to obtain the lock should check if the variable is 0 and if so set it to 1, thus marking the lock busy; if the variable is 1 (lock is busy), then it should wait for the variable to turn to 0 using the waiting algorithm. An unlock operation should simply set the variable to 0 (the release method). The following are assembly-level instructions for this attempt at a lock and unlock. (In our pseudo-assembly notation, the first operand always specifies the destination if there is one.)

```
lock:  ld   register, location /*copy location to register*/
       cmp  register, #0      /*compare with 0*/
       bnz lock              /*if not 0, try again*/
       st   location, #1      /*store 1 into location to mark it locked*/
       ret                  /*return control to caller of lock*/
```

and

```
unlock: st location, #0      /*write 0 to location*/
        ret                  /*return control to caller*/
```

The problem with this lock, which is supposed to provide atomicity for the critical section that follows it, is that it needs (but lacks) atomicity in its own implementation. To illustrate this, suppose that the lock variable was initially set to 0 and two processes P_0 and P_1 execute the above assembly code implementations of the lock operation. Process P_0 reads the value of the lock variable as 0 and thinks it is free, so it proceeds past the branch instruction. Its next step is to set the variable to 1, marking the lock as busy, but before it can do this, process P_1 reads the variable as 0, thinks the lock is free, and passes the branch instruction too. We now have two processes simultaneously proceeding past the lock and entering the same critical section, which is exactly what the lock was meant to avoid. Putting the store instruction just after the load instruction would not help either. The two-instruction sequence—reading (testing) the lock variable to check its state and writing (setting) it to busy if it is free—is not atomic, and there is nothing to prevent these operations in different processes from being interleaved in time. What we need is a way to atomically test the value of a variable and set it to another value if the test succeeds (i.e., to atomically read and then conditionally modify a memory location) and then

to return whether the atomic sequence was executed successfully or not. One way to provide this atomicity for user processes is to place the lock routine in the operating system and access it through a system call, but this is expensive and leaves the question of how the locks are supported by the operating system itself. Another option is to utilize a hardware lock around the instruction sequence for the lock routine, but this requires hardware locks and tends to be slow on modern processors.

An efficient, general-purpose solution to the lock problem is to support an atomic read-modify-write instruction in the processor's instruction set. A typical approach is to have an atomic exchange instruction: a value at a memory location specified by the instruction is read into a register, and another value is stored into the location, all in an atomic operation with no other accesses to that location allowed to intervene. Many variants of this operation exist with varying degrees of flexibility in the nature of the value that can be stored. A simple example that works for mutual exclusion is an atomic *test&set* instruction. In this case, the value in the memory location is read into a specified register, and the constant 1 is stored into the location atomically. The success of the *test&set* is determined by examining the value in the register. If it is 0, the *test&set* was successful. If it is 1, it was not successful; the value 1 written to memory by the *test&set* instruction is the same as was already there, so no harm is done. (1 and 0 are the values typically used, though any other constants might be used in their place.) Given such an instruction, with the mnemonic *t&s*, we can write a lock and unlock in pseudo-assembly language as follows:

```
lock: t&s register, location
                  /*copy location to reg, and set location to 1*/
bnz register, lock /*compare old value returned with 0*/
                  /*if not 0, i.e., lock already busy, so try again*/
ret               /*return control to caller of lock*/
and
unlock: st location, #0      /*write 0 to location*/
ret                /*return control to caller*/
```

The lock implementation keeps trying to acquire the lock using *test&set* instructions until the *test&set* leaves zero in the register, indicating that the lock was free when tested (in which case the *test&set* has set the lock variable to 1, thus acquiring it). The unlock construct simply sets the location associated with the lock to 0, indicating that the lock is now free and enabling a subsequent lock operation by any process to succeed. A simple mutual exclusion construct has been implemented in software, relying on the fact that the architecture supports an atomic *test&set* instruction.

More sophisticated variants of such atomic instructions exist and, as we will see, are used by different software synchronization algorithms. One example is a *swap* instruction. Like a *test&set*, this reads the value from the specified memory location into the specified register, but instead of writing a fixed constant into the memory location, it writes whatever value was in the register to begin with. That is, it atomically exchanges or swaps the values in the memory location and the register. Clearly,

we can implement a lock as before by replacing the test&set with a swap instruction as long as we use the values 0 and 1 and ensure that the value in the register is 1 before the swap instruction is executed; the lock has succeeded if the value left in the register by the swap instruction is 0.

Another example is the family of *fetch&op* instructions. A *fetch&op* instruction also specifies a location and a register. It atomically reads the current value of the location into the register and writes the value (which has been obtained by applying the operation specified by the *fetch&op* instruction to the current value of the location) into the location. The simplest forms of *fetch&op* to implement are the *fetch&increment* and *fetch&decrement* instructions, which change the current value by 1. A *fetch&add* would take another operand, which is a register or value, to add into the previous value of the location. A more complex primitive is the *compare&swap* operation. It takes two register operands and a memory location (i.e., it is a three-operand instruction, not commonly supported by RISC architectures); it compares the value in the location with the contents of the first register operand, and, if the two are equal, it swaps the contents of the memory location with the contents of the second register.

Performance of the Simple Lock

Figure 5.29 shows the performance of a simple test&set lock on the SGI Challenge.⁸ Performance is measured for the following microbenchmark executed repeatedly in a loop:

```
lock(L);
critical-section(c);
unlock(L);
```

where *c* is a delay parameter that determines the size of the critical section (it is only a delay in this case, with no real work done). The benchmark is configured so that the same total number of lock calls are executed as the number of processors increases, reflecting a situation where a fixed number of tasks must be dequeued from a centralized task queue, independent of the number of processors. Performance is measured as the time per lock transfer, that is, the cumulative time taken by all processes executing the benchmark divided by the number of times the lock is obtained. The cumulative time spent in the critical section itself (i.e., *c* times the number of successful locks executed) is subtracted from the cumulative execution time so that only the time for the lock transfers themselves (or any contention caused by the lock operations) is obtained. All measurements are in microseconds.

8. In fact, the processor on the SGI Challenge, which is the machine for which synchronization performance is presented in this chapter, does not provide a test&set instruction. Rather, it uses alternative primitives that will be described later in this section. For these experiments, a mechanism whose behavior closely resembles that of test&set is synthesized from the available primitives. Results for real test&set-based locks on older machines like the Sequent Symmetry can be found in the literature (Granuke and Thakkar 1990; Mellor-Crummey and Scott 1991).

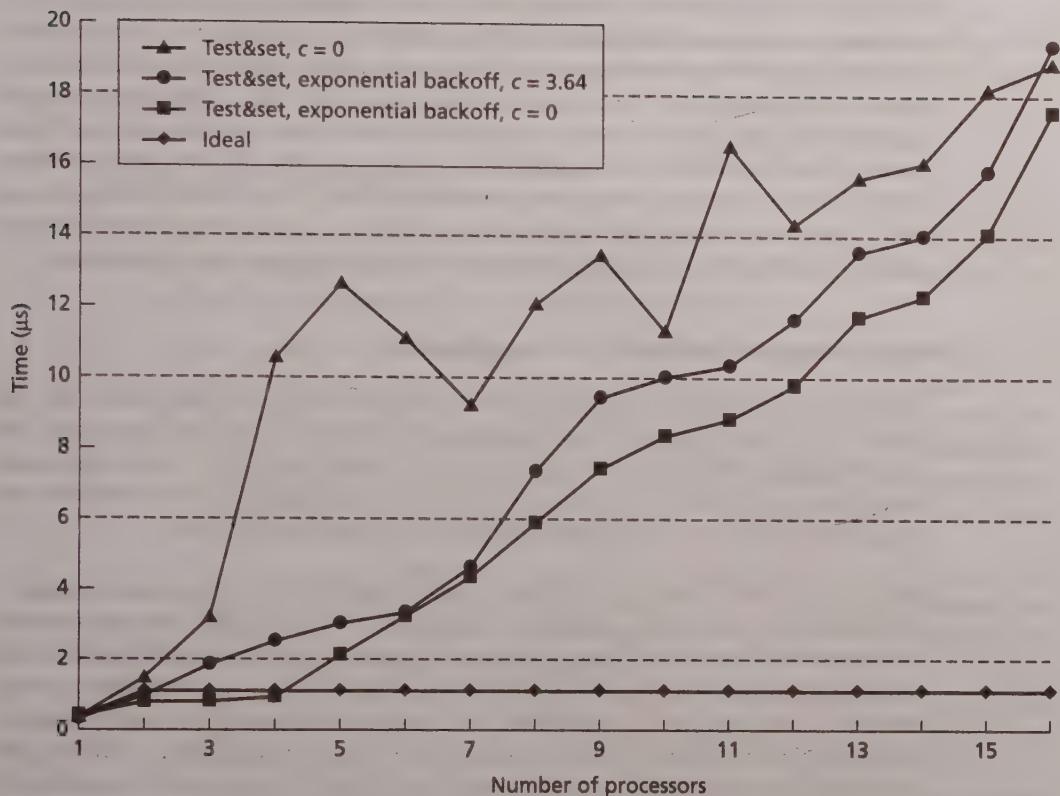


FIGURE 5.29 Performance of the synthesized test&set locks with an increasing number of competing processors on the SGI Challenge. The y-axis is the time per lock-unlock pair, excluding the critical section of size c microseconds. The irregular nature of the top curve is due to the timing dependence of the contention effects caused.

The upper curve in the figure shows the time per lock transfer with an increasing number of processors when using the test&set lock with a very small critical section (ignore the curves with “backoff” in their labels for now). Ideally, we would like the time per lock acquisition to be independent of the number of processors competing for the lock, with only one uncontended bus transaction per lock transfer, as shown in the curve labeled “ideal.” However, the figure shows that performance clearly degrades with an increasing number of processors.

The problem with the test&set lock is the traffic generated during the waiting method: every attempt to check whether the lock is free to be acquired, whether successful or not, generates a write operation to the cache block that holds the lock variable (since it uses a test&set operation and writes the value to 1); since this block is currently in the cache of some other processor (which wrote it last when doing its test&set), a bus transaction is generated by each write to invalidate the previous owner of the block. Thus, all processors put transactions on the bus repeat-

edly and consume precious bus bandwidth even during the waiting algorithm. The resulting contention slows down the lock transfer considerably as the number of processors, and hence the frequency of test&sets and bus transactions, increases. It impedes the progress of the processor releasing the lock and of the next processor that actually acquires it. In reality, it would also impede the work done in the critical section. The high degree of contention on the bus and the resulting timing dependence of obtaining locks causes the benchmark timing to vary sharply across numbers of processors used and even across executions. The results shown are for a particular, representative set of executions with different numbers of processors.

Enhancements to the Simple Lock Algorithm

We can do two simple things to alleviate this traffic. First, we can reduce the frequency with which processes issue test&set instructions while waiting; second, we can have processes busy-wait only with read operations so they do not generate invalidations and misses until the lock is actually released. These two possibilities are called the *test&set lock with backoff* and the *test-and-test&set lock*.

Test&Set Lock with Backoff The basic idea in backoff is for a process to insert a delay after an unsuccessful attempt to acquire the lock. The delay between test&set attempts should not be too long; otherwise, processors might remain idle even when the lock becomes free. But it should be long enough that traffic is substantially reduced. A natural question is whether the delay amount should be fixed or variable. Experimental results have shown that good performance is obtained by having the delay vary “exponentially”; that is, the delay after the first attempt is a small constant k that increases geometrically, so that after the i th attempt, it is $k \times c^i$, where c is another constant. Such a lock is called a test&set lock with exponential backoff. Figure 5.29 also shows the performance for the test&set lock with backoff for two different sizes of the critical section, using the starting value k for backoff that appears to perform best. Performance improves but still does not scale very well since there is still substantial traffic interfering with the release and acquire. Performance results using backoff with a real test&set instruction on older machines can be found in the literature (Granuke and Thakkar 1990; Mellor-Crummey and Scott 1991). See also Exercise 5.14, which discusses why the performance with a nonzero critical section is worse than that with a null critical section when backoff is used.

Test-and-Test&Set Lock A more subtle change to the algorithm is to have it use instructions that do not generate as much bus traffic while busy-waiting. Processes busy-wait by repeatedly reading with a standard load, not a test&set, the value of the lock variable until it turns from 1 (locked) to 0 (unlocked). On a cache-coherent machine, the reads can be performed in-cache by all processors, without generating bus traffic, since each obtains a cached copy of the lock variable the first time it reads it. When the lock is released, the cached copies of all waiting processes are invalidated, and the next read of the variable by each process will generate a read miss. The waiting processes will then find that the lock has been made available and only

then will each generate a test&set instruction to actually try to acquire the lock. One of them will succeed in this acquire attempt, while the others will fail and return to the read-based waiting method. The test-and-test&set lock substantially reduces bus traffic.

Performance Goals for Locks

Before examining more sophisticated lock algorithms and primitives, it is useful to clearly articulate some performance goals for locks and to review how the locks described here measure up. The goals include the following:

- *Low latency.* If a lock is free and no other processors are trying to acquire it at the same time, a processor should be able to acquire it with low latency.
- *Low traffic.* If many or all processors try to acquire a lock at the same time, they should be able to acquire the lock one after the other with as little generation of traffic or bus transactions as possible. As discussed earlier, contention due to high traffic can slow down lock acquisitions as well as unrelated transactions that compete for the bus (including in the critical section).
- *Scalability.* Neither latency nor traffic should scale quickly with the number of processors used. However, since the number of processors in a bus-based SMP is not likely to be large, it is not asymptotic scalability that is important but only scalability within the realistic range.
- *Low storage cost.* The information needed for a lock should be small and should not scale quickly with the number of processors.
- *Fairness.* Ideally, processors should acquire a lock in the same order as their requests are issued. At the least, starvation or substantial unfairness should be avoided. Since starvation is usually unlikely, the importance of fairness must be traded off with its impact on performance.

Consider the simple atomic exchange or test&set lock. It is very low latency if the same processor acquires the lock repeatedly without any competition, since the number of instructions executed is very small and the lock variable will stay in that processor's cache. However, we have seen that it can generate a lot of bus traffic and contention if many processors compete for the lock. The performance of the lock scales poorly as the number of competing processors increases. The storage cost is low (a single variable suffices) and does not scale with the number of processors. The lock makes no attempt to be fair, and an unlucky processor can be starved out. The test&set lock with backoff has the same uncontended latency as the simple test&set lock, generates less traffic, is somewhat more scalable, takes no more storage, and is no more fair. The test-and-test&set lock has slightly higher uncontended latency than the simple test&set lock (it does a read in addition to a test&set even when there is no competition) but generates much less bus traffic and is more scalable. It too requires negligible storage and is not fair. (Exercise 5.12 asks you to count the number of bus transactions and the time required for the test-and-test&set type of lock in different scenarios.)

In the test-and-test&set lock, since a test&set operation (and hence a bus transaction) is only issued when a processor is notified that the lock is ready, and thereafter if it fails it busy-waits (spins) on a cached block, there is no need for backoff. However, the lock does have the problem that when the lock is released, all waiting processes rush out and perform their read misses and their test&set instructions at about the same time. The bus transactions for the read misses may be combined in a smart bus protocol; however, each of the test&set instructions itself generates invalidations and subsequent misses, resulting in $O(p^2)$ bus traffic for p processors to acquire the lock once each. A random delay before issuing the test&set could help to stagger at least the test&set instructions, but it would increase the latency to acquire the lock in the uncontended case. While test-and-test&set was a major step forward at its time, better hardware primitives and better algorithms have been designed to alleviate its traffic problem.

Improved Hardware Primitives: Load-Locked, Store-Conditional

In addition to spinning with reads rather than read-modify-writes, which test-and-test&set accomplishes, we would prefer that failed attempts to complete the read-modify-write do not generate invalidations. It would also be useful to have a single primitive that allows us to implement a range of atomic read-modify-write operations—such as test&set, fetch&op, compare&swap—rather than implementing each with a separate instruction. One way to achieve both goals, increasingly supported in modern microprocessors, is to use a pair of special instructions rather than a single read-write-modify instruction to implement atomic access to a variable (let's call it a synchronization variable). The first instruction, commonly called *load-locked* or *load-linked* (LL), loads the synchronization variable into a register. It may be followed by arbitrary instructions that manipulate the value in the register—that is, the modify part of a read-modify-write. The last instruction of the sequence is the second special instruction, called a *store-conditional*. It tries to write the register back to the memory location (the synchronization variable) if and only if no other processor has written to that location (or cache block) since this processor completed its LL. Thus, if the store-conditional succeeds, it means that the load-locked, store-conditional (LL-SC) pair has read, perhaps modified in between, and written back the variable atomically. If the store-conditional detects that an intervening write has occurred to the variable or cache block, it fails and does not even try to write the value back (or generate any invalidations). This means that the atomic operation on the variable has failed and must be retried starting from the LL. Success or failure of the store-conditional is indicated by the condition codes or a return value. How the LL and store-conditional are actually implemented will be discussed later; for now, we are concerned with their semantics and performance.

Using LL-SC to implement atomic operations, the simple lock and unlock algorithms can be written as follows, where `reg1` is the register into which the current value of the memory location is loaded and `reg2` holds the value to be stored in the memory location by this atomic exchange (`reg2` could simply be the value 1 for a lock attempt, as in a test&set).

```

lock:    ll  reg1, location      /*load-locked the location to reg1*/
        bnz reg1, lock          /*if location was locked (nonzero),
                                try again*/
        sc   location, reg2      /*store reg2 conditionally into location*/
        beqz lock                /*if store-conditional failed, start again*/
        ret                      /*return control to caller of lock*/

```

and

```

unlock: st location, #0          /*write 0 to location*/
        ret                      /*return control to caller*/

```

Many processors may perform the LL at the same time, but only the first one that manages to put its store-conditional on the bus will actually succeed in its store-conditional. This processor will have succeeded in acquiring the lock, whereas the others will have failed and will have to retry the LL-SC. Note that the store-conditional may fail either because it detects the occurrence of an intervening write before even attempting to access the bus or because it attempts to get the bus but some other processor's store-conditional gets there first. Of course, if the location is 1 (nonzero) when a process does its LL, it will load 1 into reg1 and will retry the lock starting from the LL without even attempting the store-conditional.

It is worth noting that the LL itself is not a lock and the store-conditional itself is not an unlock. For one thing, the completion of the LL itself does not imply obtaining exclusive access; in fact, LL and store-conditional are used together to implement a lock operation. For another, even a successful LL-SC pair does not guarantee that the instructions between them (if any) are executed atomically with respect to those instructions on other processors, so in fact these instructions do not constitute a critical section. All that a successful LL-SC guarantees is that no conflicting writes to the synchronization variable itself intervene between the LL and store-conditional. In fact, since the instructions between the LL and store-conditional are executed unconditionally but should not be visible if the store-conditional fails, it is important that they do not modify any other important state. Typically, these instructions manipulate only the register into which the synchronization variable is loaded—for example, to perform the op part of a fetch&op—and do not modify any other program variables (modification of this register is okay since the register will be reloaded anyway by the LL in the next attempt). Microprocessor vendors that support LL-SC explicitly encourage software writers to follow this guideline and, in fact, often specify what instructions are possible to insert with a guarantee of correctness given their implementations of LL-SC. The number of instructions between the LL and store-conditional should also be kept small to reduce the probability of store-conditional failure due to an intervening write. Although the LL and store-conditional do not constitute a lock-unlock pair, they can be used directly to implement certain atomic operations on shared data structures. For example, if the desired function is a small operation on a globally shared variable (like a counter or global sum), it makes much more sense to implement it as the natural sequence (LL, register op, store-conditional, test) than to build a lock and unlock around the variable update.

Like the test-and-test&set, the spin-lock built with LL-SC does not generate bus traffic during the waiting algorithm if the LL indicates that the lock is currently held. Better than the test-and-test&set, it also does not generate invalidations on a failed attempt to obtain the lock (i.e., a failed store-conditional). However, when the lock is released, the processors spinning in a tight loop of load-locked operations will indeed miss on the location and rush out to the bus with read transactions. After this, only a single invalidation will be generated for a given lock acquisition by the processor whose store-conditional succeeds, but this will again invalidate all caches. Traffic is reduced greatly from even the test-and-test&set case, down from $O(p^2)$ to $O(p)$ per lock acquisition, but still increases with the number of processors. Since spinning on a locked location is already done through reads (load-locked operations), no analog of a test-and-test&set exists to further improve its performance. However, backoff can be used between the LL and store-conditional to reduce bursty traffic.

The simple LL-SC lock is also low in latency and storage, but it is not a fair lock and does not reduce traffic to a minimum. More advanced lock algorithms can be used that provide both fairness and reduced traffic. They can be built using either atomic read-modify-write instructions or atomic operations of equivalent semantics synthesized with LL-SC, though of course the traffic advantages are different in the two cases. Let us consider two of these algorithms that are appropriate for bus-based machines.

Advanced Lock Algorithms

Especially when using an atomic exchange instruction like test&set, instead of LL-SC, to implement locks, it is desirable to have only one process actually attempt to obtain the lock when it is released (rather than have them all rush out to do a test&set and issue invalidations as in all the preceding algorithms). It is even more desirable to have only one process incur a read miss (even with LL-SC) when a lock is released. The *ticket lock* accomplishes the first purpose; the *array-based lock* accomplishes both goals but at a little cost in space. Unlike all the previous locks, both these locks are fair and grant the lock to processors in FIFO order.

Ticket Lock The ticket lock operates just like the ticket system in the sandwich line at a delicatessen or like the teller line at a bank. Every process wanting to acquire the lock takes a ticket number and then busy-waits on a global now-serving number—like the number on the LED display that we watch intently in the sandwich line—until the now-serving number equals the ticket number it obtained. To release the lock, a process simply increments the now-serving number so that the next waiting process can acquire the lock. The atomic primitive needed is a fetch&increment, which a process uses when it first reaches the lock operation to obtain its ticket number from a shared counter. No atomic operation (e.g., test&set) is needed to actually obtain the lock upon a release since only the unique process that has its ticket number equal to now-serving attempts to enter the critical section when it sees the release. Thus, the acquire method is the fetch&increment, the

waiting algorithm is busy-waiting for now-serving to equal the ticket number, and the release method is to increment now-serving. This lock has uncontended latency about equal to the test-and-test&set lock but generates much less traffic. Although every process does a fetch&increment when it first arrives at the lock (presumably not every process at the same time), the test&set attempts upon a release of the lock are eliminated, which tend to be simultaneous and a lot more heavily contended. The ticket lock also requires constant and small storage and is fair since processes obtain the lock in the order of their fetch&increment operations.

The fetch&increment needed by the ticket lock can be implemented with LL-SC. However, since the simple LL-SC lock already avoids multiple processors issuing invalidations in trying to acquire a lock after its release, there is not a large difference in traffic between the ticket lock and the simple LL-SC lock. (The simple LL-SC lock is somewhat worse since in that case another invalidation and set of read misses occur when a processor succeeds in its store-conditional.) The key difference between these two locks is fairness.

Like the simple LL-SC lock, the ticket lock still has a read traffic problem at a release. The reason is that all processes spin on the same variable (now-serving). When that variable is written at a release, all processors' cached copies are invalidated, and they all incur a read miss. The read misses may be combined on some buses but can cause unnecessary traffic if the combining is unavailable or unsuccessful. One way to reduce this bursty read-miss traffic is to introduce a form of backoff. We do not want to use exponential backoff because we do not want all processors to be backing off when the lock is released so that none tries to acquire it for a while. A promising technique is to have each processor back off from trying to read the now-serving counter by a duration proportional to when it expects its turn to actually come—that is, by a duration proportional to the difference in its ticket number and the now-serving value it last read. Alternatively, the array-based lock completely eliminates this extra read traffic upon a release by having every process spin on a distinct location.

Array-Based Lock The idea here is to use a fetch&increment to obtain not a value but a unique location on which to busy-wait. If there are p processes that might possibly compete for a lock, then the lock data structure contains an array of p locations that processes can spin on, ideally each on a separate memory block to avoid false sharing. The acquire method then uses a fetch&increment operation to obtain the next available location in this array (with wraparound), the waiting method spins on this location, and the release method writes a value denoting “unlocked” to the next location in the array (after the one that the releasing processor was itself spinning on). Only the processor that was spinning on that next location has its cache block invalidated at the release; its consequent read miss tells it that it has obtained the lock. As in the ticket lock, no test&set is needed after the miss since only one process is notified when the lock is released. This lock is clearly also FIFO and hence fair. Its uncontended latency is likely to be similar to that of the test-and-test&set lock (a fetch&increment followed by a read of the assigned array location), and it is potentially more scalable than the ticket lock since only one processor incurs the

read miss. For the same reason, unlike the ticket lock, it does not need any form of backoff to reduce traffic. Its only drawback for a bus-based machine is that it uses $O(p)$ space rather than $O(1)$, but with both p and the proportionality constant being small, this is usually not a very significant drawback. It has a potential drawback for machines with distributed memory, but we shall discuss this drawback and lock algorithms that overcome it in Chapter 7.

Performance

Let us briefly examine the performance of the different locks on the SGI Challenge, as shown in Figure 5.30. All locks are implemented using LL-SC since the Challenge provides only these and not atomic instructions. Results are shown for a somewhat more parameterized version of the earlier microbenchmark code, in which a process is allowed to insert a delay not only for the critical section but also between its release of the lock and its next attempt to acquire it (as will happen in a real program). That is, the code is a loop over the following body:

```
lock(L);
critical_section(c);
unlock(L);
delay(d);
```

Let us consider three cases: (1) $c = 0, d = 0$; (2) $c = 3.64 \mu\text{s}, d = 0$; and (3) $c = 3.64 \mu\text{s}, d = 1.29 \mu\text{s}$ —called the *null* critical section case, the *non-null* critical section case, and the non-null critical section with *delay* case, respectively. The delays c and d are inserted in the code as round numbers of processor cycles, which translates to these microsecond numbers. Recall that in all cases, the delays c and d (multiplied by the number of lock acquisitions by each processor) are subtracted out of the total time, which is supposed to measure only the total time taken for a certain number of lock acquisitions and releases (see also Exercise 5.15).

Consider the null critical section case. The first observation, comparing Figure 5.30 with Figure 5.29, is that all the other locks are indeed better than the test&set locks, as expected.⁹ The second observation is that the simple LL-SC locks actually seem to perform better than the more sophisticated ticket lock and array-based lock. For these locks, which don't encounter as much contention as the test&set lock, performance is largely determined by the number of bus transactions between a release and a successful acquire. The reason that the LL-SC locks perform so well, particularly at lower processor counts, is that they are not fair, and the unfairness is exploited by architectural interactions! In particular, when a processor that releases a lock with a write follows it immediately with the read (LL) for its next acquire, its read and the subsequent store-conditional are likely to succeed in its cache before

9. The test&set is simulated using LL-SC as follows: every time a store-conditional fails, a write is performed to another variable in the same cache block, causing invalidations as a test&set would. This method of simulating test&set with LL-SC may lead to somewhat worse performance than a true test&set primitive, but it conveys the trend.

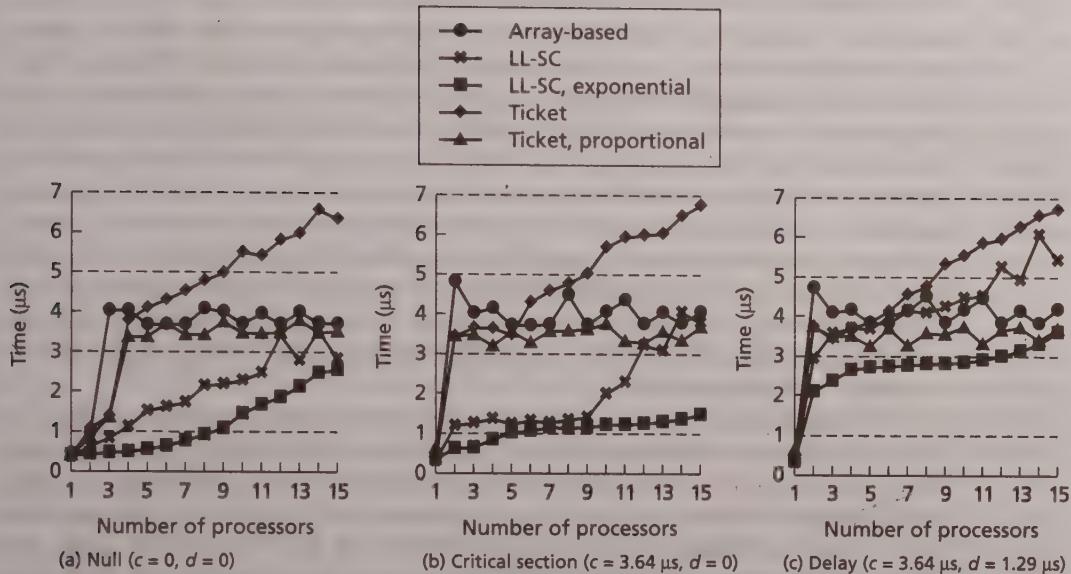


FIGURE 5.30 Performance of locks on the SGI Challenge for three different scenarios

another processor can read the block across the bus. (The bias on the SGI Challenge is actually more severe, since the releasing processor can satisfy its next read from its write buffer even before the read exclusive corresponding to the releasing write gets out on the bus.) Lock transfer is very quick, and performance is good, but the same processor keeps acquiring the lock repeatedly. As the number of processors and the competition for the bus increase, the likelihood of the last releaser's store-conditional successfully obtaining the bus decreases, and hence the likelihood of self-transfers decreases. In addition, bus traffic increases due to invalidations and read misses, so the time per lock transfer increases. Exponential backoff helps reduce the burstiness of traffic and hence slows the rate of scaling, and a nonzero critical section ($c = 3.64, d = 0$) helps this along further.

With delays both inside and outside the critical section ($c = 3.64, d = 1.29$), we see the LL-SC lock not doing quite as well, even at low processor counts. This is because a processor waits after its release before trying to acquire the lock again, making it much more likely that some other waiting processor will acquire the lock before it. Self-transfers are unlikely, so lock transfers are slower even with two processors. It is interesting that performance is particularly worse for the backoff case at small processor counts when the delay d between unlock and lock is nonzero. This is because it is quite likely that while the processor that just released the lock is waiting for d to expire before doing its next acquire, all the other processors are in a backoff period and not even trying to acquire the lock. In the $d = 0$ case, the releasing processor reacquires the lock right away, especially with a small number of processors. Backoff must be used carefully for it to be successful.

Consider the other locks. These are fair, so every lock transfer is to a different processor and involves bus transactions in the critical path of the transfer. Hence, they all start off with a jump to about three bus transactions in the critical path per lock transfer even when two processors are used. Actual differences in time are due to exactly which bus transactions are generated and how much of their latency can be hidden from the processor. The ticket lock without backoff scales relatively poorly: with all processors trying to read the now-serving counter, the expected number of bus transactions between the release and the read by the correct processor is $p/2$, leading to the observed linear degradation in the lock transfer critical path. With successful proportional backoff, it is likely that the correct processor will be the one to issue the read first after a release, so the time per transfer is constant and does not scale with p . The array-based lock also scales well since only the correct processor issues a read.

The results illustrate the importance of detailed architectural interactions in determining the performance of locks. They also show that simple LL-SC locks perform quite well on buses that have sufficient bandwidth. On this particular machine, performance for the unfair LL-SC lock becomes as bad as or a little worse than that for the more sophisticated locks beyond 16 processors due to the higher traffic, but not by much because bus bandwidth is quite high. When exponential backoff is used to reduce traffic, the simple LL-SC lock delivers the best average lock transfer time in all cases. However, these results also illustrate the difficulty and the importance of sound experimental methodology in evaluating synchronization algorithms. Null critical sections display some interesting effects, but meaningful comparisons depend on what the synchronization patterns look like in practice—in real applications. For example, the effect of critical section and delay size on the frequency of self-transfers has a substantial impact on the comparison of unfair locks with fair locks. The nonrepresentativeness of the null case in this regard is therefore an important methodological consideration. An experiment to use LL-SC while guaranteeing round-robin acquisition among processors (fairness) by using an additional variable showed performance very similar to that of the ticket lock, confirming that unfairness and self-transfers are indeed the reason for the better performance at low processor counts. Especially if fairness is desired, the ticket lock with proportional backoff and the array-based lock perform very well on bus-based machines.

Lock-Free, Nonblocking, and Wait-Free Synchronization

An additional set of performance concerns involving synchronization arises when we consider that the machine running our parallel program is used in a multiprogramming environment. Other processes run for periods of time or, even if we have the machine to ourselves, background daemons run periodically, processes take page faults, I/O interrupts occur, and the process scheduler makes scheduling decisions with limited information about the application requirements. These events can cause the rate at which processes make progress to vary considerably. One important question is how the parallel program as a whole slows down when one process is slowed. With traditional locks, the problem can be serious: if a process holding a

lock stops or slows while in its critical section, all other processes may have to wait. This problem has received a good deal of attention in work on operating system schedulers. In some cases, attempts are made to avoid preempting a process that is holding a lock. Another line of research takes the view that lock-based operations are not very robust and should be avoided; for example, if a process dies while holding a lock, other processes hang. It has been observed that most lock-unlock operations are used to support operations on a well-defined data structure or object that is shared by several processes, for example, updating a shared counter or manipulating a shared queue. These higher-level operations on the data structure can be implemented directly using atomic primitives without actually using locks, as discussed for LL-SC earlier.

A shared data structure is said to be *lock-free* if the operations defined on it do not require mutual exclusion over multiple instructions. If the operations on the data structure guarantee that some process will complete its operation in a finite amount of time, even if other processes halt, the data structure is *nonblocking*. If the operations can guarantee that every (nonfaulting) process will complete its operation in a finite amount of time, the data structure is *wait-free* (Herlihy 1993). A body of literature is available that investigates the theory and practice of such data structures, including requirements placed on the basic atomic primitives to implement them (Herlihy 1988), general-purpose techniques for translating sequential operations to nonblocking concurrent operations (Herlihy 1993), specific useful lock-free data structures (Valois 1995; Michael and Scott 1996), operating system implementations (Massalin and Pu 1991; Greenwald and Cheriton 1996), and proposals for architectural support (Herlihy and Moss 1993). The basic approach is to implement updates to a shared object by reading a portion of the object to make a copy, updating the copy, and then performing an operation to commit the change only if no conflicting updates have been made (reminiscent of LL-SC). As a simple example, consider a shared counter. The counter is read into a register, a value is added to the register copy, and the result is put in a second register. Next, a compare&swap updates the shared counter only if its value is still the same as the copy. For more sophisticated, linked-list data structures, a new element is created and then linked into the shared list if the insert is still valid. These techniques serve to limit the window in which the shared data structure is in an inconsistent state, so they improve robustness, although it can be difficult to make them efficient.

Theoretical research has identified the properties of different atomic exchange operations in terms of the time complexity of using them to implement synchronized access to variables. In particular, it has been found that simple operations like test&set and fetch&op are not powerful enough to guarantee that the time taken by a processor to access a synchronized variable is independent of the number of processors, whereas more sophisticated atomic operations like compare&swap and swapping the values of two memory locations are powerful enough to make this guarantee (Herlihy 1988).

Having discussed the options for mutual exclusion on bus-based machines, let us move on to point-to-point, and then barrier, event synchronization.

5.5.4 Point-to-Point Event Synchronization

Point-to-point synchronization within a parallel program is often implemented by busy-waiting on ordinary variables, using them as flags. If we want to use blocking instead of busy-waiting, we can use semaphores, just as they are used in concurrent programming and operating systems (Tanenbaum and Woodhull 1997).

Software Algorithms

Flags are control variables, typically used to communicate the occurrence of a synchronization event rather than to transfer values. If two processes have a producer-consumer relationship on the shared variable a , then a flag can be used to manage the synchronization as follows:

P_1	P_2
$a = f(x); \ /*set a*/$	$while (flag \text{ is } 0) \text{ do nothing};$
$flag = 1;$	$b = g(a); \ /*use a*/$

If we know that the variable a is initialized to a certain value (say, 0), which will be changed to a new value we are interested in by this production event, then we can use a itself as the synchronization flag, as follows:

P_1	P_2
$a = f(x); \ /*set a*/$	$while (a \text{ is } 0) \text{ do nothing};$
	$b = g(a); \ /*use a*/$

This eliminates the need for a separate flag variable and saves the write to and read of that variable at perhaps some cost in readability and maintainability.

Hardware Support: Full-Empty Bits

This idea of special flag values has been extended in some research machines (although mostly in machines with physically distributed memory) to provide hardware support for fine-grained producer-consumer synchronization. A bit, called a *full-empty bit*, is associated with every word in memory. This bit is set when the word is “full” with newly produced data (i.e., on a write) and unset when the word is “emptied” by a processor consuming that data (i.e., on a read). Word-level producer-consumer synchronization is then accomplished as follows. When the producer process wants to write the location, it does so only if the full-empty bit is set to empty and then leaves the bit set to full. The consumer reads the location only if the bit is full and then sets it to empty. Hardware preserves the atomicity of the read or write

with the manipulation of the full-empty bit. Given full-empty bits, our preceding example can be written without the spin loop as

P_1	P_2
$a = f(x); /*set a*/$	$b = g(a); /*use a*/$

Full-empty bits raise concerns about flexibility. For example, they do not lend themselves easily to single-producer-multiple-consumer synchronization or to the case where a producer updates a value multiple times before a consumer consumes it. Also, should all reads and writes use full-empty bits or only those that are compiled down to special instructions? The latter method requires support in the language and compiler, but the former is too restrictive in imposing synchronization on all accesses to a location (for example, it does not allow asynchronous relaxation in iterative equation solvers; see Chapter 2). For these reasons, and the hardware cost, full-empty bits have not found favor in most commercial machines.

Interrupts

Another important kind of event is the interrupt conveyed from an I/O device needing attention to a processor. In a uniprocessor machine, there is no question where the interrupt should go, but in an SMP any processor can potentially take the interrupt. In addition, there are times when one processor may need to issue an interrupt to another. In early SMP designs, special hardware was provided to monitor the priority of the process on each processor and to deliver the I/O interrupt to the processor running at lowest priority. Such measures proved to be of small value, and most modern machines use simple arbitration strategies. In addition, a memory-mapped interrupt control region usually exists, so at kernel level any processor can interrupt any other by writing the interrupt information at the associated address.

5.5.5 Global (Barrier) Event Synchronization

Finally, let us examine barrier synchronization on a bus-based machine. Software algorithms for barriers are typically implemented using locks, shared counters, and flags. Let us begin with a simple barrier among p processes, which is called a *centralized barrier* since it uses only a single lock, a single counter, and a single flag.

Centralized Software Barrier

A shared counter maintains the number of processes that have arrived at the barrier and is therefore incremented by every arriving process. These increments must be mutually exclusive. After incrementing the counter, a process checks to see if the counter equals p , that is, if it is the last process to have arrived. If not, it busy-waits

on the flag associated with the barrier; if so, it writes the flag to release the $p - 1$ waiting processes. A simple attempt at a barrier algorithm may therefore look like

```

struct bar_type {
    int counter;
    struct lock_type lock;
    int flag = 0;
} bar_name;

BARRIER (bar_name, p)
{
    LOCK(bar_name.lock);
    if (bar_name.counter == 0)           /*reset flag if first to reach*/
        bar_name.flag = 0;              /*mycount is a private variable*/
    mycount = bar_name.counter++;       /*last to arrive*/
    UNLOCK(bar_name.lock);
    if (mycount == p) {                /*reset counter for next barrier*/
        bar_name.counter = 0;
        bar_name.flag = 1;             /*release waiting processes*/
    }
    else
        while (bar_name.flag == 0) {}; /*busy-wait for release*/
}

```

Centralized Barrier with Sense Reversal

Can you see a problem with the preceding barrier? There is one. It occurs when the barrier operation is performed consecutively using the same barrier variable—for example, if each processor executes the following code:

```

some computation...
BARRIER(bar1, p);
some more computation...
BARRIER(bar1, p);

```

The first process to enter the barrier the second time reinitializes the barrier counter, so that is not a problem. The problem is the flag. To exit the first barrier, processes spin on the flag until it is set to 1. Processes that see the flag change to 1 will exit the barrier, perform the subsequent computation, and enter the barrier again. However, suppose one processor P_x does not see the flag change from the first barrier before others have reentered the barrier for the second time; for example, it gets swapped out by the operating system because it has been spinning too long. When it is swapped back in, it will continue to wait for the flag to change to 1. In the meantime, other processes may have already entered the second instance of the barrier, and the first of these will have reset the flag to 0. Now the flag can only get set to 1

again when all p processes have registered at the new instance of the barrier, which will never happen since P_x will never leave the spin loop from the first barrier.

How can we solve this problem? What we need to do is prevent a process from entering a new instance of a barrier until all processes have exited the previous instance of the same barrier. One way is to use another counter to count the processes that leave the barrier and to not let a process reset the flag in a new barrier instance until this counter has turned to p for the previous instance. However, manipulating this counter incurs further latency and contention. On the other hand, with the current setup we cannot wait for all processes to reach the barrier before resetting the flag to 0, since that is when we actually set the flag to 1 for the release. A better solution is to avoid explicitly resetting the flag value altogether and rather have processes wait for the flag to obtain a different release value in consecutive instances of the barrier. For example, processes may wait for the flag to turn to 1 in one instance and to turn to 0 in the next instance. A private variable is used per process to keep track of which value to wait for in the current barrier instance. Since by the semantics of a barrier a process cannot get more than one barrier ahead of another, we only need two values (0 and 1) that we toggle between each time. Hence we call this method *sense reversal*. Now, in the previous example, the flag need not be reset when the first process reaches the barrier; rather, the process stuck in the old barrier instance still waits for the flag to reach the old release value while processes that enter the new instance wait for the other (toggled) release value. The value of the flag is only changed once when all processes have reached the (new) barrier instance, so it will not change before processes stuck in the old instance see it. Here is the code for a simple barrier with sense reversal:

```
BARRIER (bar_name, p)
{
    local_sense = !(local_sense);      /*toggle private sense variable*/
    LOCK(bar_name.lock);
    mycount = bar_name.counter++;     /*mycount is a private variable*/
    if (bar_name.counter == p) {       /*last to arrive*/
        UNLOCK(bar_name.lock);
        bar_name.counter = 0;          /*reset counter for next barrier*/
        bar_name.flag = local_sense;  /*release waiting processes*/
    }
    else {
        UNLOCK(bar_name.lock);
        while (bar_name.flag != local_sense) {}; /*busy-wait for
                                                     release*/
    }
}
```

Note that the lock is not released immediately after the increment of the counter but only after the condition is evaluated; the reason for this is revealed in an exercise (see Exercise 5.18). We now have a correct barrier that can be reused any number of times consecutively. The remaining issue is performance, which we examine next.

(Note that the LOCK/UNLOCK protecting the increment of the counter can be replaced more efficiently by a simple LL-SC or atomic increment operation.)

Performance

The major performance goals for a barrier are similar to those for locks. They include the following:

- *Low latency (small critical path length).* The chain of dependent operations and bus transactions needed for p processors to pass the barrier should be small.
- *Low traffic.* Since barriers are global operations, it is quite likely that many processors will try to execute a barrier at the same time. The barrier algorithm should reduce the total number of bus transactions (whether in the critical path or not) and hence the possible contention.
- *Scalability.* Latency and traffic should increase slowly with the number of processors.
- *Low storage cost.* We would, of course, like to keep the storage cost low.
- *Fairness.* We should ensure that the same processor does not always become the last one to exit the barrier (or we may want to preserve FIFO ordering).

In the centralized barrier described previously, each processor accesses the lock once, hence the critical path length is at least proportional to p . Consider the bus traffic. To complete its operation, a centralized barrier involving p processors performs $2p$ bus transactions for processors to obtain the lock and increment the counter, two bus transactions for the last processor to reset the counter and write the release flag, and another $p - 1$ bus transactions to read the flag after it has been invalidated. Note that this is better than the traffic for even a test-and-test&set lock to be acquired by p processes because, in that case, each of the p releases causes an invalidation that results in $O(p)$ processes trying to perform the test&set again, thus resulting in $O(p^2)$ bus transactions. However, the contention resulting from these competing bus transactions can be substantial if many processors arrive at the barrier simultaneously, so barriers can be expensive.

Improving Barrier Algorithms for a Bus

One part of the problem in the centralized barrier is that all processors contend for the same lock and flag variables. To address this, we can construct barriers that cause fewer processors to contend for the same variable. For example, processors can signal their arrival at the barrier through a software combining tree (see Section 3.3.2). In a binary combining tree, for example, only two processors notify each other of their arrival at each node of the tree, and only one of the two moves up to participate at the next higher level of the tree. Thus, only two processors ever access a given variable. In a distributed network with multiple parallel paths, such as those found in scalable machines, a combining tree can perform much better than a centralized barrier since different pairs of processors can communicate with each other

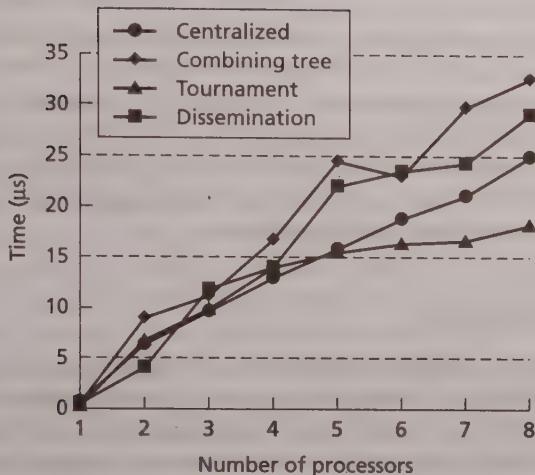


FIGURE 5.31 Performance of some barriers on the SGI Challenge. Performance is measured as average time per barrier over a loop of many consecutive barriers (with no work or delays between them). The higher critical path latency of the combining tree barrier hurts it on a bus, where it has no traffic and contention advantages.

in different parts of the network in parallel. However, with a centralized interconnect like a bus, even though pairs of processors communicate through different variables, they all generate bus transactions and hence serialization and contention on the same bus. Since a binary tree with p leaves has approximately $2p$ nodes, a combining tree requires a similar total number of bus transactions to the centralized barrier. It also has higher latency since, while it too requires $O(p)$ serialized bus transactions in all, even without bus serialization each processor must wait at least $\log p$ steps to get from the leaves to the root of the tree, each with significant work. The advantage of a combining tree for a bus is that it does not use locks but, rather, simple read and write operations, which may compensate for its larger uncontended latency if the number of processors on the bus is large. However, the simple centralized barrier performs quite well on a bus, as shown in Figure 5.31. Some of the other barriers shown in the figure for illustration will be discussed along with tree barriers in the context of scalable machines in Chapter 7.

Hardware Primitives

Since the centralized barrier uses locks and ordinary reads and writes, the hardware primitives needed depend on which lock algorithms are used. If a machine does not support atomic primitives well, combining tree barriers can be useful for bus-based machines as well.

A special bus primitive can be used to reduce the number of bus transactions for read misses in the centralized barrier (as well as for highly contended locks in which

processors spin on the same variable). This optimization takes advantage of the fact that all processors issue a read miss for the same value of the flag when they are invalidated at the release. Instead of all processors issuing a separate read-miss bus transaction, a processor can monitor the bus and abort its read miss before putting it on the bus, if it sees the response to a read miss to the same location (issued by another processor that happened to get on the bus first), and simply take the return value from the bus. In the best case, this piggybacking can reduce the number of read-miss bus transactions from p to 1.

Hardware Barriers

If a separate synchronization bus is provided, as discussed for locks, it can be used to support barriers in hardware too. This takes the traffic and contention off the main system bus and can lead to higher-performance barriers. Conceptually, a single wired-AND line is enough. A processor sets its input high when it reaches the barrier and waits until the output goes high before it can proceed. (In practice, reusing barriers requires that more than a single wire be used.) Such a separate hardware mechanism for barriers can be particularly useful if the frequency of barriers is very high, as it may be in programs that are automatically parallelized by compilers at the inner loop level and that need global synchronization after every innermost loop. However, its value in practice is unclear, and it can be difficult to manage when only a portion of the processors on the machine participate in the barrier. For example, it is difficult to dynamically change the number of processors participating in the barrier or to adapt the configuration of participating processors when processes are migrated among processors by the operating system. Having multiple participating processes running on the same processor also causes complications. Current bus-based multiprocessors therefore do not tend to provide special hardware support but build barriers in software out of locks and shared variables.

5.5.6 Synchronization Summary

Some bus-based machines have provided full hardware support for synchronization operations such as locks and barriers. However, concerns about flexibility have led most contemporary designers to provide support for only simple atomic operations in hardware and to synthesize higher-level synchronization operations from them in software libraries. The application programmer generally uses the libraries and can be unaware of the low-level atomic operations supported on the machine. The atomic operations may be implemented either as single instructions or through speculative read-write instruction pairs like load-locked and store-conditional. The greater flexibility of the latter is making them increasingly popular. We have already seen some of the interplay between synchronization primitives, algorithms, and architectural details. This interplay will be much more pronounced when we discuss synchronization for scalable shared address space machines in the coming chapters.

5.6 IMPLICATIONS FOR SOFTWARE

So far, we have looked at high-level architectural issues for bus-based cache-coherent multiprocessors and at how architectural and protocol trade-offs are affected by workload characteristics. Let us now come full circle and examine how the architectural characteristics of these small-scale machines influence parallel software. That is, instead of keeping the workload fixed and improving the machine or its protocols, we keep the machine fixed and examine how to improve parallel programs. Improving synchronization algorithms to reduce traffic and latency was an example of this, but let us look at the parallel programming process more generally.

The general techniques for load balance and inherent communication discussed in Chapter 3 also apply to cache-coherent machines. In addition, one general partitioning principle that is applicable across a wide range of computations on these machines is to try to assign computation such that only one processor writes a given set of data, at least during a single computational phase. In many computations, processors read one large shared data structure and write another. In Raytrace, for example, processors read a scene and write an image. A choice is available of whether to partition the computation so the processors write disjoint pieces of the destination structure and read share the source structure, or read disjoint pieces of the source structure and write share the same memory locations in the destination. All other considerations being equal (such as load balance and programming complexity), it is usually advisable to avoid write sharing in these situations. Write sharing not only causes invalidations and, hence, cache misses and traffic, but if different processes write the same words, it is very likely that the writes must be protected by synchronization such as locks, which are even more expensive.

The structure of communication is not much of a variable: with a single centralized memory, little incentive exists to use explicit memory-to-memory data transfers, so all communication is implicit through loads and stores that lead to the transfer of cache blocks. Mapping is not an issue (other than to try to ensure that processes migrate from one processor to another as little as possible) and is invariably left to the operating system. The most interesting issues are managing data locality and artificial communication in the orchestration step, and in particular, addressing temporal and spatial locality to reduce the number of cache misses and hence reduce latency, traffic, and contention on the shared bus.

With main memory being centralized, temporal locality is exploited in the processor caches. The specialization of the working set curve introduced in Chapter 3 for bus-based machines is shown in Figure 5.32. All capacity-related misses go to the same bus and memory and are about as expensive as coherence misses. The other three kinds of misses will occur and generate bus traffic even with an infinite cache. The major goal for temporal locality is to have working sets fit in the cache hierarchy, and the techniques are the same as those discussed in Chapter 3.

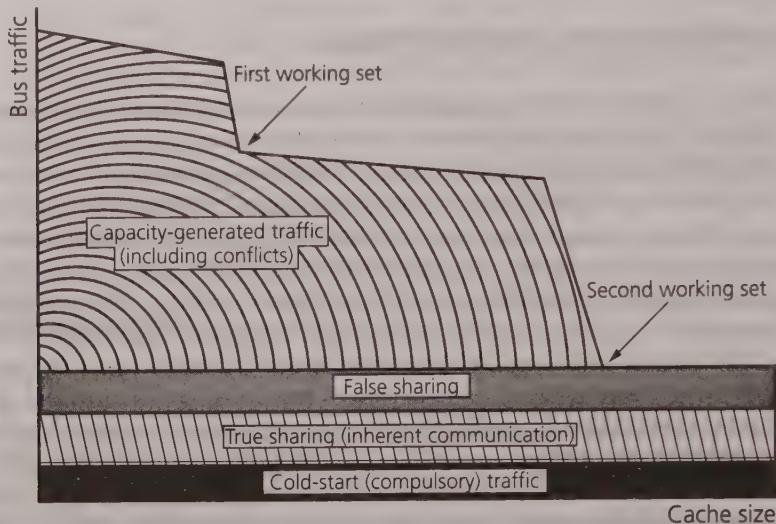


FIGURE 5.32 Data traffic on the shared bus and its components as a function of cache size. The points of inflection indicate the working sets of the program.

For spatial locality, a centralized memory makes data distribution and the granularity of allocation in main memory irrelevant (only interleaving data among memory banks to reduce contention may be an issue, just as in uniprocessors). The ill effects of poor spatial locality are *fragmentation* (i.e., fetching unnecessary data on a cache block) and false sharing. The reasons are that the granularity of communication and the granularity of coherence are both cache blocks, which are larger than a word. The former causes fragmentation, and the latter causes false sharing. (We assume here that techniques to eliminate false sharing like subblock dirty bits are not used since they are not found in most real machines.) Let us examine some techniques to alleviate these problems and effectively exploit the prefetching effects of long cache blocks, as well as techniques to alleviate cache conflicts by better spatial organization of data. Many such techniques can be found in a programmer's "bag of tricks." The following provides only a sampling of the most general ones.

- Assign tasks to reduce spatial interleaving of access patterns. It is desirable to assign tasks such that each processor tends to access large contiguous chunks of data. For example, if an array computation with n elements is to be divided among p processors, it is better to divide it so that each processor accesses n/p contiguous elements rather than to use a finely interleaved assignment of elements. This increases spatial locality and reduces false sharing of cache blocks. Of course, load balancing or other constraints may force us to do otherwise.
- Structure data to reduce spatial interleaving of access patterns. We saw an example of this in the equation solver kernel in Chapter 3, when we used higher-dimensional arrays to keep a processor's partition of an array contiguous in the

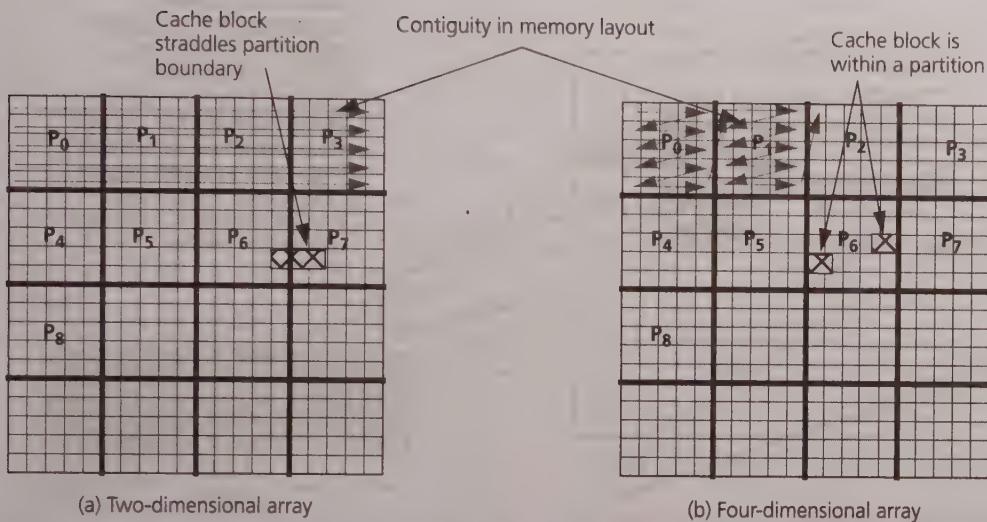


FIGURE 5.33 Reducing false sharing and fragmentation by using higher-dimensional arrays to keep partitions contiguous in the address space. In the two-dimensional array case, cache blocks straddling partition boundaries cause both fragmentation (a miss brings in useless data from the other processor's partition) as well as false sharing. The four-dimensional array representation makes partitions contiguous and alleviates these problems.

address space in order to allocate partitions locally at page granularity in physically distributed memory. This technique also helps reduce false sharing, fragmentation of data transfer, and conflict misses, as shown in Figures 5.33 and 5.34, all of which cause misses and traffic on the bus. A cache block larger than a single grid element may straddle a column-oriented partition boundary, as shown in Figure 5.33(a). If the block is larger than two grid elements, it can cause communication due to false sharing. This is easiest to see if we assume for a moment that there is no inherent communication in the algorithm; for example, suppose in each sweep a process simply adds a constant value to each of its assigned grid elements instead of performing a nearest-neighbor computation. Now, even a two-element (or larger) cache block straddling a partition boundary would be false-shared as different processors wrote different words on it. This would also cause fragmentation in communication, since a process reading its own boundary element and missing on it would also fetch other elements in the other processor's partition that are on the same cache block but that it does not need. The conflict-misses problem is explained in Figure 5.34. The issue in all these cases is noncontiguity of partitions. Thus, a single data structure transformation (as in Figure 5.33[b]) helps us solve all our spatial locality-related problems in the equation solver kernel. Figure 5.35 illustrates the performance impact of using higher-dimensional arrays to represent grids or blocked matrices in the Ocean and LU applications on the SGI

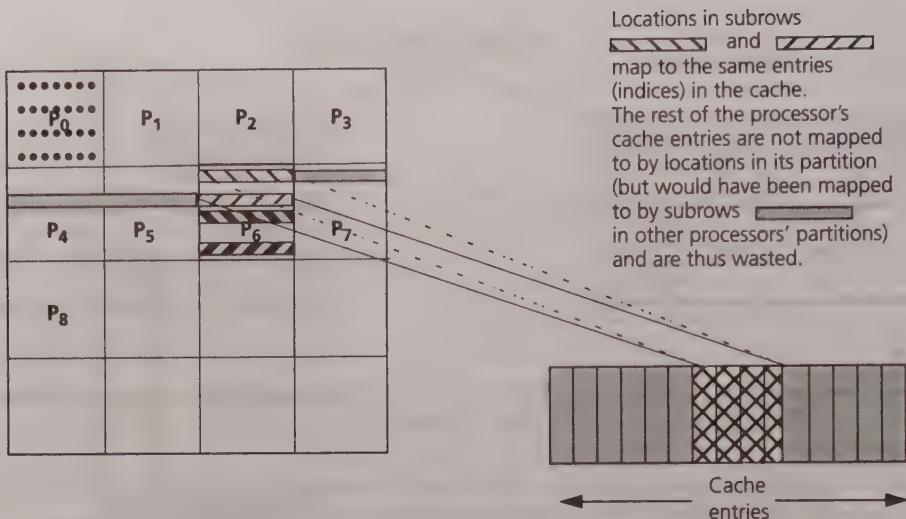


FIGURE 5.34 Cache mapping conflicts caused by a two-dimensional array representation in a direct-mapped cache. The figure shows the worst case, in which the separation between successive subrows in a process's partition (i.e., the size of a full row of the 2D array) is exactly equal to the size of the cache, so consecutive subrows map directly on top of one another in the cache. Every subrow accessed knocks the previous subrow out of the cache. In the next sweep over its partition, the processor will miss on every cache block it references, even if the cache as a whole is large enough to fit a whole partition. Many intermediately poor cases may be encountered depending on grid size, number of processors, and cache size. Since the cache size in bytes is a power of two, sizing the dimensions of allocated arrays to be powers of two is discouraged.

Challenge. The impact of conflicts and false sharing on uniprocessor and multiprocessor performance is clear.

- *Beware of conflict misses.* In illustrating conflict misses in the grid solver, Figure 5.34 shows how allocating power-of-two-sized arrays can cause pathological cache conflict problems since the cache size is also a power of two. Even if the logical size of the array that the application needs is a power of two, it is often useful to allocate a larger array that is not a power of two and then access only the amount needed. However, this strategy can interfere with allocating data at page granularity (also a power of two) in machines with physically distributed memory, so we may have to be careful. The cache mapping conflicts in this example are within a single data structure that is accessed in a predictable manner and can thus be alleviated in a structured way. Mapping conflicts are more difficult to avoid when they happen across different major data structures (e.g., across different grids used by the Ocean application), where they may have to be alleviated by ad hoc padding and alignment. However, in a shared address space they are particularly insidious when they occur on seemingly harmless shared variables or data structures that a programmer is not inclined to think about. For example, a frequently accessed pointer to an important data structure may conflict in a direct-mapped cache

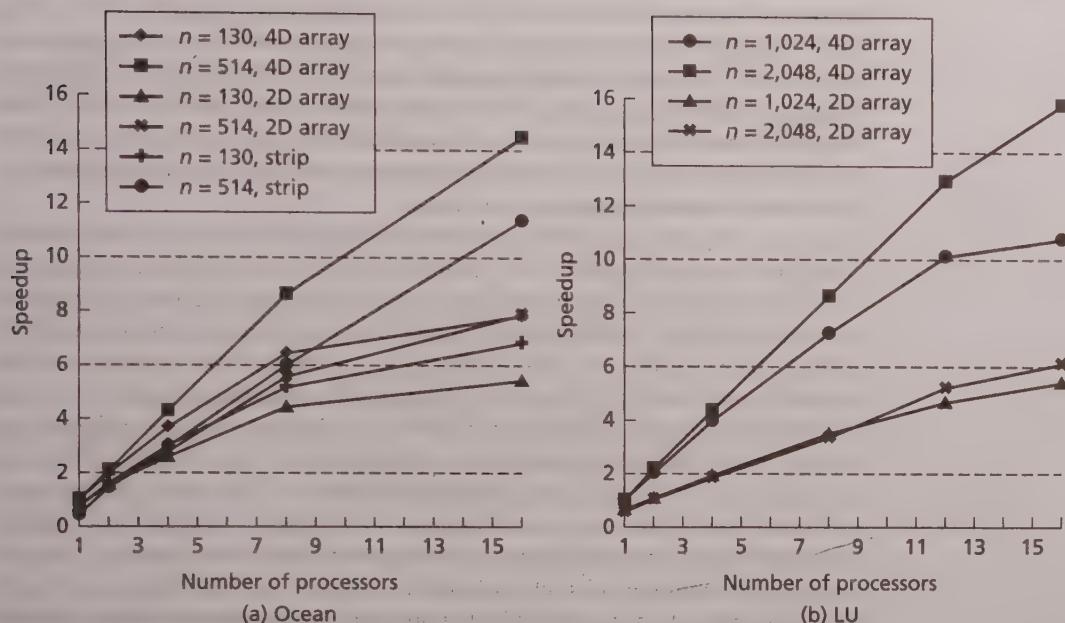


FIGURE 5.35 Performance impact of using 4D versus 2D arrays to represent two-dimensional grid or matrix data structures on the SGI Challenge. Results are shown for different problem sizes for the Ocean and LU applications. For Ocean, “strip” indicates partitioning into strips of contiguous rows (in which 2D or 4D arrays don’t matter), while all other cases assume partitioning into squarelike blocks.

with a scalar variable that is also frequently accessed during the same computation, causing a lot of traffic. Fortunately, such problems tend to be infrequent in modern (large and set-associative) second-level caches. In general, efforts to exploit locality can be wasted if attention is not paid to reducing conflict misses.

- **Use per-processor heaps.** It is desirable to have separate heap regions for each processor (or process) from which it allocates data dynamically. Otherwise, if a program performs a lot of very small memory allocations, data used by different processors may fall on the same cache block.
- **Copy data to increase spatial locality.** If a processor is going to reuse a set of data that is otherwise allocated noncontiguously in the address space, it is often desirable to make a contiguous copy of the data for that period to improve spatial locality and reduce cache conflicts. Copying requires memory accesses and has a cost, and it is not useful if the data is likely to reside in the cache anyway. For example, in blocked matrix factorization or multiplication, with a 2D array representation of the matrix a block is not contiguous in the address space (just like a partition in the equation solver kernel). However, a 2D representation makes programming easier. It is therefore not uncommon to use 2D

arrays and to copy blocks used from another processor's assigned set to a contiguous temporary data structure, during the time of active use, to reduce conflict misses. The cost of copying must be traded off against the benefit of reducing conflicts. In particle-based applications, when a particle moves from one processor's partition to another, spatial locality can be improved by moving the data for that particle so that the memory for all the particles assigned to a processor remains contiguous and dense.

- *Pad arrays.* Beginning parallel programmers often build arrays that are indexed using the process identifier. For example, to keep track of load balance, an array of p integers may be maintained, each entry of which records the number of tasks completed by the corresponding processor. Since many elements of such an array fall into a single cache block, and since these elements will be updated quite often by different processors, false sharing becomes a severe problem. One solution is to pad each entry with dummy words to make its size as large as the cache block size (or, to make the code more robust, as large as the largest cache block size on anticipated machines) and then align the array to a cache block. However, padding many large arrays can result in a significant waste of memory, and it can cause fragmentation in data transfer. A better strategy is to combine all such variables for a given process into a record, pad the entire record to a cache block boundary, and create an array of such records indexed by process identifier.
- *Determine how to organize arrays of records.* Suppose we have a number of logical records to represent, such as the particles in the Barnes-Hut gravitational simulation. Should we represent them as a single array of n particles, each entry being a record with fields like position, velocity, force, mass, and so on, as in Figure 5.36(a)? Or should we represent them as separate arrays of size n , one per field, as in Figure 5.36(b)? Programs written for vector machines such as traditional CRAY computers tend to use a separate array (vector) for each property or field of an object—in fact, even one per field per physical dimension (x , y , or z). When data is accessed by field, for example, the velocity of all particles, this increases the performance of vector operations by making accesses to memory unit stride and hence reducing memory bank conflicts. In cache-coherent multiprocessors, however, new trade-offs arise, and the best way to organize data depends on the access patterns.

An interesting tension is illustrated by the particle update and force calculation phases of the Barnes-Hut application. Consider the update phase first. A processor reads and writes only the position and velocity fields of all its assigned particles in this phase. However, its assigned particles are not contiguous in the shared particle array. Suppose there is one array of size n (number of particles) per field or property. A double-precision three-dimensional position (or velocity) is 24 bytes of data, so several of these may fit on a cache block. Since adjacent particles in the array may be read and written by different processors, false sharing can result. For this phase, it is better to have a single array of particle records, where each record holds all information about that particle; that is, to organize data by particle rather than by field.

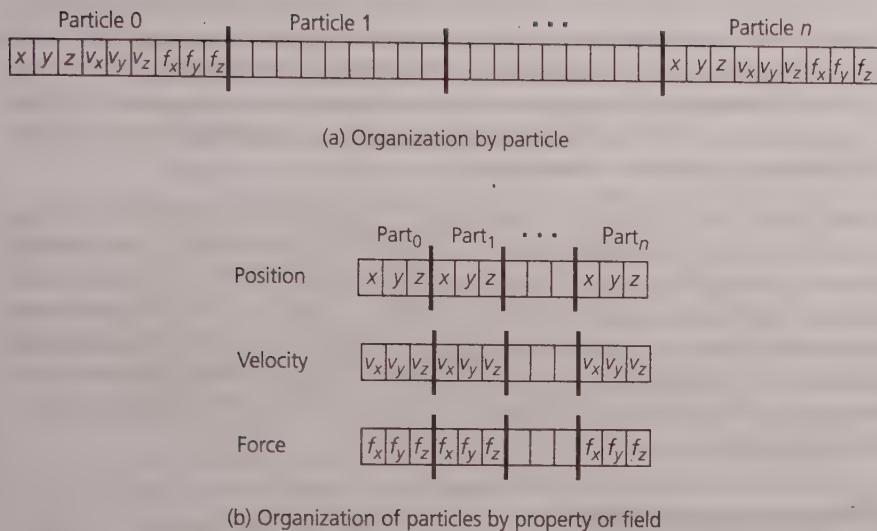


FIGURE 5.36 Alternative data structure organizations for record-based data

Now consider the force calculation phase of the same application. Suppose we use an organization by particle rather than by field as above. To compute the force on a particle, a processor reads the position values of many other particles and cells; it then updates the force components of its own particle. However, the force and position components of a particle may fall on the same cache block. In updating force components, it may therefore invalidate the position values of this particle from the caches of other processors that are using and reusing them as a result of false sharing within a particle record, even though the position values themselves are not being modified in this phase of computation. In this case, it would probably be better if we were to split the single array of particle records into two arrays of size n each, one for positions (and perhaps other properties) and one for forces. The entries of the force array themselves could be padded to reduce cross-particle false sharing. In general, it is often beneficial to split arrays of records to separate fields that are used in a read-only manner in a phase from the fields whose values are updated in the same phase. Different situations or phases may dictate different organizations for a data structure, and the ultimate decision depends on which pattern or phase dominates performance.

- **Align arrays.** In conjunction with the preceding techniques, it is often necessary to align arrays to cache block boundaries to achieve the full benefits. For example, given a cache block size of 64 bytes and 8-byte fields, we may have decided to maintain a single array of particle records with x , y , z , fx , fy , and fz . To avoid cross-particle false sharing, we pad each 48-byte record with two dummy 8-byte fields to fill a cache block. However, this wouldn't help if the

array started at an offset of 32 bytes from a page in the virtual address space, as this would mean that the data for each particle would now span two cache blocks, causing false sharing despite the padding. Even if a `malloc` call does not return data aligned to pages or blocks, alignment is easy to achieve by simply allocating a little extra memory through `malloc` and then suitably adjusting the starting address of the array.

As seen in the preceding list of techniques, the organization, alignment, and padding of data structures are all important for exploiting spatial locality and reducing false sharing and conflict misses. Experienced programmers and even some compilers use these techniques. As discussed in Chapter 3, these locality and artifactual communication issues can be more important to performance than inherent communication and can cause us to revisit our algorithmic partitioning decisions for an application (recall strip versus block partitioning for the simple equation solver as discussed in Section 3.1.2, and see Figure 5.35[a]).

5.7 CONCLUDING REMARKS

Symmetric shared memory multiprocessors are a natural extension of workstations and personal computers. A sequential application can run totally unchanged and yet benefit in performance by obtaining a larger fraction of a processor's time and by taking advantage of the large amount of shared main memory and I/O capacity typically available on such machines. Parallel applications are also relatively easy to bring up, as all shared data is directly accessible from all processors using ordinary loads and stores. Gradual parallelization is possible by selectively parallelizing computationally intensive portions of a sequential application, subject to the dictates of Amdahl's Law. For multiprogrammed workloads, a key advantage is the fine granularity at which resources can be shared among application processes and by the operating system, which can thus easily export a familiar, single-system image to each application. This is true both temporally, in that processors and/or main memory pages can frequently be reallocated among different application processes, and physically, in that main memory may be split among applications at the granularity of individual pages. Because of these appealing features, all major vendors of computer systems, from workstation suppliers like Sun, Silicon Graphics, Hewlett-Packard, Digital, and IBM to personal computer suppliers like Intel and Compaq, are producing and selling such machines. In fact, for some of the large workstation vendors, these multiprocessors constitute a substantial fraction of their revenue stream and a still larger fraction of their net profits because of the higher margins on these higher-end machines.

The key technical challenge in the design of symmetric multiprocessors is the organization and implementation of the shared memory system, which is used for communication between processors in addition to handling all regular memory accesses. Most small-scale parallel machines found today use the system bus as the interconnect for communication, and the challenge then becomes how to maintain coherency of the shared data in the private caches of the processors. A large variety

of options are available to the system architect, including the set of states associated with cache blocks, the bus transactions and actions used, the choice of cache block size, and whether updates or invalidations are used. The key task of the system architect is to make choices that will both perform well on the data sharing patterns expected in workloads and make the task of implementation easier. Another challenge is the design and implementation of efficient synchronization techniques that are both high performance and flexible.

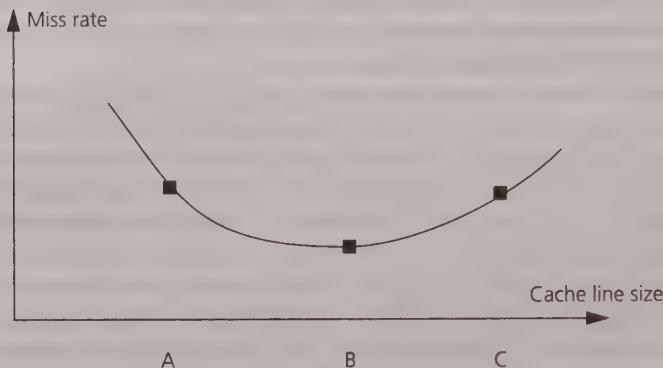
As processor, memory system, integrated circuit, and packaging technology continue to make rapid progress, questions arise about the future of small-scale multiprocessors and the importance of various design issues. We can expect small-scale multiprocessors to continue to be important for at least three reasons. The first is that they offer an attractive cost-performance combination. Individuals or small groups of people can easily afford them for use as a shared resource or as a compute or file server. Second, microprocessors today are designed to be multiprocessor-ready, and designers are aware of future microprocessor trends when they begin to design the next-generation multiprocessor, so there is no longer a significant time lag between the latest microprocessor and its incorporation in a multiprocessor. As we saw in Chapter 1, the Intel Pentium Pro processor line plugs “gluelessly” into a shared bus. The third reason is that the essential software technology for parallel machines (compilers, operating systems, programming languages) is maturing rapidly for small-scale shared memory machines. For example, most computer system vendors have efficient parallel versions of their operating systems ready for their bus-based multiprocessors. As levels of integration increase, multiple processors on a chip become attractive. While the optimal design points may change, the design issues that we have explored in this chapter are fundamental and will remain important with progress in technology.

This chapter has explored many of the key design aspects of bus-based multiprocessors at the “logical” level, involving cache block state transitions and complete (atomic) bus transactions. At this level, the design and implementation appears to be a rather simple extension of traditional cache controllers. However, much of the difficulty in such designs and many of the opportunities for optimization and innovation occur at the next lower level of protocol design and at the more detailed “physical” level. The next chapter goes down a level deeper into the design and organization of bus-based cache-coherent multiprocessors and some of their natural generalizations.

5.8 EXERCISES

- 5.1 Is the cache coherence problem an issue with processor registers? Given that registers are not kept consistent in hardware, how do current systems guarantee the desired semantics of a program?
- 5.2 Consider the following graph indicating the miss rate of an application as a function of cache block size on a multiprocessor. As might be expected, the curve has a U-shaped appearance. Consider the three points A, B, and C on the curve. Indicate

under what circumstances, if any, each may be a sensible operating point for the machine (i.e., the machine might give better performance at that point rather than at the other two points). How would you expect the shape and placement of the curve to differ for a uniprocessor?



- 5.3 Assume the following average data memory traffic for a bus-based shared memory multiprocessor: private reads—70%; private writes—20%; shared reads—8%; shared writes—2%. Also assume that 50% of the instructions (32 bits each) are either loads or stores. With a split instruction/data cache of 32-KB total size, we get hit rates of 97% for private data, 95% for shared data, and 98.5% for instructions. The cache line size is only 16 bytes.

We want to place as many processors as possible on a bus that has 64 data lines and 32 address lines. The processor clock is twice as fast as that of the bus, and the processor CPI is 2.0 before considering memory penalties. How many processors can the bus support without saturating if we use (a) write-through caches with write-allocate strategy? (b) write-back caches? Ignore cache consistency traffic and bus contention. The probability of having to replace a dirty block in the write-back caches on a miss that fetches a new block is 0.3. For reads, memory responds with data 2 cycles after being presented the address. For writes, both address and data are presented to memory at the same time. Assume that the bus is atomic and that processor miss penalties are equal to just the number of bus cycles required for each miss.

- 5.4 For each of the memory reference streams given in the following, compare the cost of executing it on a bus-based machine that supports (a) the Illinois MESI protocol and (b) the Dragon protocol. Explain the observed performance differences in terms of the characteristics of the streams and the coherence protocols.

stream 1: r1 w1 r1 w1 r2 w2 r2 w2 r3 w3 r3 w3

stream 2: r1 r2 r3 w1 w2 w3 r1 r2 r3 w3 w1

stream 3: r1 r2 r3 r3 w1 w1 w1 w1 w2 w3

All of the references in the streams are to the same location: r/w indicates read or write, and the digit refers to the processor issuing the reference. Assume that all

caches are initially empty, and use the following cost model: read/write cache hit—1 cycle; misses requiring simple transaction on bus (BusUpgr, BusUpd)—60 cycles; and misses requiring whole cache block transfer—90 cycles. Assume all caches are write allocated.

- 5.5 a. As miss latencies increase, does an update protocol become more or less preferable as compared to an invalidate protocol? Explain.
 b. In a multilevel cache hierarchy, would you propagate updates all the way to the first-level cache or only to the second-level cache? Explain the trade-offs.
 c. Why is update-based coherence not a good idea for multiprogramming workloads typically found on multiprocessor compute servers today?
 d. To provide an update protocol as an alternative, some machines have given control of the type of protocol to software at the granularity of page; that is, a given page can be kept coherent either using an update scheme or an invalidate scheme. An alternative to page-based control is to provide special opcodes for writes that will cause updates rather than invalidates. Comment on the advantages and disadvantages.
- 5.6 Given the following code segments, say what results are possible (or not possible) under sequential consistency (SC). Assume that all variables are initialized to 0 before this code is reached.

a.

P_1	P_2	P_3
$A = 1$	$u = A$	$v = B$
$B = 1$		$w = A$

b.

P_1	P_2	P_3	P_4
$A = 1$	$u = A$	$B = 1$	$w = B$
	$v = B$		$x = A$

- c. In the following sequence, first consider the operations within a dashed box to be part of the same instruction, say, a fetch&increment. Then, suppose they are separate instructions. Answer the questions for both cases.

P_1	P_2
$\boxed{u = A}$	$\boxed{v = A}$
$A = u + 1$	$A = v + 1$

- 5.7 a. Is the reordering problem due to write buffers, mentioned in Section 5.2.2, also a problem for concurrent programs on a uniprocessor? If so, how would you prevent it? If not, why not?
- b. Can a read complete before a previous write in program order issued by the same processor to the same location has completed (for example, if the write has been placed in the writer's write buffer but has not yet become visible to other processors) and still provide a coherent memory system? If so, what value should the read return? If not, why not? Can this be done and still guarantee SC?
- c. If we care only about coherence and not about sequential consistency, can we declare a write to be complete as soon as the processor is able to proceed past it?
- 5.8 Are the sufficient conditions for SC necessary? Make them less constraining (a) as much as possible and (b) in a reasonable intermediate way, and comment on the effects on implementation complexity.
- 5.9 Consider the following conditions proposed as sufficient conditions for SC:
- Every process issues memory requests in the order specified by the program.
 - After a read or write operation is issued, the issuing process waits for the operation to complete before issuing its next operation.
 - Before a processor P_j can return a value written by another processor P_i , all operations that were performed with respect to P_i before it issued the store must also be performed with respect to P_j .
- Are these conditions indeed sufficient to guarantee SC executions? If so, say why. If not, construct a counterexample, and say why the conditions that were listed in the chapter are indeed sufficient in that case. [Hint: think about in what way these conditions are different from the ones in the chapter.]
- 5.10 Consider a four-processor bus-based multiprocessor using the Illinois MESI protocol. Each processor executes a test&set lock to gain access to a null critical section. Assume the test&set instruction always goes on the bus and it takes the same time as a normal read transaction. The initial condition is such that processor 1 has the lock and processors 2, 3, and 4 are spinning on their caches waiting for the lock to be released. Every processor gets the lock once and then exits the program. Considering only the bus transactions related to lock-unlock operations:
- a. What is the least number of transactions executed to get from the initial to the final state?
 - b. What is the worst-case number of transactions?
 - c. Repeat parts (a) and (b) assuming the Dragon protocol.
- 5.11 What are the main advantages and disadvantages of exponential backoff in locks? Consider the test&set lock, the test-and-test&set lock, the ticket lock, and the array-based lock. How does the situation change if LL-SC is used instead of atomic instructions?

- 5.12 Suppose all 16 processors in a bus-based machine try to acquire a test-and-set lock simultaneously (and only once each). Assume all processors are spinning on the lock in their caches and are invalidated by a release at time 0.
- How many bus transactions will it take until all processors have acquired the lock if all the critical sections are empty (i.e., each processor simply does a `LOCK` and `UNLOCK` with nothing in between)?
 - Assuming that the bus is fair (services pending requests before new ones) and that every bus transaction takes 50 cycles, how long would it take before the first processor acquires and releases the lock? How long before the last processor to acquire the lock is able to acquire and release it?
 - What is the best you could do with an unfair bus, letting whatever processor you like win an arbitration regardless of the order of the requests?
 - Can you improve the performance by choosing a different (but fixed) bus arbitration scheme than a fair one?
 - If the variables used for implementing locks are not cached, will a test-and-set lock still generate less traffic than a test&set lock? Explain your answer.
- 5.13 For the same machine configuration as in Exercise 5.12(b) and assuming a fair bus, how many bus transactions and how much time is needed for the first and last processors to acquire and release the lock when using a ticket lock? Answer the same question for the array-based lock.
- 5.14 For the performance curves for the test&set lock with exponential backoff shown in Figure 5.29, why do you think the curve for the nonzero critical section is a little worse than the curve for the null critical section?
- 5.15
 - Why do we make the delay after an unlock d smaller than the size of the critical section c in our lock experiments? What problems might occur in measurement if we used d larger than c ? [Hint: draw timelines for two processor executions.]
 - How would you expect the results comparing lock algorithms to change if we used much larger values for c and d ?
- 5.16
 - Write pseudocode (high level plus assembly) to implement the ticket lock and array-based lock using (i) `fetch&increment`; (ii) LL-SC.
 - Suppose you did not have a `fetch&increment` primitive but only a `fetch & store` (a simple atomic exchange). Could you implement the array-based lock with this primitive? Describe the resulting lock algorithm.
- 5.17 Implement a `compare&swap` operation using LL-SC.
- 5.18 Consider the barrier algorithm with sense reversal that was described in Section 5.5.5. Would there be a problem if the `UNLOCK` statement were placed just after the increment of the counter rather than after each branch of the if condition? What would it be?

- 5.19 Suppose we have a machine that supports full-empty bits on every word in hardware. This particular machine allows for the following C code functions:

`ST_Special(loc, val)` writes `val` to data location `loc` and sets the full bit. If the full bit was already set, a trap is signaled.

`int LD_Special(loc)` waits until the data location's full bit is set, reads the data, clears the full bit, and returns the data as the result.

Write a C function `swap(i,j)` that uses these primitives to atomically swap the contents of two locations `A[i]` and `A[j]`. You should allow high concurrency (if multiple processors want to swap distinct pairs of locations, they should be able to do so concurrently) and you must avoid deadlock.

- 5.20 The `fetch&add` atomic operation can be used to implement barriers, semaphores, and other synchronization mechanisms. The semantics of `fetch-and-add` is such that it adds its second argument to the memory location in its first argument and returns the value of the memory location as it was before the addition. Use the `fetch-and-add` primitive to implement a barrier operation suitable for a shared memory multiprocessor. To use the barrier, a processor must execute `BARRIER(BAR, N)`, where `BAR` is the barrier name and `N` is the number of processes that need to arrive at the barrier before any of them can proceed. Assume that `N` has the same value in each use of barrier `BAR`. The barrier should be capable of supporting the following code:

```
while (condition) {
    Compute for a while
    BARRIER(BAR, N);
}
```

A proposed solution for implementing the barrier is the following:

```
BARRIER(Var B: BarVariable, N: integer)
{
    if (fetch-and-add(B, 1) = N-1) then
        B := 0;
    else
        while (B != 0) do {};
}
```

What is the problem with this code? Write the code for `BARRIER` in a way that avoids the problem.

- 5.21 Consider the following implementation of the `BARRIER` synchronization primitive, used at the end of each phase of computation of an application. Assume that `bar.releasing` and `bar.count` are initially zero and `bar.lock` is initially unlocked.

```
struct bar_struct {
    LOCKDEC(lock);
    int count, releasing;
```

```

} bar;
...
BARRIER(N)
{
    LOCK(bar.lock);
    bar.count++;

    if (bar.count == N) {
        bar.releasing = 1;
        bar.count--;
    } else {
        UNLOCK(bar.lock);
        while (! bar.releasing)
            ;
        LOCK(bar.lock);
        bar.count--;
        if (bar.count == 0) {
            bar.releasing = 0;
        }
    }
    UNLOCK(bar.lock);
}

```

- a. This code fails to provide a correct barrier. Describe the problem with this implementation.
- b. Change the code as little as possible so it provides a correct barrier implementation. Either clearly indicate your changes on the code or clearly describe the changes.
- 5.22 Consider migratory data: shared data objects that bounce around among processors, with each processor reading and then writing them before another processor reads them. Under the standard MESI protocol, the read miss and the write both generate bus transactions.
- Given the data in Table 5.1, estimate the maximum bandwidth that can be saved when using upgrades (BusUpgr) instead of BusRdX.
 - It is possible to enhance the state stored with cache blocks and the state transition diagram so that such read operations that are shortly followed by writes to the same block can be recognized and so that migratory blocks can be directly brought in exclusive state into the cache on the first read miss (rather than in shared state). Suggest the extra states and the state transition diagram extensions to achieve this. Using the data in Tables 5.1, 5.2, and 5.3, compute the bandwidth savings that can be achieved. Are there any benefits other than bandwidth savings? Describe program situations where the migratory protocol may hurt performance.

- 5.23 The Firefly update protocol eliminates the Sm state present in the Dragon protocol by suitably updating main memory on updates. Can we further reduce the states in the Dragon and/or Firefly protocols by merging the E and M states? What are the trade-offs?
- 5.24 It has been observed that processors sometimes write only one word in a cache block. To optimize for this case, instead of using write-back caches in all cases, a protocol has been proposed with the following characteristics: (1) on the initial write of a block, the processor writes through to the bus and places the block in the cache in a new state called the reserved state; and (2) on a write for a block that is present in the reserved state, the line transitions to the modified state, which uses write back instead of write through.
- Draw the state transitions for this protocol, using the INVALID, SHARED, RESERVED, and MODIFIED states. Be sure that you show an arc for each of BusRd, BusWr, ProcRd, and ProcWr for each state. Indicate the action that the processor takes after a slash (e.g., BusWr/WriteBlock). Since both word- and block-sized writes are used, indicate FlushWord or FlushBlock.
 - How does this protocol differ from the four-state Illinois protocol?
 - Describe concisely why you think this protocol is not used on a system like the SGI Challenge.
- 5.25 Consider the case when a processor writes a block that is shared by many processors (thus invalidating their caches). If the line is subsequently reread by the other processors, each will miss on the line. Researchers have proposed a read-broadcast scheme, in which if one processor reads the line, all other processors with invalid copies of the line read it into their second-level caches as well. Do you think this is a good protocol extension? Give at least two reasons to support your choice and at least one that argues the opposite.
- 5.26 Classify the misses in the following reference stream from three processors into the categories shown in Figure 5.20 (follow the format in Table 5.4). Assume that each processor's cache consists of only a single four-word cache block and that words w0 through w3 fall on the same cache block, as do words w4 through w7.

Operation Number	P ₁	P ₂	P ₃
1	st w0		st w7
2	ld w6	ld w2	
3		ld w7	
4	ld w2	ld w0	
5		st w2	
6	ld w2		
7	st w2	ld w5	ld w5
8	st w5		

Operation Number	P ₁	P ₂	P ₃
9		ld w3	ld w7
10		ld w6	ld w2
11		ld w2	st w7
12	ld w7		
13	ld w2		
14		ld w5	
15			ld w2

- 5.27 You are given a bus-based shared memory machine. Assume that the processors have a cache block size of 32 bytes and A is an array of four-byte integers. Now consider the following simple loop:

```
for i ← 0 to 16
    for j ← 0 to 255 {
        A[j] ← do_something(A[j]);
```

- a. Under what conditions would it be better to use a dynamically scheduled loop?
 - b. Under what conditions would it be better to use a statically scheduled loop?
 - c. For a dynamically scheduled inner loop, how many iterations should a processor pick each time?
- 5.28 You are writing an image processing program, where the image is represented as a 2D array of pixels. The basic iteration in this computation looks like

```
for i = 1 to 1024
    for j = 1 to 1024
        newA[i,j] = (A[i,j-1]+A[i-1,j]+A[i,j+1]+A[i+1,j])/4;
```

Assume A is a matrix of four-byte single-precision floating-point numbers stored in row-major order (i.e., A[i, j] and A[i, j+1] are at consecutive addresses in memory). A starts at memory location 0. You are writing this code for a 32-processor machine. Each processor has a 32-KB direct-mapped cache, and the cache block size is 64 bytes.

- a. You first try assigning 32 rows of the matrix to each processor in an interleaved assignment. What is the actual ratio of computation to bus traffic that you expect (inherent or artificial)? Assume that each loop iteration is four units of computation, ignore all other control and assignment operations, and state any other assumptions you use.
- b. Next you assign 32 contiguous rows of the matrix to each processor. Answer the question in part (a).

- c. Finally, you use a contiguous assignment of columns instead of rows. Answer the same question now.
- d. Suppose the matrix A started at memory location 32 rather than 0. If you use the same decomposition as in part (c), do you expect this to change the actual ratio of computation to traffic generated in the machine? If yes, will it increase or decrease, and why? If not, why not?
- 5.29 Consider the following simplified n -body code using an $O(N^2)$ algorithm (i.e., computing all pairwise interactions among bodies, here molecules). Estimate the number of misses per time-step in the steady state. Restructure the code using techniques discussed in the chapter to increase spatial locality and reduce false sharing. Try to make your restructuring robust with respect to the number of processors and cache block size. Assume 16 processors and 1-MB direct-mapped caches with a 64-byte block size. Estimate the number of misses for the restructured code. State all assumptions that you make.

```

typedef struct moltype {
    double x_pos, y_pos, z_pos; /*position components*/
    double x_vel, y_vel, z_vel; /*velocity components*/
    double x_f, y_f, z_f;      /*force components*/
} molecule;

#define numMols 4096
#define numProcs 16
molecule mol[numMols]

main()
{
    ...
    ... declarations ...
    for (time=0; time < endTime; time++)
        for (i=myPID; i < numMols; i+=numProcs)
        {
            for (j=0; j < numMols; j++)
            {
                x_f[i] += x_fn(position of mols i & j);
                y_f[i] += y_fn(position of mols i & j);
                z_f[i] += z_fn(position of mols i & j);
            }
            barrier(numProcs);
            for (i=myPID; i < numMols; i+= numProcs)
            {
                write velocity and position components
                of mol[i] based on force on mol[i];
            }
            barrier(numProcs);
        }
}

```

Snoop-Based Multiprocessor Design

The large differences we see in the performance, cost, and scale of symmetric multiprocessors on the market rest not so much on the choice of the cache coherence protocol but rather on the design and implementation of the organizational structure that supports the logical operation of the protocol. Protocol trade-offs are well understood, and most machines use a variant of the protocols described in the last chapter. However, the latency and bandwidth that is achieved with a protocol depend on the bus design, the cache design, and the integration with memory, as does the engineering cost of the system. This chapter examines the detailed physical design issues in snoop-based cache-coherent symmetric multiprocessors.

While the abstract state transition diagrams for coherence protocols that we saw in Chapter 5 are conceptually simple, subtle issues arise at the implementation level. An implementation must contend with at least three related goals: correctness, high performance, and minimal extra hardware. The correctness issues arise mainly because actions that are considered atomic at the abstract level are not necessarily atomic at the hardware level. The performance issues arise mainly because we want to pipeline memory operations and allow many operations to be outstanding at a time (using different components of the memory hierarchy) rather than waiting for each operation to complete before starting the next one. Unfortunately, it is in exactly these situations that correctness is likely to be compromised, due to the numerous complex interactions between these events. The product shipping dates for several commercial systems, even for microprocessors that have on-chip coherence controllers, have been delayed significantly because of subtle bugs in the coherence hardware. Overall, the design of modern communication assists (controllers) for aggressive cache-coherent multiprocessors presents a set of challenges similar in complexity and form to those of modern processor design, which also allows a large number of outstanding instructions and out-of-order execution. We need to peel off another layer in the design of snoop-based multiprocessors to understand the practical requirements embodied by state transition diagrams.

This chapter begins by enumerating the key correctness requirements for a cache-coherent memory system. A base design, using single-level caches and a one-transaction-at-a-time atomic bus, is developed in Section 6.2, and the critical events

in processing individual transactions are outlined. This section assumes an invalidation protocol for concreteness, but the main issues apply directly to update protocols as well. Section 6.3 expands this design to address multilevel cache hierarchies, showing how protocol events propagate up and down the hierarchy. Section 6.4 expands the base design to utilize a split-transaction bus. In such a bus, a bus transaction is split into request and response phases that arbitrate for the bus separately, so multiple transactions can be outstanding at a time on the bus and can be handled in a pipelined fashion. The section then brings together multilevel caches and split transactions. From this design point, it is a small step to support multiple outstanding misses from each processor since all transactions are already heavily pipelined and many take place concurrently. The fundamental underlying challenge throughout is maintaining the illusion of order as required by coherence and the memory consistency model. How this is done with each increasing level of design complexity is discussed in these sections.

Once we understand the key design issues in general terms, we will be ready to study concrete designs in some detail. Section 6.5 presents two case studies, the SGI Challenge and the Sun Enterprise, and illustrates their performance with micro-benchmarks and our sample applications. Finally, Section 6.6 examines a number of advanced topics that extend the design techniques in functionality and scale.

6.1 CORRECTNESS REQUIREMENTS

A cache-coherent memory system must, of course, satisfy the requirements of coherence and preserve the semantics dictated by the memory consistency model. In particular, for coherence it should ensure that stale copies are found and invalidated or updated on writes, and it should provide write serialization. If sequential consistency is to be preserved, it should provide write atomicity and the ability to detect the completion of writes. In addition, the design should have the desirable properties of any protocol implementation, which means it should be free of deadlock and livelock and should either eliminate starvation or make it very unlikely. Finally, it should cope with error conditions beyond its control (e.g., parity errors) and try to recover from them where possible.

Deadlock occurs when operations are still outstanding but all system activity has ceased. The potential for deadlock arises when multiple concurrent entities incrementally obtain shared resources and hold them in a nonpreemptible fashion, generating a cycle of resource dependences. A simple analogy is in traffic at an intersection, as shown in Figure 6.1. In the traffic example, the entities are cars and the resources are lanes. Each car needs to acquire two lane resources to proceed through the intersection, but each car is holding one and won't let it go.

In computer systems, the entities are typically controllers and the resources are buffers. For example, suppose two controllers *A* and *B* communicate with each other through buffers, as shown in Figure 6.2(a). *A*'s input buffer is full, and it refuses all incoming requests until *B* accepts a request from it (thus freeing up buffer space in *A* to accept requests from other controllers). But *B*'s input buffer is full too, and it

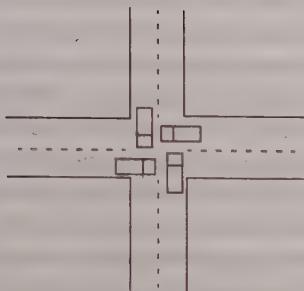


FIGURE 6.1 Deadlock at a traffic intersection. Four cars arrive at an intersection and all proceed one lane each into the intersection. They block one another since each is occupying a resource that another needs in order to make progress. Even if each decides to yield to the car on its right, the intersection is deadlocked. To break the deadlock, some cars must retreat to allow others to make progress so that they too can then make progress.

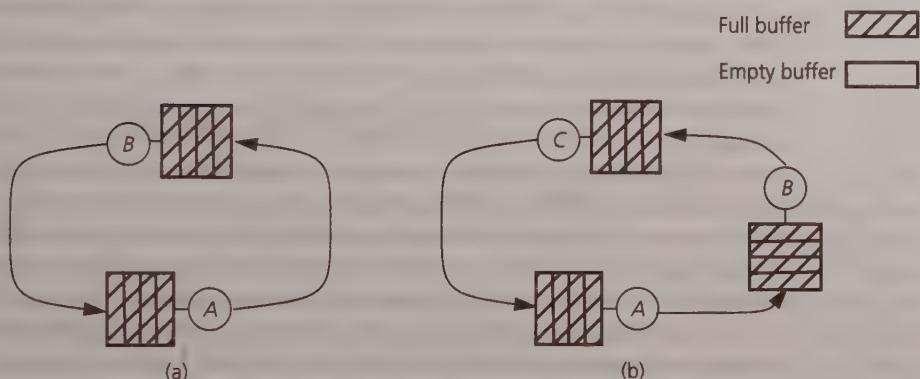


FIGURE 6.2 Deadlock in a computer system. Deadlock can easily occur in a system if independent controllers with finite buffering need to communicate with each other. If cycles are possible in the communication graph, then each controller can be stalled waiting for the one in front to free up resources. The figure illustrates cases for (a) two and (b) three controllers.

refuses all incoming requests until A accepts a request from it. Neither controller can accept a request, so deadlock sets in. To illustrate the problem with more than two controllers, a three-controller example is shown in Figure 6.2(b). To prevent deadlock, it is essential to either avoid such dependence cycles or break them when they occur.

A system is in *livelock* when no processor is making forward progress in its computation even though transactions are being executed in the system. Continuing the traffic analogy, each of the vehicles might elect to back up, clearing the intersection, and then try again to move forward. However, if they all repeatedly move backward and forward at the same time, there will be a lot of activity but they will end up in the same situation repeatedly with no real progress. In computer systems, livelock typically arises when independent controllers compete for a common resource, with each snatching it away from another before the other has finished with its use for the current operation.

Starvation does not stop overall progress, but is an extreme form of unfairness in which one or more processors make no progress while others continue to do so. For instance, in the traffic example, the livelock problem can be solved by a simple priority scheme. If a northbound car is given higher priority than an eastbound car, the latter must pull back and let the former through before trying to move forward again; similarly, a southbound car may have higher priority than a westbound car. Unfortunately, this does not solve starvation: in heavy traffic, an eastbound car may never pass the intersection since a new northbound car may always be ready to go through. Northbound cars make progress whereas eastbound cars are starved. A possible remedy here is to place an arbiter (e.g., a police officer or traffic light) to orchestrate the resource usage in a fair manner. The analogy extends easily to computer systems.

In general, the possibility of starvation is considered a less catastrophic problem than livelock or deadlock. Starvation does not cause the entire system to stop making progress and is usually not a permanent state. That is, just because a processor has been starved for some time in the past does not mean that it will be starved for all future time (at some point, northbound traffic will usually ease up and eastbound cars will get through). In fact, starvation is much less likely in computer systems than in this unmonitored traffic example, since it is usually timing dependent and the necessary pathological timing conditions usually do not persist. Starvation often turns out to be quite easy to eliminate in bus-based systems by having the bus arbitration be fair and using FIFO queues to access hardware resources. However, in scalable systems that we will see in later chapters, eliminating starvation completely can add substantial complexity to the protocols and can slow down common-case transactions. Many systems, therefore, do not completely eliminate starvation, though almost all try to reduce the potential for it to occur.

6.2 BASE DESIGN: SINGLE-LEVEL CACHES WITH AN ATOMIC BUS

In Chapter 5, we discussed how cache coherence protocols ensure write serialization and can satisfy the sufficient conditions for sequential consistency. We assumed that the bus was atomic, that operations from a given process were atomic with respect to one another, and that the memory operations that generate bus transactions were also atomic with respect to one another, from issue to completion even if they were from different processors. In this section, the assumptions are somewhat more physically realistic. There is still a single level of cache per processor, and transactions on the bus are atomic. The cache can stall the processor while it performs the series of steps involved in a memory operation, so operations within a process are atomic with respect to one another. However, no further assumptions are made. The section discusses the basic issues and trade-offs that arise in implementing snooping and state transitions in such a system, along with new issues that arise in providing write serialization, detecting write completion, and preserving write atomicity. Subsequent sections consider more aggressive systems, including more complex cache hierarchies and buses, as discussed earlier. In all cases, write-back caches are assumed, at least for the caches closest to the bus so they can reduce bus traffic.

Several design decisions must be made even for this simple case of single-level caches and an atomic bus. First, how should we design the cache tags and controller, given that both the processor and the snooping agent from the bus side need access to the tags? Second, the results of a snoop from the cache controllers need to be presented as part of the bus transaction; how and when should this be done? Third, even though the bus is atomic, the overall set of actions needed to satisfy a processor's memory operation uses other resources as well (such as cache controllers) and is not atomic, introducing possible race conditions. How should we design protocol state machines for the cache controllers given this lack of atomicity? What new issues arise with regard to write serialization, write completion detection, or write atomicity, as well as with regard to deadlock, livelock, and starvation? Finally, write backs from the caches can introduce interesting race conditions as well, and we must devise mechanisms to support atomic read-modify-write operations. We consider these issues one by one.

6.2.1 Cache Controller and Tag Design

Consider first a conventional uniprocessor cache. It consists of a storage array containing data blocks, tags, and state bits, as well as a comparator, a controller, and a bus interface. When the processor performs an operation against the cache, a portion of the address is used to access a cache set that potentially contains the block. The tag is compared against the remaining address bits to determine if the addressed block is indeed present. Then the appropriate operation is performed on the data and the state bits are updated. For example, a write hit to a clean cache block causes a word to be updated and the state to be set to modified. The cache controller sequences the reads and writes of the cache storage array. If the operation requires that a block be transferred from the cache to memory or vice versa, the cache controller initiates a bus operation. The bus operation requires the bus interface to perform a sequence of steps, which are typically the following: (1) assert request for bus, (2) wait for bus grant, (3) drive address and command, (4) wait for command to be accepted by the relevant device, and (5) transfer data. The sequence of actions taken by the cache controller is itself implemented as a finite state machine, as is the sequencing of steps in a bus transaction. It is important not to confuse these state machines with the state transition diagram of the protocol followed by each cache block.

To support a snooping coherence protocol, the basic uniprocessor cache controller design must be enhanced. First, since the cache controller must monitor bus operations as well as respond to processor operations, it is simplest to view the cache as having two controllers, a bus-side controller and a processor-side controller, each monitoring external events from its side. In either case, when an operation occurs the controller must access the cache tags. On every bus transaction, the bus-side controller must capture the address from the bus and use it to perform a tag check. If the check fails (a snoop miss), no action need be taken: the bus operation is irrelevant to this cache. If the snoop "hits," the controller may have to intervene in the bus transaction according to the cache coherence protocol. This may involve a

read-modify-write operation on the state bits or placing a block on the bus (or both).

With only a single array of tags, it is difficult to allow the two controllers to access the array at the same time. During a bus transaction, the processor will be locked out from accessing the cache, which will degrade processor performance. If the processor is given priority, effective bus bandwidth will decrease because the snoop controller will have to delay the bus transaction until it gains access to the tags. To alleviate this problem, a coherent cache design may utilize a *dual-ported* RAM for the tags and state or it may duplicate the tag and state for every block. The data portion of the cache is not duplicated since it is not accessed so frequently. If tags are duplicated, the contents of the two sets of tags are exactly the same, except that one is used by the processor-side controller for its lookups and the other is used by the bus-side controller for its snoops (see Figure 6.3). The two controllers can read the tags and perform checks simultaneously. Of course, when the state or tag for a block is updated (e.g., when the state changes on a write or a new block is brought into the cache) both copies must ultimately be modified, so one of the controllers may have to be locked out for a time. Machine designs can play several tricks to reduce the time for which a controller is locked out, for instance, in the above case by updating the processor-side tags only when the cache data is later modified rather than immediately when the bus-side tags are updated. The frequency of tag updates is also much smaller than that of tag lookups, so bus-side tag updates are expected to have little impact on processor cache access.

Another major enhancement from a uniprocessor cache controller is that the controller now acts not only as an initiator of bus transactions but also as a responder to them. A conventional responding device, such as the controller for a memory bank, monitors the bus for transactions on the fixed subset of addresses that it contains and possibly responds to the relevant read or write operations after some number of “wait” cycles. It may even have to place data on the bus. The cache controller behaves similarly, only it is not responsible for a fixed subset of addresses but must monitor the bus and perform a tag check on every transaction to determine if the transaction is relevant. For an update-based protocol, the controller may need to snoop the new data off the bus as well. Most modern microprocessors already implement such enhanced cache controllers so that they are “multiprocessor-ready.”

6.2.2 Reporting Snoop Results

Snooping introduces a new element to the bus transaction as well. In a conventional bus transaction on a uniprocessor system, one device (the initiator) places an address on the bus, all other devices monitor the address, and one device (the responder) recognizes it as being relevant. Then data is transferred between the two devices. The responder acknowledges its role by raising a wired-OR signal; if no device decides to respond within a time-out window, a bus error occurs. For snooping caches, each cache must check the address against its tags, and the collective result of the snoop from all caches must be reported on the bus before the transaction can proceed. In particular, one function of the snoop result is to inform main

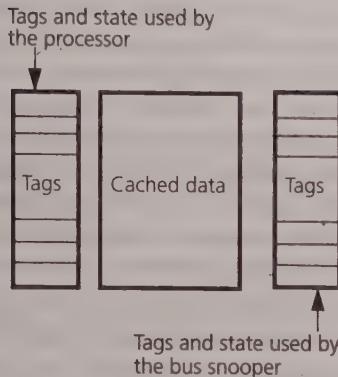


FIGURE 6.3 Organization of single-level snooping caches. For single-level caches, a duplicate set of tags and state are provided to reduce contention. One set is used exclusively by the processor while another is used by the bus snooper. Any changes to cache content or state, however, involve updating both sets of tags simultaneously.

memory whether it should respond to the request or whether some cache is holding a modified copy of the block so an alternative action is necessary. The questions are, When is the snoop result reported on the bus, and In what form?

Let us focus first on the “when” question. Obviously, it is desirable to keep the delay as small as possible so that main memory can decide quickly what to do.¹ The three major options are as follows:

1. The design could guarantee that the snoop results are available within a fixed number of clock cycles from the issue of the address on the bus. This, in general, requires the use of a dual set of tags because the processor, which usually has priority, could be accessing the tags heavily when the bus transaction appears. Even with a dual set of tags, we may need to be conservative about the fixed snoop latency because both sets of tags are made inaccessible when the processor updates the tags; for example, in the E → M state transition in the MESI protocol.² The advantages of this option are that the design of main memory is not affected, and the cache-to-cache handshake is very simple; the disadvantages are extra hardware and potentially longer snoop latency. The Pentium Pro quads use this approach, with the ability to extend or defer the snoop phase when necessary (see Chapter 8), as do the HP corporate business servers (Chan et al. 1993) and the Sun Enterprise.

-
1. Note that on an atomic bus there are ways to make the system less sensitive to the snoop delay. Since only one memory transaction can be outstanding at any given time, the main memory can start fetching the memory block regardless of whether it or the cache would eventually supply the data; the main memory subsystem would have to sit idle otherwise. Reducing this delay, however, is very important for a split-transaction bus, discussed later. There, multiple bus transactions can be outstanding, so the memory subsystem can be used in the meantime to service another request, for which it (and not the cache) may have to supply the data.
 2. It is interesting that in the base three-state invalidation protocol we described, a cache block state is never updated unless a corresponding bus transaction is also involved. This usually gives plenty of time to update the tags.

2. The design could alternatively support a variable delay snoop. The main memory assumes that one of the caches will supply the data until all the cache controllers have snooped and have indicated otherwise. A handshake is required, but cache controllers do not have to worry about tag-access conflicts inhibiting a timely lookup, and the designer does not have to conservatively assume the worst-case delay for snoop results. The SGI Challenge multiprocessors use a slight variant of this approach, where the memory subsystem fetches the data to service the request but then stalls if the snoops have not completed by that time (Galles and Williams 1993).
3. A third alternative is for the main memory subsystem to maintain a bit per block that indicates whether this block is modified in one of the caches or not. This way, the memory subsystem does not have to rely on snooping to decide what action to take. The disadvantage here is the extra complexity added to the main memory subsystem.

In what form should snoop results be reported on the bus? For the MESI scheme, the requesting cache controller needs to know whether the requested memory block is in other processors' caches so that it can decide whether to load the block in exclusive (E) or shared (S) state. In addition, the memory system needs to know whether any cache has the block in modified state, in which case the memory need not respond. One reasonable option is to use three wired-OR signals, two for reporting these aspects of the snoop results and one indicating that the snoop result is valid. The first signal is asserted when any of the processors' caches (excluding the requesting processor) has a copy of the block. The second is asserted if any cache has the block in modified state in its cache. We don't need to know the identity of that cache since it knows what action to take itself. The third signal is an inhibit signal, asserted until all caches have completed their snoop; when it is deasserted, the requestor and memory can safely examine the other two signals. The full Illinois version of the MESI protocol is more complex because a block can be preferentially retrieved from another cache rather than from memory even if it is in shared state. If multiple caches have a copy, a priority mechanism is needed to decide which cache will supply the data. This is one reason why most commercial machines that use the MESI protocol limit cache-to-cache transfers. The Silicon Graphics Challenge and the Sun Enterprise use cache-to-cache transfers only for data that is in modified state in a cache, in which case there is a single supplier. The Challenge updates memory in the process of a cache-to-cache transfer, whereas the Enterprise does not update memory and uses the fifth, owned state of the MOESI protocol, as discussed in Chapter 5.

6.2.3 Dealing with Write Backs

Write backs complicate implementation since they involve an incoming block as well as an outgoing (modified) block that is being replaced, and hence two bus transactions. In general, to allow the processor to continue as soon as possible on a cache miss that causes a write back, we would like to delay the write back and

instead first service the miss that caused it. This optimization imposes two requirements. First, it requires the machine to provide additional storage, a *write-back buffer*, where the block being replaced can be temporarily stored while the new block is brought into the cache and before the bus can be reacquired for a second transaction to complete the write back. Second, before the write back is completed, it is possible that we will see a bus transaction containing the address of the block being written back. In that case, the controller must supply the data from the write-back buffer and cancel its earlier pending request to the bus for a write back. This requires that an address comparator be added to snoop on the write-back buffer as well. We see in Chapter 8 that write backs introduce further correctness subtleties in machines with physically distributed memory.

6.2.4 Base Organization

Figure 6.4 shows a block diagram for our resulting base snooping architecture. Each processor has a single-level write-back cache. The cache is dual tagged so the bus-side controller and the processor-side controller can do tag checks in parallel. The processor-side controller initiates a transaction by placing an address and command on the bus. On a write-back transaction, data is conveyed from the write-back buffer. On a read transaction, it is captured in the data buffer. The bus-side controller snoops the write-back tag as well as the cache tags. Bus arbitration places the requests that go on the bus in a total order. For each transaction, the command and address in the request phase drive the snoop lookups in this total order. The wired-OR snoop results serve as acknowledgment to the initiator that all caches have seen the request and taken relevant action.

Using this simple design, let us examine more subtle correctness concerns that either require the state machines and protocols to be extended or require care in implementation. These include nonatomic state transitions, serialization for coherence and consistency, deadlock, livelock, and starvation.

6.2.5 Nonatomic State Transitions

In the state transition diagrams in Chapter 5, the state transitions and their associated actions were assumed to happen instantaneously or at least atomically. In fact, a request issued by a processor takes some time to complete, often including a bus transaction. While the bus transaction itself is atomic in our simple system, it is only one among the set of actions needed to satisfy a processor's request. These actions include looking up the cache tags, arbitrating for the bus, actions taken by other controllers at their caches, and the action taken by the issuing processor's controller at the end of the bus transaction (which may include actually writing data into the block). Taken as a whole, the set is not atomic. Even with an atomic bus, multiple requests from different processors may be outstanding in different parts of the system at a time, and it is possible that while a processor (or controller) P has a request outstanding—for example, waiting to obtain bus access—a request from another processor may appear on the bus and need some service from P, perhaps even for the

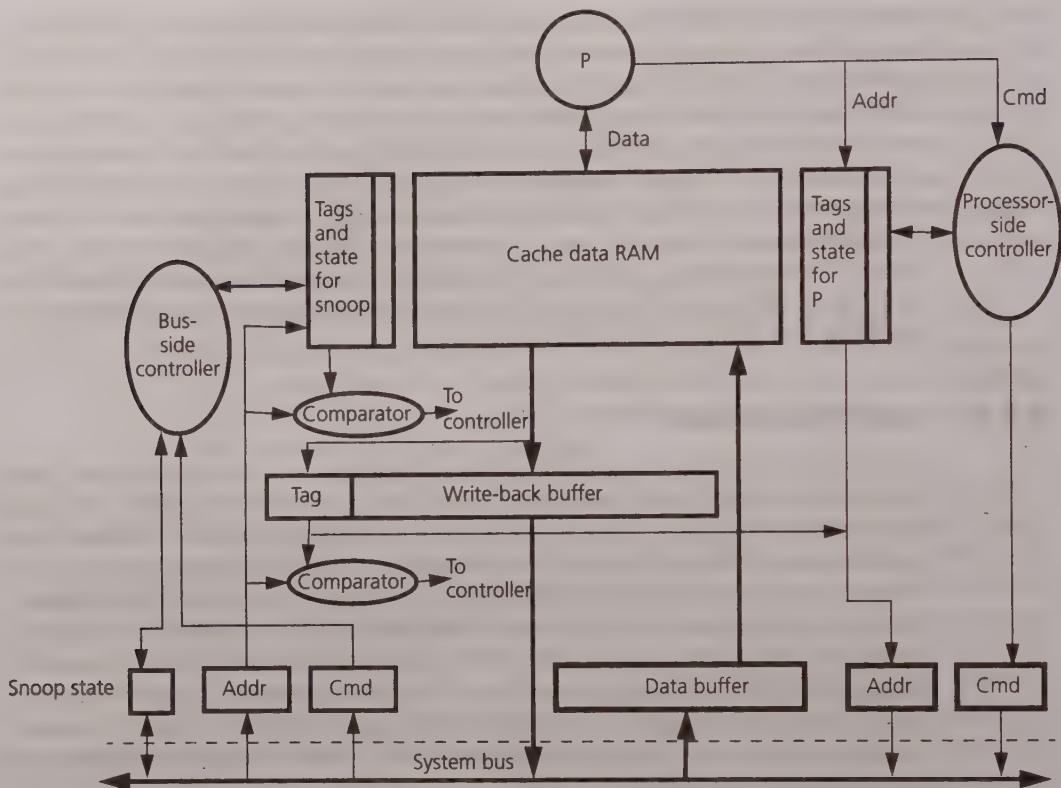


FIGURE 6.4 Design of a snooping cache for the base machine. We assume that each processor has a single-level write-back cache, an invalidation protocol is used, the processor can have only one memory request outstanding, and the system bus is atomic. To keep the figure simple, we do not show the bus arbitration logic and some of the low-level signals and buffers that are needed. We also do not show the coordination signals needed between the bus-side controller and the processor-side controller.

same memory block as P's outstanding request. The types of complications that arise are illustrated in Example 6.1.

EXAMPLE 6.1 Suppose two processors P₁ and P₂ cache the same memory block A in shared state, and both simultaneously issue a write to block A. Show how P₁ may have a request outstanding waiting for the bus while a transaction from P₂ appears on the bus and how you might solve the complication that results.

Answer Here is a possible scenario. P₁'s write will check its cache, determine that it needs to promote the block's state from shared to modified before it can actually write new data into the block, and issue an upgrade bus request. In the meantime, P₂ has also issued a similar upgrade or read-exclusive transaction for A, and it may have won arbitration for the bus first. P₁'s controller will see the bus transaction and must downgrade the state of block A from shared to invalid in its cache. Otherwise, when P₂'s transaction is over, A will be in modified state in P₂'s cache.

and in shared state in P_1 's cache, which violates the protocol. But now the upgrade bus request that has P_1 outstanding is no longer appropriate and must be replaced with a read-exclusive request. Thus, a controller must also be able to check addresses snooped from the bus against its own outstanding request and modify the latter if necessary. (If there were no upgrade transactions in the protocol and read exclusives were used even on writes to blocks in shared state, the request would not have to be changed in this case even though the block state would have to be changed. These implementation requirements should therefore be considered when assessing the complexity of protocol optimizations.) ■

A convenient way to deal with the “nonatomic” nature of state transitions, and the consequent need to sometimes revise requests and actions based on observed events, is to expand the protocol state diagram with intermediate or *transient* states (the original protocol states that we have been discussing so far, such as MESI, will be referred to as *stable* states). For example, a separate state can be used to indicate that an upgrade request is outstanding. Figure 6.5 shows an expanded state diagram for a MESI protocol. In response to a processor write operation, for example, the cache controller begins arbitration for the bus by asserting a request for the bus (BusReq) and transitions to the intermediate $S \rightarrow M$ state. The transition out of this state occurs when the bus arbiter asserts a BusGrant signal for this device. At this point, the BusUpgr transaction is placed on the bus and the cache block state is updated. However, if a BusRdX or BusUpgr is observed on the bus for this block while in the $S \rightarrow M$ state, the controller treats its block as having been invalidated before this transaction and transitions to the $I \rightarrow M$ state. (We could instead retract the bus request and transition to the I state, whereupon the still pending PrWr would be handled again.) On a processor read from invalid state, the controller advances to an intermediate state ($I \rightarrow S, E$); the next stable state to transition to is determined by the value of the shared line when the read is granted the bus. These intermediate states are not typically encoded in the cache block state bits, which are still the stable MESI states, since it would be wasteful to expend bits in every cache slot to indicate the one block in the cache that may be in a transient state. They are reflected in the combination of state bits and controller state. However, when we consider caches that allow multiple outstanding transactions, it will be necessary to have an explicit representation for the (multiple) blocks from a cache that may be in a transient state.

Expanding the number of states in the protocol increases the difficulty of proving that an implementation is correct or of testing the design. Thus, designers seek mechanisms that avoid transient states. The Sun Enterprise, for example, does not use a BusUpgr transaction in the MESI protocol but uses the result of the snoop to eliminate unnecessary data transfers in the BusRdX. Recall that on a BusRdX the caches holding the block invalidate their copy. If a cache has the block in the modified state, it raises the dirty line, thereby preventing the memory from supplying the data, and flushes the data onto the bus. No use is made of the shared line. The trick is to have the processor that issues the BusRdX snoop its own tags when the transaction actually goes on the bus. If the block is still in its cache in a valid state, it raises the shared line, which inhibits main memory. Since it already has the valid block, no cache can have it in modified state, and the data phase of the transaction is ignored.

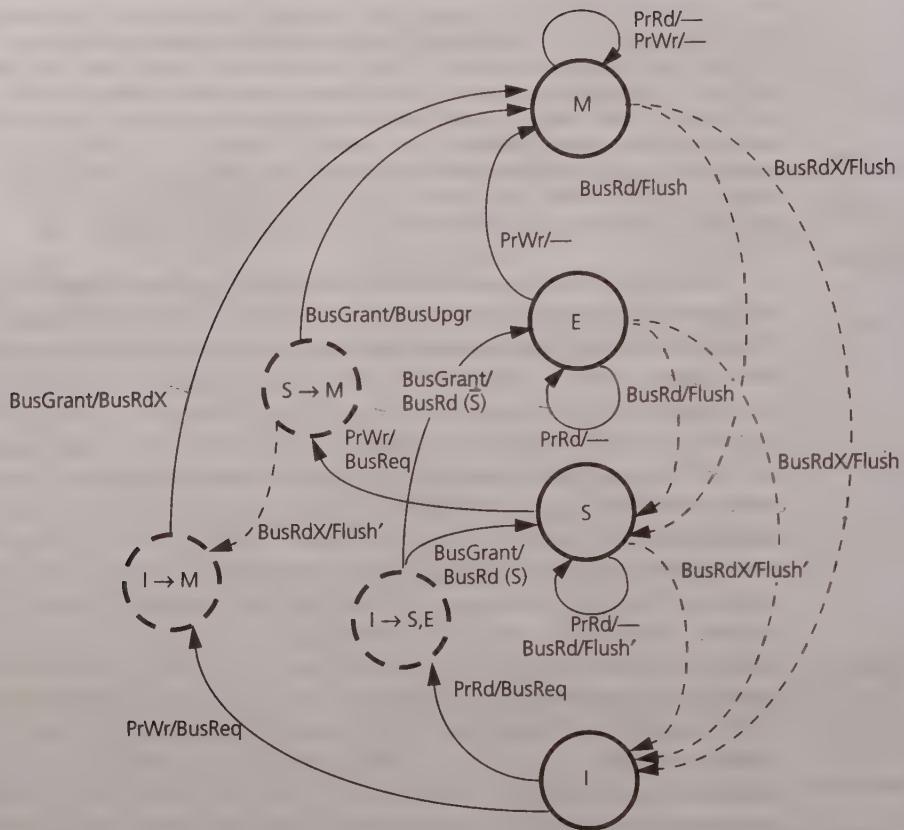


FIGURE 6.5 Expanded MESI protocol state diagram indicating transient states for bus acquisition. The cache controller monitors the bus while arbitration is ongoing for its request. A conflicting transaction may change the transition between stable states.

The cache controller does not need a transient state because, regardless of what happens, it has one action to take—place a BusRdX transaction on the bus.

6.2.6 Serialization

With the nonatomicity of memory operations issued by different processors, care must be taken in the processor-cache handshake to preserve the order determined by the serialization of bus transactions. For a read, the processor needs the result of the operation. To gain greater performance on writes, it is tempting to update the cache block and allow the processor to continue with useful instructions while the cache controller acquires exclusive ownership of the block—and possibly loads the rest of the block—via a bus transaction. The problem is that a window is open between the

time the processor gives the write to the cache and the time the cache controller acquires the bus for the read-exclusive (or upgrade) transaction. As we have seen, other bus transactions (including writes) may occur in this window, which may change the state of this or other blocks in the cache. This can complicate write serialization for coherence (if the transactions are to the same block) as well as SC (if they are to other blocks). To provide write serialization or SC, these transactions must appear to the processor as occurring before the write since that is how they are serialized by the bus and appear to other processors. Conservatively, the cache controller should not allow the processor issuing the write to consider the write complete and to complete other operations past it in program order until the read-exclusive transaction occurs on the bus and makes the write visible to other processors.

In fact, the cache does not have to wait until the read-exclusive transaction is finished—that is, until other copies have actually been invalidated in their caches—before allowing the processor to continue; it can service read and write hits once the transaction is on the bus, as long as access to the block in transit is handled properly. The crux of the argument for coherence and for sequential consistency presented in Section 5.3 was that all cache controllers observe the exclusive ownership transactions (BusRdX or BusUpgr) generated by write operations in the same order and that the data is written in the cache immediately after the exclusive ownership transaction. Once the bus transaction starts, in our base design the writer knows that all other caches will invalidate their copies before another bus transaction occurs. The write is *committed*, that is, the position of the write in the serial bus order is completely determined, regardless of further actions. The writer never knows exactly where the invalidation is inserted in the local program order of the other processors; it knows only that it is before whatever operation generates the next bus transaction and that all processors insert the invalidations in the same order. Similarly, the writer's subsequent local sequence of cache hits only becomes visible at the next bus transaction. This is all that is important to maintain the necessary orderings for coherence and SC, and it allows the writer to substitute commitment for actual completion in following the sufficient conditions for SC. In fact, this basic observation is what makes it possible to implement cache coherence and sequential consistency with pipelined buses, multilevel memory hierarchies, and multiple outstanding operations per processor. Write atomicity follows the same argument as presented before in Section 5.3.

This discussion of serialization raises an important but somewhat subtle point. Write serialization and write atomicity have very little to do with when the transactions that write data back to memory occur or with when the actual location in memory is updated. Either a write or a read can cause a write back if it causes a dirty block to be replaced. The write backs are bus transactions, but they do not need to be ordered. On the other hand, a write does not necessarily cause the new value to appear on the bus, even if it misses; it causes a read exclusive. What is important to the program is when the new value is bound to the address. The write completes, in the sense that any subsequent read will return the new or later value once the BusRdX or BusUpgr transaction takes place. By invalidating the old cache blocks, it

ensures that all reads that returned the old value precede the transaction. The controller issuing the transaction ensures that the new value is written in the cache after the bus transaction and that no other memory operations intervene.

6.2.7 Deadlock

A two-phase protocol, such as the request-response protocol of a memory operation, presents a form of protocol-level deadlock, sometimes called *fetch deadlock* (Leiserson et al. 1996), that is not simply a question of buffer usage. While an entity is attempting to issue its request, it needs to service incoming transactions. In an SMP with an atomic bus, this situation arises when the cache controller is awaiting the bus grant: it needs to continue performing snoops and handling requests, which may require it to flush blocks onto the bus. Otherwise, the system may deadlock if each of two controllers has an outstanding transaction that the other needs to respond to, and both are refusing to handle requests. For example, suppose a BusRd for a block *B* appears on the bus while a processor P_1 has a read-exclusive request outstanding to another block *A* and is waiting for the bus. If P_1 has a modified copy of *B*, its controller should be able to supply the data to the current bus transaction (which does not require bus arbitration with an atomic bus) and change the state from modified to shared while it is waiting to acquire the bus. Otherwise, the current bus transaction is waiting for P_1 's controller while P_1 's controller is waiting for the bus transaction to release the bus.

6.2.8 Livelock and Starvation

The classic potential livelock problem in an invalidation-based cache-coherent memory system is caused by all processors attempting to write to the same memory location at about the same time. Suppose that, initially, no processor has a copy of the location in its cache. A processor's write requires the following nonatomic set of events: its cache obtains exclusive ownership for the corresponding memory block (i.e., it invalidates other copies and obtains the block in modified state); a state machine in the processor realizes that the block is now present in the cache in the appropriate state; and the state machine reattempts the write. Unless the processor-cache handshake is designed carefully, it is possible that the block is brought into the cache in modified state, but before the processor is able to complete its write, the block is invalidated by a BusRdX request from another processor. The processor's write attempt misses again, and the cycle can repeat indefinitely. To avoid livelock, a write that has obtained exclusive ownership must be allowed to complete before the exclusive ownership is taken away.

With multiple processors competing for a bus, it is possible that some processors may be granted the bus repeatedly while others may not and may become starved. Starvation can be avoided by using first-come-first-served service policies at the bus arbiter and elsewhere. These usually require additional buffering, however, so sometimes heuristic techniques are used to reduce the likelihood of starvation. For exam-

ple, a count can be maintained of the number of times that a request has been denied and, after a certain threshold, action is taken so that no other new request is serviced until this request is serviced, or the request's priority may be increased.

6.2.9 Implementing Atomic Operations

The last implementation aspect that we should understand for the base architecture before moving on to more realistic architectures is the implementation of atomic read-modify-write instructions, such as `test&set` and `fetch&rop`, and the LL-SC primitives that can synthesize atomic operations (see Section 5.5).

Consider a simple `test&set` instruction. It has a read component (the test) and a write component (the set). The first question is whether the `test&set` (lock) variable should be cacheable so the `test&set` can be performed in the processor cache or uncacheable so the atomic operation is performed at main memory. The discussion of synchronization in Section 5.5 assumed cacheable lock variables. This has the advantage of allowing locality to be exploited and, hence, reducing latency and traffic when the lock is repeatedly acquired by the same processor: the lock variable remains in modified state in the cache and no invalidations or misses are generated. It also allows processors to spin in their caches, thus reducing useless bus traffic when the lock is not ready. However, performing the operations at memory can cause faster transfer of a lock from one processor to another. With cacheable locks, the processor that is busy-waiting will first be invalidated, at which point it will try to access the lock from the other processor's cache or from main memory. With uncached locks, the release goes only to memory (no invalidations are needed), and by the time it gets there the next busy-waiting read by the waiting processor is likely to be on its way to memory already, so it will obtain the lock from memory with low latency. Overall, traffic and locality considerations tend to dominate, and lock variables are usually cacheable so that processors can busy-wait without loading the bus.

A conceptually natural way to implement a cacheable `test&set` that is not satisfied in the cache itself is with two bus transactions: a read transaction for the test component and a write transaction for the set component. One strategy to keep this sequence atomic is to lock down the bus at the read transaction until the write completes, keeping other processors from putting accesses (especially to that variable) on the bus between the read and write components. While this can be done quite easily with an atomic bus, it is much more difficult with a split-transaction bus: not only does locking down the bus impact performance substantially but it can raise deadlock complications if one of the transactions cannot immediately be satisfied without giving up the bus.

Fortunately, better approaches are available. Consider an invalidation-based protocol with write-back caches. What a processor really needs to do is obtain exclusive ownership of the cache block (e.g., by issuing a single read-exclusive bus transaction), and then it can perform the read component and the write component in the cache as long as it does not give up exclusive ownership of the block in between; that is, even on a nonatomic bus, incoming accesses from the bus to that block

would be buffered and hence delayed until the data is written in the cache. More complex atomic operations, such as `fetch&rop`, must retain exclusive ownership until the operation is completed.

An atomic instruction that is more complex to implement is `compare&swap`. It requires specifying three operands in a memory instruction: the memory location, the register to compare with, and the value/register to be swapped with the memory location. RISC instruction sets are usually not equipped for this.

Implementing LL-SC requires a little special support. A typical implementation uses a hardware lock flag and a lock address register at each processor. An LL operation reads the block but also sets the lock flag and puts the address of the block in the lock address register. Incoming invalidation (or update) requests from the bus are matched against the lock address register, and a successful match (called a conflicting write) resets the lock flag. A store-conditional checks the lock flag as the indicator for whether an intervening conflicting write has occurred; if the flag has been reset, it fails, and if not, it succeeds. The lock flag is also reset (and the store-conditional will fail) if the lock variable is replaced from the cache, since then the processor may no longer see invalidations or updates to that variable. Finally, the lock flag is reset at context switches since a context switch between an LL and its store-conditional may incorrectly cause the LL of the old process to lead to the success of a store-conditional in the new process that is switched in.

Some subtle issues arise in avoiding livelock when implementing LL-SC. First, we should in fact not allow replacement of the cache block that holds the lock variable to occur between the LL and the store-conditional. Replacement would clear the lock flag and could establish a situation in which a processor keeps trying the store-conditional but never succeeds because of continual replacement of the block between repeated LL and store-conditional operations. To disallow replacements due to conflicts with instruction fetches, we can use split instruction and data caches or set-associative unified caches. For conflicts with other data references, a common solution is to simply disallow memory-referencing instructions between an LL and a store-conditional. Techniques to hide latency (e.g., out-of-order issue) can complicate matters since memory operations that are not between the LL and the store-conditional in the program code may be between LL and store-conditional in the execution. A simple solution to this problem is not to allow reorderings of memory operations across LL or store-conditional operations.

The second potential livelock situation would occur if two processes continually failed on their store-conditionals and each process's failing store-conditional invalidated or updated the other process's block, thus clearing the lock flag. Neither of the two processes would ever succeed if this pathological situation persisted. This is why it is important that a store-conditional not be treated as an ordinary write and that it not issue invalidations or updates when it fails.

Compared to implementing an atomic read-modify-write instruction, LL-SC can have a performance disadvantage since both the LL and the store-conditional can miss in the cache even when they are successful, if the LL loads the block in shared state, leading to two misses instead of one. For better performance, it may be desirable to obtain (or prefetch) the block in exclusive or modified state at the LL so that

the store-conditional does not miss unless it fails. However, this reintroduces the second livelock situation: other copies are invalidated to obtain exclusive ownership, so their store-conditionals may fail without guarantee of this processor's store-conditional succeeding. If this optimization is employed, some form of backoff should be used between failed operations to minimize (though not completely eliminate) the probability of livelock.

6.3 MULTILEVEL CACHE HIERARCHIES

The simple design presented in the preceding section was illustrative, but it made two simplifying assumptions that are not valid on most modern systems: single-level caches and an atomic bus. This section relaxes the first assumption and examines the resulting design issues.

The trend in microprocessor design since the early 1990s has been to have an on-chip first-level cache and a much larger second-level cache, either on chip or off chip.³ Many systems use on-chip secondary caches as well and an off-chip tertiary cache. Multilevel cache hierarchies would seem to complicate coherence since changes made by the processor to the first-level cache may not be visible to the lower-level cache controller responsible for bus operations, and bus transactions are not directly visible to the first-level cache. However, the basic mechanisms for cache coherence extend naturally to multilevel cache hierarchies. Let us consider a two-level hierarchy, as shown in Figure 6.6, for concreteness; the extension to the multi-level case is straightforward.

One obvious way to handle multilevel caches is to have independent bus snooping hardware for each level of the cache hierarchy. This is unattractive for several reasons. First, the L₁ cache is usually on the processor chip, and an on-chip snooper will consume precious pins to monitor the addresses on the shared bus. Second, duplicating the tags to allow concurrent access by the snooper and the processor may consume too much precious on-chip real estate. Third, duplication of effort occurs between the L₂ and L₁ snoops since, most of the time, blocks present in the L₁ cache are also present in the L₂ cache; therefore, the snoop of the L₁ cache is unnecessary.

The solution used in practice is based on this last observation. When using multilevel caches, designers ensure that they preserve the *inclusion property*, which requires the following:

1. If a memory block is in the L₁ cache, then it must also be present in the L₂ cache. In other words, the contents of the L₁ cache must be a subset of the contents of the L₂ cache.
2. If the block is in an owned state (e.g., modified in MESI or MOESI, shared-modified in Dragon or owned in MOESI) in the L₁ cache, then it must also be marked modified in the L₂ cache.

3. The HP PA-RISC microprocessors are a notable exception, maintaining a large off-chip first-level cache for many years after other vendors went to small on-chip first-level caches.

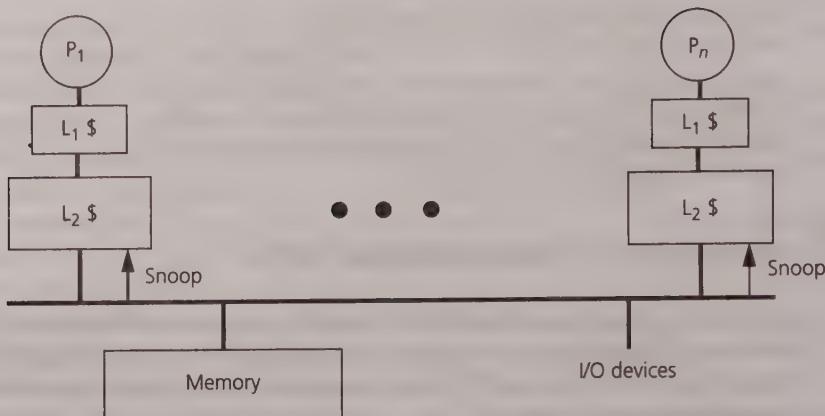


FIGURE 6.6 A bus-based machine containing processors with two-level caches

The first requirement ensures that all bus transactions that are relevant to the L_1 cache are also relevant to the L_2 cache, so having the L_2 cache controller snoop the bus is sufficient. The second ensures that if a bus transaction requests a block that is in modified state in the L_1 or L_2 cache, then the L_2 snoop can determine this fact on its own.

6.3.1 Maintaining Inclusion

The requirements for inclusion are not trivial to maintain. Three aspects need to be considered. First, processor references to the L_1 cache cause it to change state and perform replacements; these need to be handled in a manner that maintains inclusion. Second, bus transactions cause the L_2 cache to change state and flush blocks; these need to be forwarded to the first level. Finally, the modified state must be propagated out to the L_2 cache.

At first glance, it might appear that inclusion would be satisfied automatically since all L_1 cache misses go to the L_2 cache. The problem, however, is that two caches may choose different blocks or data to replace on a miss. Inclusion falls out automatically only for certain combinations of cache configuration. It is an interesting exercise to see what conditions in typical cache hierarchies can cause inclusion to be violated if no special care is taken (Baer and Wang 1988). Let us consider this before we look at how inclusion is typically maintained. For notational purposes, assume that the L_1 cache has associativity a_1 , number of sets n_1 , block size b_1 , and thus a total capacity of $S_1 = a_1 \times b_1 \times n_1$. The corresponding parameters for the L_2 cache are a_2 , n_2 , b_2 , and S_2 . We also assume that all parameter values are powers of two.

- Set-associative L_1 caches with history-based replacement. The problem with replacement policies based on the history of accesses to a block, such as least

recently used (LRU) replacement, is that the L_1 cache sees a different history of accesses than L_2 and other caches, since all processor references look up the L_1 cache but not all get to lower-level caches. Suppose the L_1 cache is two-way set associative with LRU replacement, both L_1 and L_2 caches have the same block size ($b_1 = b_2$), and L_2 is k times larger than L_1 ($n_2 = k \times n_1$). It is easy to show that inclusion does not hold in this simple case. Consider three distinct memory blocks m_1 , m_2 , and m_3 that map to the same set in the L_1 cache. Assume that m_1 and m_2 are currently in the two available slots within that set in the L_1 cache and are present in the L_2 cache as well. Now consider what happens when the processor references m_3 , which happens to collide with and replace one of m_1 and m_2 in the L_2 cache as well. Since the L_2 cache is oblivious to the L_1 cache's access history, which determines whether the latter replaces m_1 or m_2 , it is easy to see that the L_2 cache may replace one of m_1 and m_2 while the L_1 cache may replace the other. This is true if the L_2 cache is direct mapped or even if it is two-way set associative and m_1 and m_2 fall into the same set in it as well. In fact, we can generalize this example to see that inclusion can be violated if L_1 is not direct mapped and uses an LRU replacement policy, regardless of the associativity, block size, or cache size of the L_2 cache.

- *Multiple caches at a level.* A similar problem with replacements is observed when the first-level caches are split between instructions and data, even if they are direct mapped and are backed up by a unified second-level cache. Suppose first that the L_2 cache is direct mapped as well. An instruction block m_1 and a data block m_2 that conflict in the L_2 cache do not conflict in the L_1 caches since they go into different caches. If m_2 resides in the L_2 cache and m_1 is referenced, m_2 will be replaced from the L_2 cache but not from the L_1 data cache, violating inclusion. This can be generalized to show that if multiple independent caches are backed up by even a highly associative unified cache below them, inclusion is not guaranteed (see Exercise 6.7[b]).
- *Different cache block sizes.* Finally, caches with different block sizes can violate inclusion. Consider a miniature system with direct-mapped, unified L_1 and L_2 caches ($a_1 = a_2 = 1$), with block sizes 1 word and 2 words, respectively ($b_1 = 1$, $b_2 = 2$), and number of sets 4 and 8, respectively ($n_1 = 4$, $n_2 = 8$). Thus, the size of L_1 is 4 words, and word locations 0, 4, 8, . . . map to set 0, locations 1, 5, 9, . . . map to set 1, and so on. The size of L_2 is 16 words, and word locations 0&1, 16&17, 32&33, . . . map to set 0, locations 2&3, 18&19, 34&35, . . . map to set 1, and so on. It is now easy to see that while the L_1 cache can contain the words at both word locations 0 and 17 at the same time (they map to sets 0 and 1, respectively), the L_2 cache cannot because the words map to the same set (set 0) and they are not consecutive words (so a block size of 2 words does not help). Inclusion can be shown to be violated even if the L_2 cache is much larger or has greater associativity as long as the block size is different, and we have already seen the problems when the L_1 cache has greater associativity.

Fortunately, in one of the most commonly encountered cases, inclusion is maintained automatically. This is the situation in which the L_1 cache is direct mapped

$(a_1 = 1)$, L_2 can be direct mapped or set associative ($a_2 \geq 1$) with any replacement policy (e.g., LRU, FIFO, random) as long as the new block brought in is put in both L_1 and L_2 caches, the block size is the same ($b_1 = b_2$), and the number of sets in the L_1 cache is equal to or smaller than in the L_2 cache ($n_1 \leq n_2$). Using such a configuration is one popular way to get around the inclusion problem.

However, many of the cache configurations used in practice do not automatically maintain inclusion on replacements. Instead, inclusion is maintained explicitly by extending the mechanisms used for propagating coherence events in the cache hierarchy. Whenever a block in the L_2 cache is replaced, the address of that block is sent to the L_1 cache, asking it to invalidate or flush (if dirty) the corresponding blocks (there can be multiple blocks if $b_2 > b_1$).

Enhancements are also needed to handle bus transactions and processor writes. Consider bus transactions seen by the L_2 cache. Some, but not all, of the bus transactions relevant to the L_2 cache are also relevant to the L_1 cache and must be propagated to it. For example, if a block is invalidated in the L_2 cache due to an observed bus transaction (e.g., BusRdX), the invalidation must also be propagated to the L_1 cache if the data is present in it. There are several ways to do this. One is to inform the L_1 cache of all transactions that were relevant to the L_2 cache and let it ignore the ones whose addresses do not match any of its tags. This sends a large number of unnecessary interventions to the L_1 cache and can hurt performance by making cache tags unavailable for processor accesses. A more attractive solution is for the L_2 cache to keep extra state (inclusion bits) with cache blocks, which record whether the block is also present in the L_1 cache. It can then suitably filter interventions to the L_1 cache at the cost of a little extra hardware and complexity.

Finally, on an L_1 write hit, the modification needs to be communicated to the L_2 cache so it can supply the most recent data to the bus if necessary. One solution is to make the L_1 cache write through. This has the additional advantage that single-cycle writes are simple to implement (Hennessy and Patterson 1996). However, writes can consume a substantial fraction of the L_2 cache bandwidth, and a write buffer is needed between the L_1 and L_2 caches to avoid processor stalls. The requirement can also be satisfied with write-back L_1 caches since it is not necessary that the data in the L_2 cache be up-to-date but only that the L_2 cache knows when the L_1 cache has more recent data. Thus, the state information for L_2 cache blocks is augmented so that blocks can be marked “modified-but-stale.” The block in the L_2 caches behaves as a modified block for the coherence protocol, but data is fetched from the L_1 cache when it needs to be flushed to the bus. (One simple approach for the modified-but-stale state is to set both the modified and invalid bits.) Both the write-through and write-back L_1 cache solutions have been used in many bus-based multiprocessors. More information on maintaining cache inclusion can be found in (Baer and Wang 1988).

6.3.2 Propagating Transactions for Coherence in the Hierarchy

Given that we have inclusion and we propagate invalidations and flush requests up to the L_1 cache as necessary, let us see how transactions percolate up and down with-

in a processor's cache hierarchy. The intrahierarchy protocol handles processor requests by percolating them downward (away from the processor) until either they encounter a cache that has the requested block in the proper state or they reach the bus. Responses to these processor requests are sent up the cache hierarchy, updating each cache as they progress toward the processor. Read responses are loaded into each cache in the hierarchy in the shared or exclusive state whereas read-exclusive responses are loaded into all levels, except the innermost (L_1), in the modified-but-stale state. In the innermost cache, read-exclusive data is loaded in the modified state, as after the new data is written this will be the most up-to-date copy.

Requests from the bus percolate upward from the external interface (the bus), modifying the state of the cache blocks as they progress. Requests that require a block to be flushed back to the bus can be divided into flush requests that cause the block to be invalidated as well and copy-back requests that don't require invalidation. These requests percolate upward until they encounter the modified copy, at which point a response is generated for the external interface. For simple invalidations, it is not necessary for the bus transaction to be held up until all the copies are actually invalidated. The lowest-level cache controller (closest to the bus) sees the transaction when it appears on the bus, and this serves as a point of commitment to the requestor that the invalidation will be performed in the appropriate order. The response to the invalidation may be sent to the requesting processor from its own bus interface as soon as the invalidation request is placed on the bus, so no responses are generated within the destination cache hierarchies. All that is required is that certain orders be maintained between the incoming invalidations and other transactions flowing through the cache hierarchy, which we shall discuss further in the context of split-transaction buses that allow many transactions to be outstanding at a time.

Interestingly, dual tags are less critical when we have multilevel caches. The L_2 cache acts as a filter for the L_1 cache, screening out irrelevant transactions from the bus, so the tags of the L_1 cache are available almost wholly to the processor. Similarly, since the L_1 cache acts as a filter for the L_2 cache from the processor side (hopefully satisfying most of the processor's requests), the L_2 tags are almost wholly available for the bus snooper's queries (see Figure 6.7). Nonetheless, many machines retain dual tags even in multilevel cache designs.

With only one outstanding transaction on the bus at a time, the major correctness issues do not change much by using a multilevel hierarchy as long as inclusion is maintained. The necessary transactions are propagated up and down the hierarchy, and bus transactions may be held up until the necessary propagation occurs. Of course, the performance penalty for holding up the bus until a response is obtained is more onerous, so we are motivated to try to decouple these operations. Before going further down this path, let us remove the second simplifying assumption, that of an atomic bus, and examine a more aggressive, split-transaction bus. We first return to assuming a single-level processor cache for simplicity and then incorporate multilevel cache hierarchies.

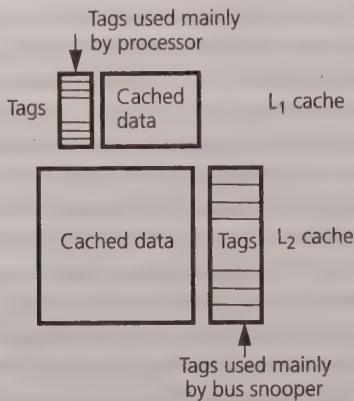


FIGURE 6.7 Organization of two-level snoopy caches. Only a single set of tags is needed for each cache.

6.4

SPLIT-TRANSACTION BUS

An atomic bus limits the achievable bus bandwidth substantially, since the bus wires are idle from the time when the address is taken off the bus until the memory system or another cache supplies the data or response. In a split-transaction bus, transactions that require a response are split into two independent subtransactions—a *request* transaction and a *response* transaction. Other transactions (or subtransactions) are allowed to intervene between them so that the bus can be used while the response to the original request is being generated. Buffering is used between the bus and the cache controllers to allow multiple transactions to be outstanding on the bus waiting for snoop and/or data responses from the controllers. The advantage, of course, is that by pipelining bus operations the bus is utilized more effectively, and hence more processors can share the same bus. The disadvantage is increased complexity.

As examples of request-response pairs, a BusRd transaction is now a request that needs a data response. A BusUpgr does not need a data response, but it does require an acknowledgment indicating that it has committed and hence been serialized. To ensure this acknowledgment does not appear on the bus as a separate transaction, it is usually sent down toward the requesting processor by its own bus controller when it is granted the bus for the BusUpgr request. A BusRdX needs a data response and an acknowledgment of commitment; typically, these are combined as part of the data response. Finally, a write back usually does not have a response.

The major new issues raised by split-transaction buses are as follows:

1. A new request can appear on the bus before the snoop and/or servicing of an earlier request are complete. In particular, *conflicting* requests (two requests to the same memory block, at least one of which is due to a write operation) may be outstanding on the bus at the same time, a case that must be handled very

carefully. Note that this is different from the earlier case of nonatomicity of overall actions despite using an atomic bus. There, a conflicting request could be observed by a cache controller before its request even obtained the bus, so the request could be suitably modified before being placed on the bus. Here, both request subtransactions have already appeared on the bus. Example 6.2 illustrates the difference.

2. The number of buffers for incoming requests and potential data responses from bus to cache controller is usually fixed and small, so we must either avoid or handle buffers filling up. This is called *flow control* since it affects the flow of transactions through the system.
3. Since requests from the bus are buffered, we need to revisit the issue of when and how snoop responses and data responses are produced on the bus. For example, are they generated in order with respect to the requests appearing on the bus or not, and are the snoop and the data part of the same response transaction?

EXAMPLE 6.2 Consider the previous example of two processors P_1 and P_2 having the block cached in shared state and deciding to write it at the same time (Example 6.1). Show how a split-transaction bus may introduce complications that would not arise with an atomic bus.

Answer With a split-transaction bus, P_1 and P_2 may generate BusUpgr requests that are granted the bus on successive cycles. For example, P_2 may get the bus before it has been able to look up the cache for P_1 's request and detect it to be conflicting. If they both assume that they have acquired exclusive ownership, the protocol breaks down because both P_1 and P_2 now think they have the block in modified state. On an atomic bus, this would never happen because the first BusUpgr transaction would complete—snoops, responses, and all—before the second one got on the bus, and the latter would have been forced to change its request from BusUpgr to BusRdX. (Note that even the breakdown on the atomic bus discussed in Example 6.1 resulted in only one processor having the block in modified state and the other having it in shared state.) ■

The design space for split-transaction, cache-coherent buses is large, and a great deal of innovation is ongoing in the industry. Perhaps the most critical issue from the viewpoint of the coherence protocol is how ordering is established and when snoop results are reported. Are they part of the request phase or the response phase? The position adopted in fact influences how conflicting operations can be handled, that is, the first major issue described earlier. Decisions about flow control (as well as conflicting operations) are affected by the number of outstanding requests permitted on the bus at a time. In general, a larger number of outstanding requests allows better bus utilization but requires more buffering and design complexity. The remaining high-level design decision is whether data responses need to be returned in the same order as that in which the requests are issued. The Intel Pentium Pro and DEC Turbo Laser buses are examples of the “in order” approach whereas the SGI Challenge and Sun Enterprise buses allow responses to be out of order. The latter approach is more tolerant of variations in memory access times (memory may be

able to satisfy a later request quicker than an earlier one because of memory bank conflicts or off-page DRAM access) but is more complex. Let us first examine fully how one concrete example design resolves these issues and then discuss alternatives.

6.4.1 An Example Split-Transaction Design

The example is based loosely on the Silicon Graphics Challenge bus architecture, the Powerpath-2. It takes the following positions on the three design issues. Conflicting requests are dealt with very simply, if conservatively: the design disallows multiple requests for a block from being outstanding on the bus at once. In fact, it allows only eight outstanding requests at a time on the bus, thus making the necessary conflict detection tractable. Limited buffering is provided between the bus and the cache controllers, and flow control for these buffers is implemented through *negative acknowledgment*, or NACK, lines on the bus. That is, if a buffer is full when a request or response transaction is observed, which can be detected as soon as the transaction appears on the bus, the transaction is rejected and NACKed; this renders the transaction invalid and asks the initiator to retry. Finally, responses are allowed to be provided in a different order than that in which the original requests appeared on the bus. It is the request phase that establishes the total (bus) order on coherence transactions; however, snoop results from the cache controllers are presented on the bus as part of the response phase, together with the data, if any.

Let us examine this example bus architecture in more detail. We begin with the high-level bus design and how responses are matched up with requests. Then we look at the flow control and snoop result issues in more depth. Finally, we examine the path of a request through the system, including how conflicting requests are kept from being simultaneously outstanding on the bus.

6.4.2 Bus Design and Request-Response Matching

The split-transaction bus design essentially consists of two separate buses, a request bus for command and address and a response bus for data. The request bus provides the type of request (e.g., BusRd, BusWB) and the target address. Since responses may arrive out of order with regard to requests, there should be a way to match returning responses with their outstanding requests. When a request (command-address pair) is granted the bus by the arbiter, it is also assigned a unique tag (3 bits since the design allows eight outstanding requests). A response consists of data on the data bus as well as the original request tag on the 3-bit-wide tag lines. The use of tags means that responses do not need to use the address lines, keeping them available for other requests. The address and the data buses can therefore be arbitrated for separately. There are separate bus lines for arbitration as well as for flow control and snoop results.

Cache blocks are 128 bytes (1,024 bits) and the data bus is 256 bits wide in this particular design, so four bus cycles plus a one-cycle turnaround time are required for the response phase. A uniform pipeline strategy is followed, so the request phase

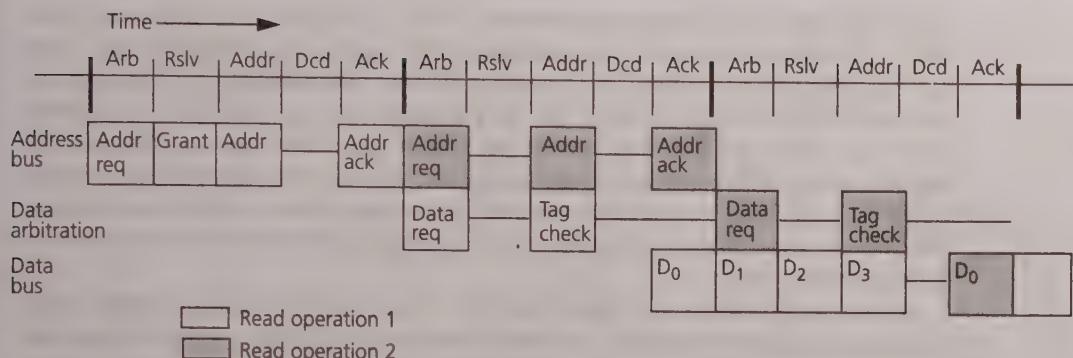


FIGURE 6.8 Complete read transaction for a split-transaction bus. A pair of consecutive read operations is performed on consecutive phases, distinguished by shaded boxes. Each phase consists of five specific cycles: arbitration, resolution, address, decode, and acknowledgment. Transactions are split into three phases: address request (which uses the address bus), data request (which uses the data bus arbitration and related logic), and data response (which uses the data bus).

is also five bus cycles: arbitration, resolution, address, decode, and acknowledgment. Overall, a complete request-response transaction takes three or more of these five-cycle phases—at the minimum an address request phase (which uses the address bus), a data request phase (which uses the data bus arbitration logic and obtains access to the data bus for the response subtransaction), and a data transfer or response phase (which uses the data bus). Three different memory operations can be in the three different phases at the same time. This basic pipelining strategy underlies several of the higher-level design decisions.

To understand this strategy, let's follow a single read operation through to completion, as shown in Figure 6.8. We begin with the address request phase. In the request arbitration cycle, a cache controller presents its request for the bus. In the request resolution cycle, all requests are considered, a single one is granted, and a tag is assigned. The winner drives the address in the following address cycle and then all controllers have a cycle to decode it and look up the cache tags to determine whether there is a snoop hit (the snoop result will be presented on the bus later). At this point, cache controllers can take the action that makes the operation visible to the processor. On a BusRd, an exclusive block is downgraded to shared; on a BusRdX or BusUpgr, blocks are invalidated. In either case, a controller owning the block as dirty knows that it will need to flush the block to the bus in the response phase. If a cache controller is not able to complete the snoop and take the necessary action during the address phase (say, if it is unable to gain access to the cache tags), it can inhibit the completion of this phase in the acknowledgment cycle until it completes the snoop. (During the acknowledgment cycle, the first data transfer cycle for the previous memory operation can take place, occupying the data lines for four cycles; see Figure 6.8.)

After the address request phase of the overall transaction, it is known which module should respond with the data: the memory or a cache. The responder may

request the data bus during the arbitration cycle of the next 5-cycle phase. (Note that in this cycle a requestor also initiates a new request on the address bus.) The data bus arbitration is resolved in the next cycle, and in the address cycle the tag can be checked. If the target is ready, the data transfer starts on the acknowledgment cycle and continues for three additional cycles (i.e., into the data transfer or response phase). After a single turnaround cycle, the next data transfer (whose arbitration is proceeding in parallel) can start. The cache block sharing state (snoop result) is conveyed with the response phase, and state bits are set when the data is updated in the cache.

As discussed earlier, write backs (BusWB) consist only of a request phase. They require use of both the address and data lines together and thus must arbitrate for simultaneous use of both resources. Finally, upgrades (BusUpgr) performed to acquire exclusive ownership for a block also have only a request part since no data response is needed on the bus. The processor performing a write that generates the BusUpgr is sent a response by its own bus controller when the BusUpgr is actually placed on the bus, indicating that the write is committed and has been serialized in the bus order.

To keep track of the eight outstanding requests on the bus, each cache controller maintains an eight-entry table, called a *request table* (see Figure 6.9). Whenever a new request is issued on the bus, it is added to all request tables at the same index as part of the arbitration process. The index is the 3-bit tag assigned to that request during arbitration. (Requests are also buffered separately on their way to cache hierarchy.) A request table entry contains the address of the block associated with the request, the request type, the state of the block in the local cache (if it has already been determined), and a few other bits. The request table is fully associative, so all request table entries are examined for a match by both requests issued by the local processor and by other requests (using the address field) and responses (using the tag) observed from the bus. A request table entry is freed when a response to the request is observed on the bus. The 3-bit tag value associated with that request is reassigned by the bus arbiter only at this point, so there are no conflicts in the request tables.

6.4.3 Snoop Results and Conflicting Requests

Like the SGI Challenge, this example design uses variable delay snooping. The snoop portion of the bus consists of the three wired-OR lines discussed earlier: shared, dirty, and inhibit (which extends the duration of the current response phase). While it is determined at the end of the address request phase which module is to respond with the data, it may be many cycles before that data is ready and the responder gains access to the data bus. During this time, the snoop response is held in the request table, and other requests and responses may take place. To simplify matching snoop results with their requests, in this design the snoop results are presented on the bus by all controllers at the time they see the actual response to a

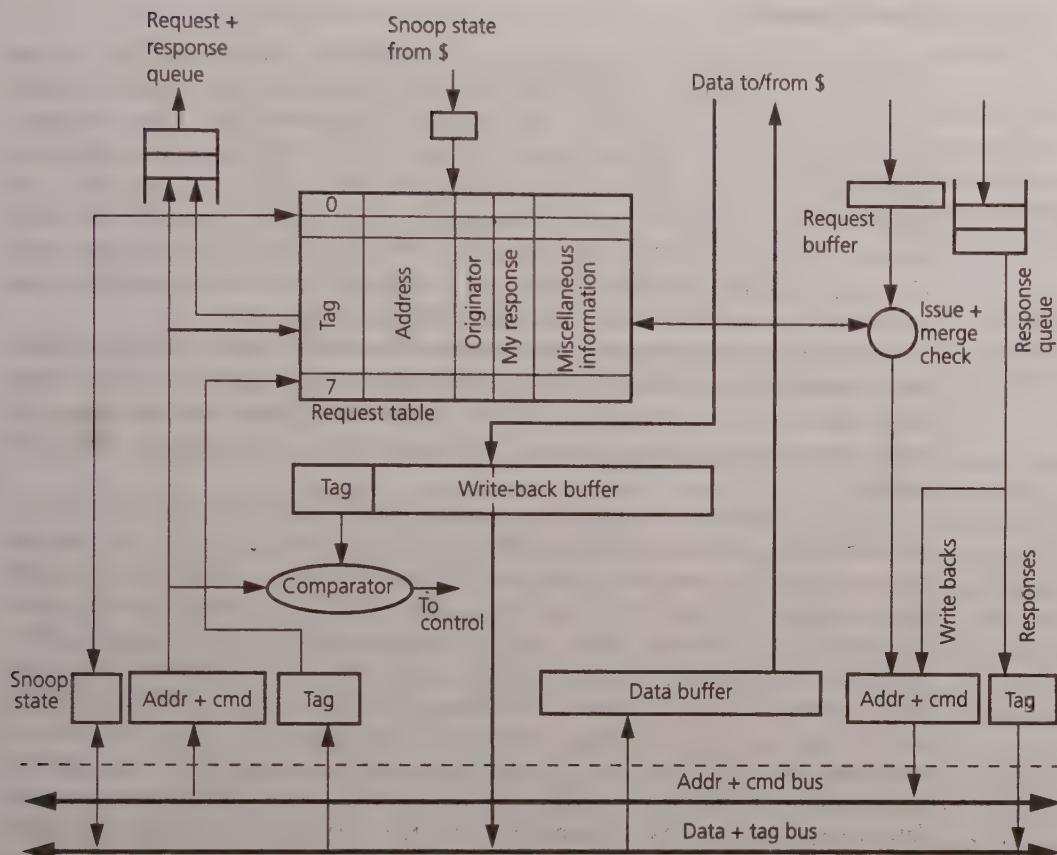


FIGURE 6.9 Extension of the bus interface logic shown in Figure 6.4 to accommodate a split-transaction bus. The key addition is an eight-entry request table that keeps track of all outstanding requests on the bus. Whenever a new request is issued on the bus, it is added at the same index in all processors' request tables. The request table serves many purposes, including request merging and ensuring that only a single request can be outstanding for any given memory block.

request being put on the bus, that is, during the response phase. Write-back and upgrade requests do not have a data response, but then they do not require a snoop response either.

Avoiding conflicting requests is easy: since every controller has a record of the pending transactions that have been issued to the bus in its request table, no request is issued for a block that has a transaction outstanding. Thus, even though the bus is pipelined, the operations for an individual location are serialized as in the atomic case. Writes are committed during the request phase, which affects the serialization.

6.4.4 Flow Control

In addition to its use for incoming requests from the bus, flow control may also be required in other parts of the system. The cache subsystem has a buffer in which responses to its requests can be stored, in addition to the write-back buffer discussed earlier. If the processor or cache allows only one outstanding request at a time, as we have been implicitly assuming, this response buffer is only one entry deep. The number of buffer entries is usually kept small anyway, since a response buffer entry contains not only an address but also a cache block of data and is therefore large. The cache controller provides flow control by limiting the number of requests it has outstanding so that buffer space is available for every response.

Flow control is also needed at main memory. Each of the (eight) pending requests can generate a write back that main memory must accept in addition to the request itself. Since write-back transactions do not require a response, they can happen in quick succession on the bus, possibly overflowing buffers in the main memory subsystem.

The SGI Challenge design provides separate NACK lines for the address and data portions of the bus since the bus allows independent arbitration for each portion. Before a request or response subtransaction has reached its acknowledgment cycle and completed, the main memory or any other processor can assert a NACK signal, for example, if it finds its buffers full. The subtransaction is then canceled everywhere and must be retried. One common option, used in the Challenge, is to have the requestor for that subtransaction retry periodically until it succeeds. Backoff and priorities can be used to reduce bandwidth consumption for failed retries and to avoid starvation. The Sun Enterprise uses an interesting alternative for data transfers that encounter a full buffer. In this case, the receiver—which could not accommodate the data on the first attempt—initiates the retry when it has enough buffer space. The original supplier simply keeps watch for the retry transaction on the bus and places the data on the data bus. The operation of the Enterprise bus ensures that the space in the destination buffer is still available when the data arrives. This guarantees that data transfers will succeed with only one retry bus transaction.

6.4.5 Path of a Cache Miss

Given this example design, we are ready to examine how various requests may be handled and what race conditions might occur. Let us first look at the case where a processor has a read miss in the cache so that the request part of a BusRd transaction should be generated. The request first checks the currently pending entries in the request table. If it finds one with a matching address, it can take two possible courses of action, depending on the nature of the pending request:

1. If the earlier request was a BusRd request for the same block, this is great news for this processor: the request needn't be put on the bus but can just obtain the data when the response to the earlier request appears on the bus. To accomplish this, we add two new bits to each entry in the request table, which say: Do I wish to obtain the data response for this request? Am I the

original generator of this request? In our situation, these bits will be set to 1 and 0, respectively. The purpose of the first bit is obvious; the purpose of the second bit is to help determine in which state (exclusive versus shared) the data response will be loaded. If a processor is not the original requestor, then it must assert the sharing line on the snoop bus when it obtains the response data from the bus so that all caches will load this block in shared state and not exclusive. If a processor is the original requestor, it does not assert the sharing line when it obtains the response from the bus, and if the sharing line is not asserted at all, then it will load the block in exclusive state.

2. If the earlier request conflicts with a BusRd (e.g., a BusRdX), the controller must hold on to the request until it sees a response to the previous request on the bus and only then attempt the request. The processor-side controller is typically responsible for this.

If the controller finds no matching entries in the request table, it can go ahead and issue the request on the bus. However, it must watch out for a race condition of the type we discussed earlier. When the controller first examines the request table, it may find no conflicting requests, so it may request arbitration for the bus. However, before it is granted the bus, a conflicting request may appear on the bus, and then it may be granted the very next use of the bus. Since this design does not allow conflicting requests on the bus, when the controller sees a conflicting request in the slot just before its own, it should (1) issue a null request (a no-action request) on the bus to occupy the slot it had been granted and (2) withdraw from further arbitration until a response to the conflicting request has been generated.

Suppose the processor does manage to issue the BusRd request on the bus. What should other cache controllers and the main memory controller do? The request is entered into the request tables of all cache controllers, including the one that issued it, as soon as it appears on the bus. The controllers start checking their caches for the requested memory block. The main memory subsystem has no idea whether this block is dirty in one of the processor's caches, so it independently starts fetching this block. Now we have three different scenarios to consider:

1. One of the caches may determine that it has the block in modified state and may acquire the bus to generate a response before main memory can respond. On seeing the response on the bus, main memory simply aborts the fetch that it had initiated, and the cache controllers that are waiting for this block load the data in a state based on the values of the snooping lines. If a cache controller has not finished snooping by the time the response appears on the bus, it will keep the inhibit line asserted and the response transaction will be extended (i.e., will stay on the bus). Main memory also receives the response since the block was dirty in a cache. If main memory does not have the buffer space needed, it asserts the NACK signal provided for flow control, and it is the responsibility of the controller holding the block dirty to retry the response transaction later.
2. Main memory may fetch the data and acquire the bus before the cache controller holding the block dirty has finished its snoop and/or acquired the bus.

The controller holding the block dirty will first assert the inhibit line until it has finished its snoop and then assert the dirty line and release the inhibit line, indicating to the memory that it has the latest copy and that memory should not actually put its data on the bus. On observing the dirty line, memory cancels its response transaction and does not actually put the data on the bus. The cache with the dirty block will acquire the bus sometime later and put the data response on it.

3. The simplest scenario is that no other cache has the block dirty. Main memory will acquire the bus and generate the response. Cache controllers that have not finished their snoop will assert the inhibit line when they see the response from memory, but once they deassert it, memory can supply the data. (Cache-to-cache sharing is not used for data in shared state in this system.)

Processor writes are handled similarly to reads. If the writing processor does not find the data in its cache in a valid state, a BusRdX is generated. As before, it checks the request table and then goes on the bus. Everything is the same as for a bus read, except that main memory will not take the data response if it comes from another cache (since it's going to be modified again by the writer) and no other processor can grab the data. If the block being written is valid but in shared state, a BusUpgr is issued. This requires no response transaction (the currently valid block is known to be in main memory as well as in the writer's cache); however, if any other processor was just about to issue a BusUpgr for the same block, it will now need to convert its request to a BusRdX as in the atomic bus.

6.4.6 **Serialization and Sequential Consistency**

Consider serialization to a single location. If a request subtransaction appearing on the bus is a read, no subsequent write appearing on the bus after the read should be able to change the value returned by the read. Despite multiple outstanding transactions on the bus, here this is easy since conflicting requests to the same location are not allowed simultaneously on the bus; the read response subtransaction will therefore precede the write request, and the read will complete before the write can affect the cached value. If the transaction appearing on the bus is a BusRdX or BusUpgr generated by a write operation, the requesting cache will perform the write into the cache array after the response phase and before issuing any other memory operations; subsequent (conflicting) reads to the block from any processor are allowed on the bus only after the response phase for the write, so they are guaranteed to obtain the new value. (Recall that the response phase for a write operation may be a separate action on the bus, as in a BusRdX, or may be implicitly generated once the request wins arbitration, as in a BusUpgr.)

Now consider the serialization of operations to different locations needed for sequential consistency. The logical total order on bus transactions is established by the order in which requests for the address bus are granted. Once a BusRdX or BusUpgr has obtained the bus, the associated write is committed. However, with

multiple outstanding requests on the bus, the invalidations are buffered as well, and it may be a while before they are actually applied to the cache (unlike in the atomic bus where this was assumed to happen immediately). Commitment of a write does not guarantee that the value produced by the write is already visible to all other processors; only actual completion guarantees that. (Performing with respect to a processor guarantees it for that processor.) Further mechanisms are needed to ensure that the necessary orders are preserved between the bus and the processor. Example 6.3 will help make this concrete.

EXAMPLE 6.3 Consider the two code fragments shown below. What results for (A,B) are disallowed under SC? Assuming a single level of cache per processor and multiple outstanding transactions on the bus, and no special mechanisms to preserve orders between bus and cache or processor, show how the disallowed results may be obtained. Assume an invalidation-based protocol and initial values for A and B of 0 in both caches.

P ₁	P ₂	P ₁	P ₂
A = 1	rd B	A = 1	B = 1
B = 1	rd A	rd B	rd A

Answer In the first example, on the left, the result not permitted under SC is (A,B) = (0,1). However, consider the following scenario. P₁'s write of A commits, so it continues with the write of B (under the revised sufficient conditions for SC). The invalidation for B is applied to the cache of P₂ before that for A because they get reordered in the buffers. P₂ incurs a read miss on B and obtains the new value of 1. However, the invalidation for A is still in the buffer and is not applied to P₂'s cache even by the time P₂ issues the read of A. The read of A is a hit and completes returning the old value 0 for A from the cache.

The example on the right does not require invalidations to be reordered to violate SC. The disallowed result is (0,0). However, consider the following scenario. P₁ issues and commits its write of A and then completes the read of B, reading in the old value of 0. P₂ then writes B, which commits, so P₂ proceeds to read A. The write of B appears on the bus (commits) after the write of A, so they should be serialized in that order and P₂ should read the new value of A. However, the invalidation corresponding to the write of A by P₁ is sitting in P₂'s incoming buffer and has not yet been applied to P₂'s cache. P₂ sees a read hit on A and completes returning the old value of A, which is 0. ■

With commitment substituting for completion and multiple outstanding operations being buffered between bus and processor, the key property that must be preserved for sequential consistency is the following: a processor should not be allowed to actually see the new value due to a write before previous writes (in bus order, as usual) are visible to it. There are two ways to preserve this property: by not letting certain types of incoming transactions from bus to cache be reordered in the incoming queues; and by allowing these reorderings in the queues, but then ensuring that the important orders are preserved at the necessary points in the machine. Let us examine each approach briefly.

A simple way to follow the first approach is to ensure that all incoming transactions from the bus (invalidations, read-miss replies, write commitment acknowledgments, etc.) propagate to the processor in FIFO order. However, such strict ordering is not necessary. Consider preserving the desirable property just described with an invalidation-based protocol. Here, there are two ways for a new value to be brought into the cache and made available to the processor to read without it incurring another bus operation: One is through a read miss, and the other is through a write by that processor. On the other hand, writes from other processors become visible to a processor (even though the values are not yet available locally) when the corresponding invalidations are applied to its cache. For writes to be defined as previous to the operation that provides the new value, they must have appeared on the bus before the operation (or a previous bus transaction from that processor in the case of a write hit). Thus, the invalidations due to those writes are already in the incoming queue or applied to the cache when the relevant transaction appears on the bus and, hence, when its reply comes back. All we need to ensure, therefore, is that a reply (read miss or write commitment acknowledge) does not overtake an invalidation between the bus and the cache, that is, that all previous invalidations are applied before the reply is received by the cache.

Note that incoming invalidations may be reordered with regard to one another. This is because the new value corresponding to an invalidation is seen only through the corresponding read miss, and the read-miss reply is not allowed to be reordered with respect to the previous invalidation. In an update-based protocol, on the other hand, the new value due to a write can be seen as soon as the incoming update has been applied. This means not only that replies should not overtake updates but that updates should not overtake updates either.

An alternative is to allow incoming transactions from the bus to be reordered arbitrarily on their way to the cache but to ensure that all previously committed writes are applied to the cache (by servicing them from the incoming queue) before an operation from the local processor that will enable it to see a new value can be completed. After all, what really matters is not the order in which invalidations or updates are applied but the order in which the corresponding new values can be seen by the processor. There are two natural ways to accomplish this. One is to service the incoming invalidations and updates from the queue every time the processor tries to complete an operation that places a new value in the cache. In an invalidation-based protocol, this means servicing the queue before the processor is allowed to complete a read miss or a write that generates a bus transaction; in an update-based protocol, it means servicing it on every read hit as well. The other way is to service the queue when a processor is about to actually access a value (complete a read hit or miss), if a new value (i.e., a reply or an update since the last time the queue was serviced) has indeed been applied to the cache. The fact that operations are reordered from bus to cache and a new value has been applied to the cache means that invalidations or updates may be in the queue that correspond to writes that are previous to that new value; those writes should now be applied before the read can complete. Showing that these techniques disallow the undesirable results in Example 6.3 is left as an exercise that may help make the techniques concrete. As we

will see soon, the extension of the techniques to multilevel cache hierarchies is quite natural.

Regardless of which approach is used, write atomicity is provided naturally by the broadcast nature of the bus. The bus implies that writes are committed in the same order with respect to all processors and that a read cannot see the value produced by a write until that write has committed with respect to all processors (recall that the writing processor ensures this locally too). With the preceding techniques, we can substitute complete for commit in this statement, thus ensuring atomicity. The other major correctness issues—deadlock, livelock, and starvation—for a split-transaction bus are discussed after we have introduced multilevel cache hierarchies in this context. First, let us look at some alternative approaches to organizing a protocol with a split-transaction bus.

6.4.7 Alternative Design Choices

Alternative positions exist for request-response ordering, dealing with conflicting requests, and flow control other than the ones taken by our example (SGI Challenge-based) split-transaction bus design. For example, ensuring that responses are generated on the bus in order with respect to requests—as cache controllers are inclined to do—would simplify the design. The fully associative request table could be replaced by a simple FIFO buffer for the purpose of request-response matching (fully associative lookups may still be needed if conflicting requests are to be disallowed). As before, a request is put into the FIFO only when it actually appears on the bus, ensuring that all entities (processors and memory system) have exactly the same view of pending requests. The cache controllers and the memory system process requests in FIFO order. At the time the response is presented (as in the earlier design), if others have not completed their snoops, they assert the inhibit line and extend the transaction duration. That is, snoops are still reported together with responses. The difference is in the case where the memory generates a response first even though a processor has that block dirty in its cache. In the previous, unordered design, the cache controller that had the block dirty released the inhibit line and asserted the dirty line and arbitrated for the bus again later when it had retrieved the data from the cache. But now to preserve the FIFO order, this response has to be placed on the bus before the response to any later request. So the controller with the dirty block does not release the inhibit line but extends the current bus transaction until it has fetched the block from its cache and supplied it on the bus. Accomplishing this does not depend on anyone else having to access the bus, so there is no deadlock problem.

Although FIFO request-response ordering is simpler, it can have performance problems. Consider a multiprocessor with an interleaved memory system. Suppose three requests A , B , and C are issued on the bus in that order and that A and B go to the same memory bank while C goes to a different one. Forcing the system to generate responses in order means that C will have to wait for both A and B to be processed, though data for C will be available well before data for B is available because of B 's bank conflict with A . The behavior of main memory is the major motivation

for allowing out-of-order responses since caches are likely to respond to requests in order anyway.

Keeping responses in order also makes it more tractable to allow conflicting requests to the same block to be outstanding on the bus, thus eliminating the need for the fully associative request table lookup as well as increasing bandwidth. Suppose two BusRdX requests are issued on a block in rapid succession. The controller issuing the later request will invalidate its block when it sees the earlier request, as before. The tricky part with a split-transaction bus is that the controller issuing the earlier request sees the later request appear on the bus before the data response that it awaits. It cannot simply invalidate its block in reaction to the later request since the block is in flight and its own write needs to be performed before a flush or invalidate. With out-of-order responses, allowing this conflicting request may be difficult. With in-order responses, the earlier requestor knows its response will appear on the bus first, so this is actually an opportunity for a performance-enhancing optimization. The earlier requesting controller reacts to the later request by simply noting that the latter is pending. When its response block arrives, it updates the word to be written and “shortcuts” the modified block back out to the bus to serve as the response to the later request, leaving its own block invalid. This optimization reduces the latency of ping-ponging a block under write-write false sharing.

If the delay from request to snoop result is fixed, conflicting requests can be allowed even without requiring data responses to be in order. However, since conflicting requests to a block go into the same queue at memory as well, the data responses for these requests themselves usually appear in order anyway, so they can be handled using the shortcut method just described (this is done in the Sun Enterprise systems).

In fact, as long as a well-defined order can be identified among the request transactions, they do not even need to be issued sequentially on the same bus. For example, the Sun SparcCenter 2000 used two distinct split-transaction buses and the CRAY 6400 used four to improve bandwidth for large configurations. Multiple requests may thus be issued on a single cycle. However, a simple priority is established among the buses so that a logical order is defined even among the concurrent requests.

6.4.8 Split-Transaction Bus with Multilevel Caches

We are now ready to combine the two major enhancements to the basic protocol from which we started: multilevel caches and a split-transaction bus. The design we examine is a (Challenge-like) split-transaction bus and a two-level cache hierarchy. The issues and solutions generalize to deeper hierarchies. We have already seen the basic issues of request, response, and invalidation propagation up and down the hierarchy. The key new issue we need to grapple with is that it takes a considerable number of cycles for a request to propagate through the cache controllers. During this time, we must allow other transactions to propagate up and down the hierarchy as well. To maintain high bandwidth while allowing the individual units (e.g., controllers and caches) to operate at their own rates, queues are placed between levels

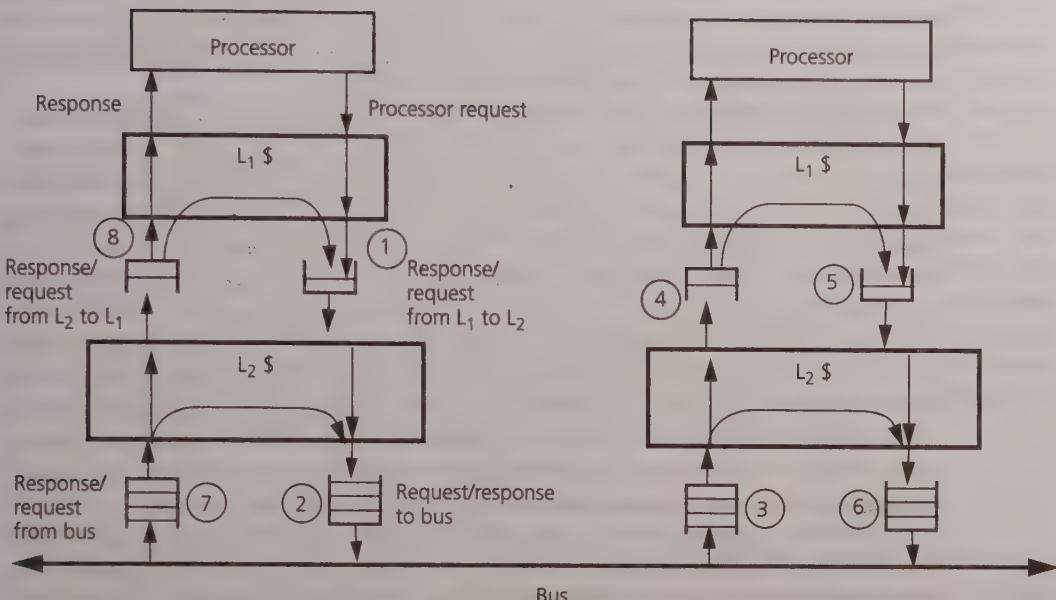


FIGURE 6.10 Internal queues that may exist inside a multilevel cache hierarchy. Each level of the hierarchy has input queues from above and below that it must service. An operation may produce a request or response to the adjacent levels. For example, a read request that misses in the L₁ cache is passed on to the L₂ cache (1). If it misses there, a request is placed on the bus (2). The read request is captured by all other cache controllers in the incoming queue (3). Assuming the block is currently in modified state in the L₁ cache of another processor, the request is queued for L₁ service (4). The L₁ demotes the block to shared and flushes it to the L₂ cache (5), which places it on the bus (6). The response is captured by the requestor (7) and passed to the L₁ (8), whereupon the word is provided to the processor.

of the hierarchy as well. However, this raises a family of questions related to deadlock and serialization.

A simple multilevel cache organization is shown in Figure 6.10. Assume that a processor can have only one request outstanding at a time, so there are no queues between the processor and first-level cache. One concern with such queue structures is deadlock. To avoid the fetch deadlock problem discussed earlier, an L₂ cache needs to be able to buffer incoming requests or responses while it has a request outstanding (as before) so that the bus may be freed up. With one outstanding request per processor, the incoming queues between the bus and the L₂ cache need to be large enough to hold the number of requests that can be outstanding on the bus from other processors plus a response to its own request. This takes care of the case where all requests are destined for a given cache while that cache has a request outstanding. If the queues are made smaller than this to conserve real estate, bus requests are NACKed when room is not available to enqueue them. This discussion applies to single-level or multilevel cache hierarchies with a split-transaction bus. One slot in the bus-to-L₂

and in the L_2 -to- L_1 queues is reserved for the response to the processor's outstanding request so that each processor can always drain its outstanding responses. If NACKs are used, the bus arbitration needs to include a mechanism, such as a simple priority scheme, to ensure forward progress under heavy contention.

In addition to fetch deadlock, classical buffer deadlock can occur within the multilevel cache hierarchy as well. For example, suppose there is a queue in each direction between the L_1 and L_2 cache, both of which are write-back caches, and each queue can hold one entry. It is possible that the $L_1 \rightarrow L_2$ queue holds an outgoing read request, which can be satisfied in the L_2 cache but will generate a reply to L_1 , and the $L_2 \rightarrow L_1$ queue holds an incoming read request, which can be satisfied in the L_1 cache but will generate a reply to L_2 . We now have a classical circular buffer dependence, and hence deadlock. Note that this problem occurs only in hierarchies in which a higher-level cache (closer to the processor) than the one closest to the bus is a write-back cache. Otherwise, incoming requests do not generate replies from higher-level caches, so there is no circularity and no buffer deadlock problem (recall that invalidations are acknowledged implicitly from the bus itself and do not need acknowledgments from the caches).

A hardware-intensive way to deal with this buffer deadlock problem in a multi-level write-back cache hierarchy is to limit the number of outstanding requests from processors and then provide enough buffering for incoming requests and responses at each level. However, this requires a lot of real estate and is not scalable. Each request may need two outgoing buffer entries—one for the request and one for the write back it might generate. With a large number of outstanding bus transactions being allowed, the incoming buffers may need to have many entries as well. An alternative way uses a general deadlock avoidance technique for situations with limited buffering, which we discuss more fully in Chapter 7 in the context of systems with physically distributed memory, where the problem is more acute. The basic idea is to separate the operations that flow through the buffers and communication medium into requests and responses. An operation is classified as a response if it does not generate any further operations but is simply sunk by its destination. A request may generate a response, but no operation may generate another request (although in this case a request may be transferred to the next level of the hierarchy, initiating a new request-response pair and ending the first request, if it does not generate a response at the original level). With this classification, we can avoid deadlock if we provide separate queues for requests and responses in each direction and ensure that responses are always extracted (sunk) from the queues, thus allowing requests to make progress as well. After we discuss this technique in Chapter 7, we apply it to this particular situation with multilevel write-back caches in the exercises.

There are other potential deadlock considerations. For example, if the number of outstanding transactions on the bus is smaller than the number of outstanding requests allowed by the caches, it may be important for a response from a processor's cache to get to the bus before new outgoing requests from it are allowed. Otherwise, the existing requests may never be satisfied and there will be no progress. The outgoing queue or queues must be able to support responses bypassing requests when necessary.

Other than deadlock, a concern with these queue structures is maintaining sequential consistency. With multilevel caches, it is all the more important that the bus not wait for an invalidation to reach all the way up to the first-level cache and return a reply; it should instead consider the write committed when it has been placed on the bus and hence in the input queue to the lowest-level cache. The separation of commitment and completion is even greater in this case. However, the techniques discussed for single-level caches extend naturally to this case: we simply apply them at each level of the cache hierarchy. Thus, in an invalidation-based protocol, the first technique extends to ensuring at each level of the hierarchy that replies are not reordered with respect to invalidations in the incoming queues to that level (replies from a lower-level cache to a higher-level cache are treated as replies, too, for this purpose). The second technique extends to either not letting an outgoing memory operation proceed past a level of the hierarchy before the incoming invalidations to that level are applied there or draining the incoming invalidations to a level if a reply has been applied to that level since the last drain.

6.4.9

Supporting Multiple Outstanding Misses from a Processor

Although we have examined split-transaction buses, we have implicitly assumed so far that a given processor can have only one outstanding memory request at a time. This assumption is simplistic for modern processors, which permit multiple outstanding requests to tolerate the latency of cache misses even on uniprocessor systems. Whereas allowing multiple outstanding references from a processor improves performance, it can also complicate semantics since memory accesses from the same processor may complete in a different order in the memory system than that in which they were issued.

One example of multiple outstanding references is the use of a write buffer. Since we would like to let the processor proceed to other computation and even memory operations after it issues a write, we put the write in the write buffer. Until the write is serialized, it should not be made visible since, otherwise, it may violate write serialization and coherence. One possibility is to write it into the local cache but not make it available until exclusive ownership is obtained (i.e., not let the cache respond to requests for it until then). The more common approach is to keep it in the write buffer and put it in the cache (making it available to other processors through the bus) only when exclusive ownership is obtained.

Most processors use write buffers more aggressively, issuing a sequence of writes in rapid succession into the write buffer without stalling the processor. In a uniprocessor, this approach is very effective as long as reads check the write buffer to satisfy dependences. The problem in the multiprocessor case is that, in general, the processor cannot be allowed to proceed with (or at least complete) memory operations past the write until the exclusive ownership transaction for the block has been placed on the bus and hence serialized. However, there are special cases where the processor can issue a sequence of writes and consider them complete without stalling. One example is if it can be determined that the writes are to blocks that are in the local cache in modified state. Then they can be buffered between the processor

and the cache as long as the cache processes the writes before servicing a request from the bus side (to the same block for coherence and to any block for SC). An important special case exists in which a sequence of writes can be buffered regardless of the cache state: the writes are all to the same block and no other memory operations from that processor are interspersed between those writes in the program order. The writes may be coalesced while the controller is obtaining the bus for the read-exclusive transaction. When that transaction occurs, it makes the entire sequence of writes visible at once. The behavior is the same as if the writes were performed locally as hits after the bus transaction but before the next one. Note that there is no problem with sequences of write backs since the protocol does not require them to be ordered.

More generally, to satisfy the sufficient conditions for sequential consistency, a processor having the ability to proceed past outstanding write, and even read, operations raises the question of which entity should wait to “issue” an operation until the previous one in program order completes. Forcing the processor itself to wait can eliminate any benefits of the sophisticated processor mechanisms (such as write buffers and out-of-order execution). Instead, since the issue is visibility, the buffers that hold the outstanding operations—such as the write buffer or the reorder buffer in dynamically scheduled out-of-order execution processors—can serve this purpose. The processor can issue the next operation right after the previous one, and the buffers take charge of making sure that write operations are not visible to the memory and interconnect systems (i.e., not issuing them to the externally visible memory system) until the appropriate time or that read operations are not allowed to complete out of program order with respect to the commitment of outstanding writes even though the processor may issue and execute them out of order. The mechanisms needed in the buffers are often already available for the purpose of providing precise interrupts in uniprocessors, and we will discuss them in later chapters. Of course, simpler processors that do not proceed past reads or writes make it easier to maintain sequential consistency. Further semantic implications of multiple outstanding references for memory consistency models are discussed when we examine consistency models in detail in Chapter 9.

From a design perspective, exploiting multiple outstanding references most effectively requires that the caches allow multiple cache misses to be outstanding at a time so that the latencies of these misses can be overlapped. This in turn requires that either the cache or some auxiliary data structure keep track of the outstanding misses, which can be quite complex since the responses may return out of order. Caches that allow multiple outstanding misses are called *lockup-free* caches (Kroft 1981), as opposed to *blocking* caches that allow only one outstanding miss. We discuss the design of lockup-free caches when we discuss latency tolerance in Chapter 11.

Finally, consider the interactions with split-transaction buses and multilevel cache hierarchies and the requirements for deadlock avoidance. Given a design that supports a split-transaction bus and a multilevel cache hierarchy, the extensions needed to support multiple outstanding operations per processor are few and are mostly for performance. We simply need to provide deeper request queues from the

processor to the bus (the request queues pointing downward in Figure 6.10), so that the multiple outstanding requests can be buffered and the processor or cache does not stall. It may also be useful to have deeper response queues and more write-back and other types of buffers, since the system now affords more concurrency. As long as deadlock is handled by separating requests from replies and providing them with logically separate buffers, the exact length of any of these queues is not critical for correctness. The reason for so few changes is that the lockup-free caches themselves perform the complex task of merging requests and managing replies to the same block, so to the caches and the bus subsystem below, it simply appears that multiple requests to distinct blocks are coming from the processor. Some potential fetch deadlock scenarios might become exposed that do not arise with only one outstanding request per processor; for example, we may now see the situation where the number of requests outstanding from all processors is more than the bus can take, so we have to ensure responses can bypass requests on the way out. Nevertheless, the support discussed earlier for multiple outstanding transactions on split-transaction buses makes the rest of the system capable of handling multiple requests from a processor without deadlock.

6.5 CASE STUDIES: SGI CHALLENGE AND SUN ENTERPRISE 6000

This section places the general design and implementation issues discussed in the preceding sections into a concrete setting by describing two bus-based multiprocessor systems—the SGI Challenge and the Sun Enterprise 6000. It focuses less on logical issues and more on the organizational and engineering issues as manifested in these real systems. It illustrates how two systems take very different positions on these issues.

The SGI Challenge is designed to support up to 36 MIPS R4400 processors (peak 2.7 GFLOPS total) or up to 18 MIPS R8000 processors (peak 5.4 GFLOPS). Both systems use the same system bus, the Powerpath-2 bus, which provides a peak bandwidth of 1.2 GB/s. The Challenge supports up to 16 GB of eight-way interleaved main memory and up to four PowerChannel-2 I/O buses. Each I/O bus provides a peak bandwidth of 320 MB/s and can support multiple Ethernet connections, VME/SCSI buses, graphics cards, and other peripherals. The total disk storage on the system can be several terabytes. The operating system is a variant of SVR4 UNIX called IRIX; it is a symmetric multiprocessor kernel in that any of the operating system's tasks can be done on any of the processors in the system. Figure 6.11 presents a high-level diagram of the SGI Challenge system organization.

The Sun Enterprise 6000, introduced later than the Challenge, is designed to support up to 30 UltraSparc processors (peak 9 GFLOPs). The Gigaplane system bus provides a peak bandwidth of 2.67 GB/s, and the system can support up to 30 GB of up to 16-way interleaved memory. The 16 slots in the machine can be populated with a mix of processing boards and I/O boards, as long as at least one of each is present. Each processing board has two CPU modules and two (512-bit-wide) memory banks of up to 1 GB each, so the memory capacity and bandwidth scales with the

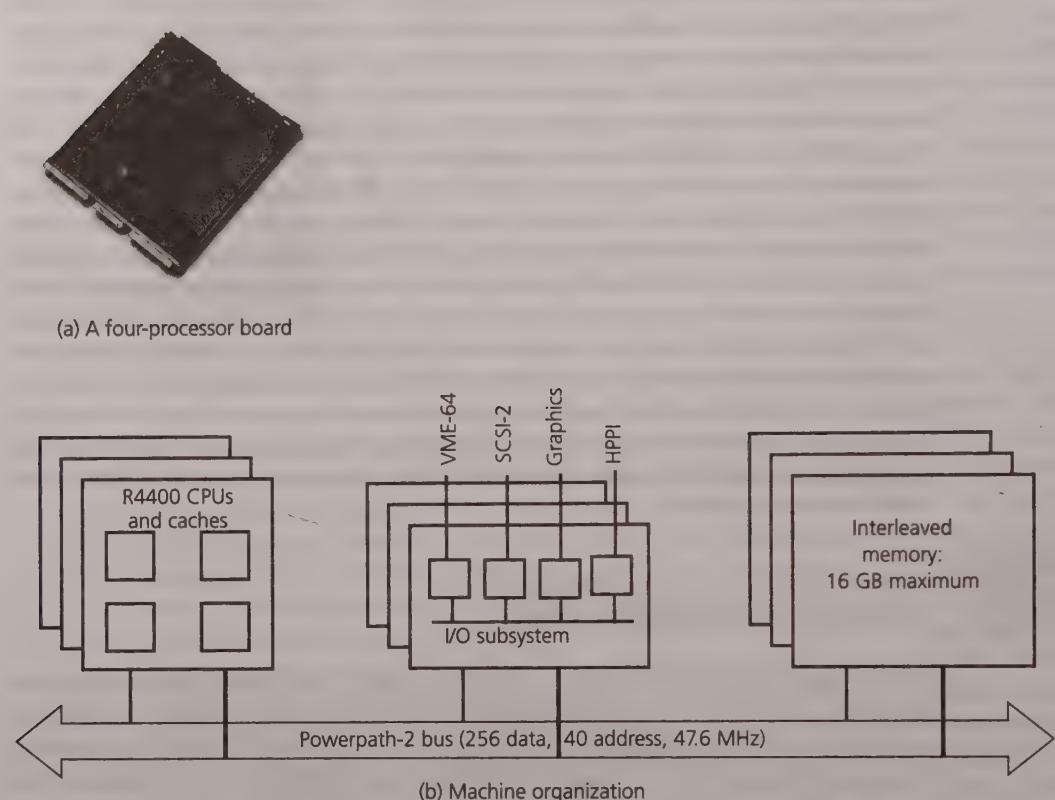


FIGURE 6.11 The SGI Challenge multiprocessor. With 4 processors per board, the 36 processors consume nine bus slots. The Challenge can support up to 16 GB of eight-way interleaved main memory. The I/O boards each provide a separate 320-MB/s I/O bus, to which other standard buses and devices interface. The system bus has a separate 40-bit address path and a 256-bit datapath, plus command, and other signals, and supports a peak bandwidth of 1.2 GB/s. The bus is split transaction, and up to eight requests can be outstanding on the bus at any given time. Photo: CHALLENGE is a trademark of Silicon Graphics, Inc.

number of processors. Although some memory is physically local to a pair of processors, all of memory is accessed through the system bus and hence is of uniform access time. The board containing the memory for a particular address is called the *home board* of the address. Each I/O card provides two independent 64-bit \times 25-MHz SBUS I/O buses, so the I/O bandwidth scales with the number of I/O cards. The total disk storage can be tens of terabytes. The operating system is Solaris UNIX. Figure 6.12 shows a block diagram of the Sun Enterprise system.

The next few subsections describe the SGI Challenge architecture and characterize some of its performance attributes. The following subsections do the same for the Sun Enterprise 6000.

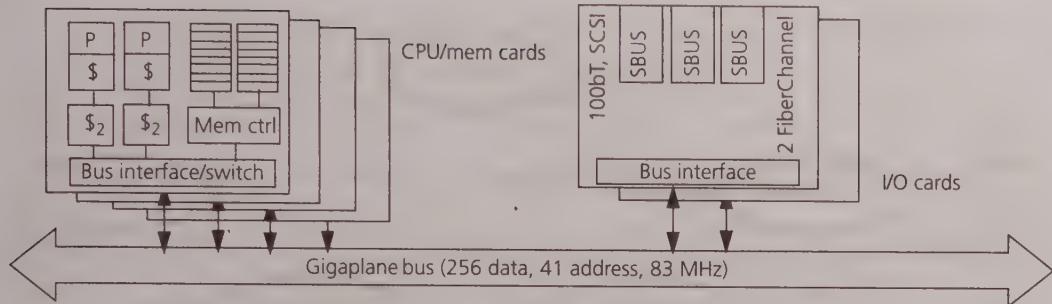


FIGURE 6.12 The Sun Enterprise 6000 multiprocessor. The system provides 16 bus slots that can be occupied by either processor or I/O boards, but there must be at least one of each. The processor board contains two processors and two banks of memory, which are uniformly accessible to all boards. The I/O board provides connectors for multiple independent peripheral buses and appears like another cache controller on the system bus. The split-transaction bus allows up to 112 outstanding transactions at a time.

6.5.1 SGI Powerpath-2 System Bus

The system bus forms the core interconnect for all components in the system. As a result, its design is affected by the requirements of all other components, and design choices made for it affect the design of other components in turn. The design choices for buses include multiplexed versus nonmultiplexed address and data buses, a wide (e.g., 256- or 128-bit) versus narrower (64-bit) data bus, clock rate of the bus (affected by signaling technology used, length of bus, and number of slots on bus), split-transaction versus atomic design, flow control strategy, and so on. The Powerpath-2 bus is nonmultiplexed, having a 256-bit-wide data portion and a separate 40-bit-wide address portion, plus command and other signals. It is clocked at 47.6 MHz, and it is a split-transaction design supporting eight outstanding read requests. While the wide datapath implies that the hardware cost of connecting to the bus is higher (it requires multiple bit-sliced chips to interface to it), the benefit is that the high bandwidth of 1.2 GB/s can be achieved at a reasonable clock rate. The bus supports sixteen slots, nine of which can be populated with 4-processor boards to obtain a 36-processor configuration. The width of the bus also affects (and is affected by) many other design issues. For example, the block size chosen for the cache closest to the bus (here the second-level cache) is 128 bytes, implying that the whole cache block can be transferred in four bus clocks; because of the dead cycle between transfers, a much smaller block size would have resulted in less effective use of the bus pipeline or a more complex design. Also, the individual board is fairly large in order to support such a large bus connector. The bus interface occupies roughly 20% of the board, in a strip along the edge, making it natural to place multiple processors on each board.

Let us look at the Powerpath-2 bus design in a little more detail. The bus consists of a total of 329 signals: 256 data, 8 data parity, 40 address, 8 command, 2 address +command parity, 8 data resource ID, and 7 miscellaneous. The types and variations

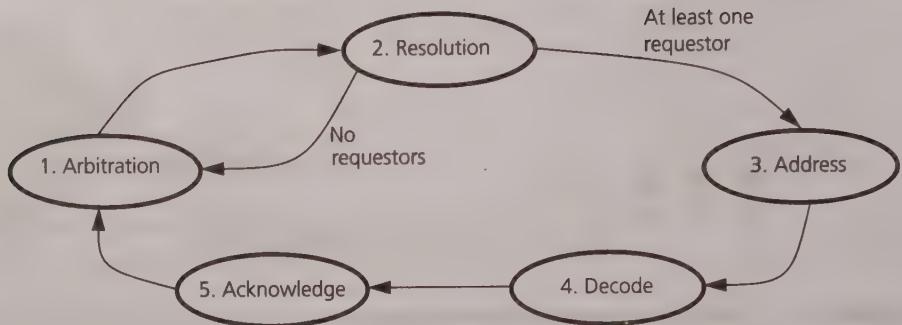


FIGURE 6.13 Powerpath-2 bus state transition diagram. The bus interfaces of all boards attached to the system bus synchronously cycle through the five states shown in the figure; this is also the duration of all address and data transactions on the bus. When the bus is idle, however, it only loops between states 1 and 2.

of transactions on the bus are small, and all transactions take exactly 5 cycles, as discussed earlier in our example design. All bus controller ASICs execute the following five-state machine synchronously: arbitration, resolution, address, decode, and acknowledge. When no transactions are occurring, each bus controller drops into a two-state idle machine. The shorter, two-state idle machine allows new requests to arbitrate immediately rather than waiting for the arbitration state to occur in the five-state machine. (Two states are required, rather than one, to prevent different requestors from driving arbitration lines on successive cycles.) Figure 6.13 shows the state machine underlying the basic bus protocol.

Since the bus is a split-transaction design, the address and data buses must be arbitrated for separately. In the arbitration cycle, the 48 address+command lines are used for arbitration. The lower 16 of these lines are used by the 16 boards (one per board) to request the data bus, and the middle 16 lines are used for address bus arbitration. For transactions that require both address and data buses together (e.g., write backs), corresponding bits for both buses can be set high. The top 16 lines are used to make *urgent*, or high-priority, requests. Urgent requests are used to avoid starvation, for example, if a processor times out waiting to get access to the bus. The availability of urgent requests allowed the designers considerable flexibility in favoring the service of some requests over others for performance reasons (e.g., reads are given preference over writes) while still being confident that no requestor will get starved.

Figure 6.14, which expands upon Figure 6.8, shows the cycles during which various bus lines are driven and their semantics. At the end of the arbitration cycle, all bus interface ASICs capture the 48-bit state of the address+command lines and thus see the bus requests from all boards. A distributed arbitration scheme is used; every controller sees all of the bus requests, and in the resolution cycle, each one independently computes the same winner. While distributed arbitration consumes more of

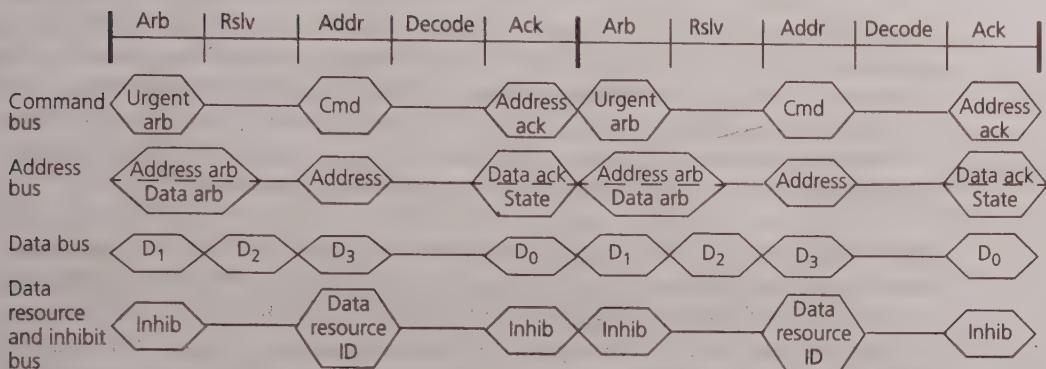


FIGURE 6.14 Powerpath-2 bus timing diagram. During the arbitration cycle, the 48 bits of the address+command bus indicate requests from the 16 bus slots for data transactions, address transactions, and urgent transactions. Each bus interface determines the results of the arbitration independently following a common algorithm. For the address request that is granted, the address+command is transferred in the address cycle, and the requests can be NACKed in the acknowledgment cycle. Similarly, for the data request that is granted, the tag associated with it (data resource ID) is transferred in the address cycle; it can be NACKed in the ack cycle, or the data is transferred in the following D₀–D₃ cycles (where D₀ is the ack cycle).

ASIC's gate resources, it saves the latency incurred by a centralized arbitrator of communicating winners to everybody via bus grant lines.

During the address cycle, the address bus winner drives the address and command buses with corresponding information. Simultaneously, the data bus winner drives the *data resource ID* line corresponding to the response. (The data resource ID is the 3-bit global tag that was assigned to the read request when it was originally issued on the bus. The use of the global tags is described in Section 6.4.2.)

During the decode cycle, no signals are driven on the address bus. Internally, each bus interface slot decides how to respond to this transaction. For example, if the transaction is a write back and the memory system currently has insufficient buffer resources to accept the data, in this cycle it will decide that it must NACK (negative acknowledge or reject) this transaction on the next cycle so that the transaction can be retried at a later time. In addition, all slots prepare to supply the proper cache coherence information.

During the acknowledge cycle, each bus interface slot responds to the data/address bus transaction. The 48 address+command lines are used as follows. The top 16 lines indicate if the device in the corresponding slot is rejecting the address bus transaction due to insufficient buffer space. Similarly, the middle 16 lines are used to possibly reject the data bus transaction. The lowest 16 lines indicate the cache state of the block (present versus not present) being transferred on the data bus. These lines help determine the state in which the data block will be loaded in the requesting processor (e.g., exclusive versus shared). Finally, in case one of the processors

has not finished its snoop by this cycle, it indicates so by asserting the corresponding inhibit line. (The data resource ID lines double as inhibit lines during the acknowledgment and arbitration cycles.) It continues to assert this line until it has finished the snoop. If the snoop indicates a clean cache block, the snooping node simply drops the inhibit line and allows the requesting node to accept the memory's response. If the snoop indicates a dirty block, the node rearbitrates for the data bus, supplies the latest copy of the data, and only then drops the inhibit line.

For data bus transactions, once a slot becomes the master, the 128 bytes of cache block data are transferred in four consecutive cycles over the 256-bit-wide datapath. This four-cycle sequence begins with the acknowledgment cycle and ends at the address cycle of the following five-cycle bus phase. Since the 256-bit-wide datapath is used only for four out of five cycles, the maximum possible efficiency of these data lines is 80%. In some sense, though, this is the best that could be done; the signaling technology used in the Powerpath-2 bus requires one-cycle turnaround time between different controllers driving the lines.

6.5.2 SGI Processor and Memory Subsystems

In the Challenge architecture, each board contains multiple processors. To reduce the cost of interfacing to the bus, many of the bus interface chips are shared between the processors. Figure 6.15 shows the high-level organization of the processor board.

The processor board uses three different types of chips to interface to the bus and to support cache coherence. There is a single A-chip for all four processors that interfaces to the address bus. It contains logic for distributed arbitration, the eight-entry request tables storing currently outstanding transactions on the bus (see Section 6.4), and other control logic for deciding when transactions can be issued on the bus and how to respond to them. It passes on requests observed on the bus to the CC-chip (one for each processor), which uses a duplicate set of tags to determine the presence of that memory block in the local cache and communicates the results back to the A-chip. All requests from the processor also flow through the CC-chip to the A-chip, which then presents them on the bus. To interface to the 256-bit-wide data bus, four bit-sliced D-chips are used. The D-chips are quite simple and are shared among the processors; they provide limited buffering capability and simply pass data between the bus and the CC-chip associated with each processor (cache).

The Challenge main memory subsystem uses high-speed buffers to fan out addresses to a 576-bit-wide internal DRAM bus. The 576 bits consist of 512 bits of data and 64 bits of error correcting code (ECC), allowing for single-bit in-line correction and double-bit error detection. Fast page-mode access allows an entire 128-byte cache block to be read in two memory cycles, and data buffers pipeline the response to the 256-bit-wide system data bus. Twelve bus cycles (approximately 250 ns) after the address appears on the bus, the response data appears on the data bus. A single memory board can hold 2 GB of memory and supports a two-way interleaved memory system that is capable of saturating the 1.2-GB/s system bus.

Given the raw latency of 250 ns that the main memory subsystem takes, it is instructive to see the overall latency for a second-level cache miss experienced by

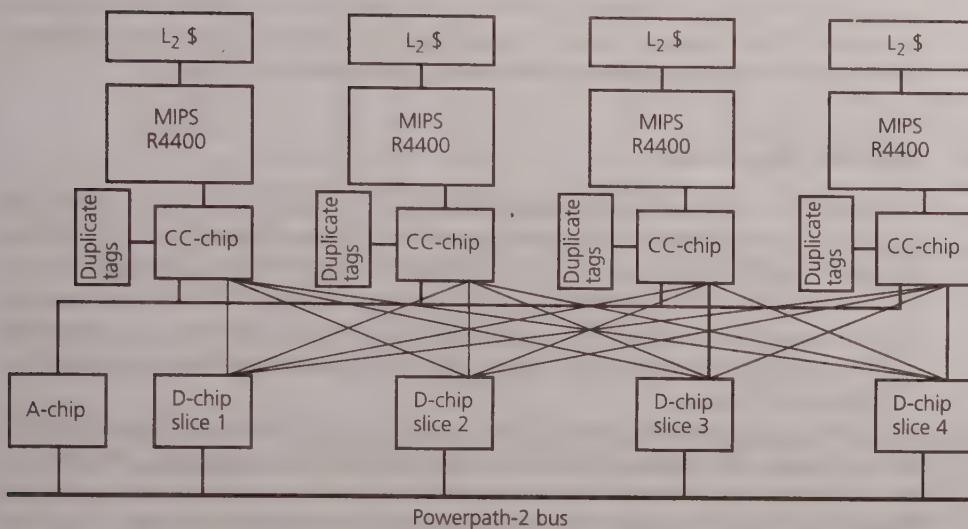


FIGURE 6.15 Organization and chip partitioning of the SGI Challenge processor board. To reduce the number of bus slots required to support 36 processors, 4 processors are put on each board. To maintain coherence and to interface to the bus, there is one cache coherence (CC) chip per processor, one shared A-chip that keeps track of requests to and from all four processors and interfaces to the address bus, and four shared bit-sliced D-chips that interface to the 256-bit-wide data bus.

the processor. On the Challenge, this number is close to 1 μ s. It takes approximately 300 ns for the request to first appear on the bus; this includes time taken for the processor to realize that it has a first-level cache miss and then a second-level cache miss and then to filter through the CC-chip down to the A-chip. It takes approximately another 400 ns for the complete cache block to be delivered to the D-chips across the bus. These include the 3 bus cycles until the address stage of the request transaction, 12 bus cycles (250 ns) to access the main memory, and another 5 cycles for the data transaction to deliver the data over the bus. Finally, it takes another 300 ns for the data to flow through the D-chips, through the CC-chip, and through the 64-bit-wide interface onto the processor chip (16 cycles for the 128-byte cache block) where the data is loaded into the primary cache, and then to restart the processor pipeline.^{4,5}

-
4. Note that on both outgoing and return paths, the memory request passes through an asynchronous boundary. This adds a double synchronizer delay in both directions, about 30 ns on average in each direction. The benefit of decoupling is that the CPU can run at a different clock rate than the system bus, thus allowing for migration to higher-clock-rate CPUs while keeping the same bus clock rate. The cost, of course, is the extra latency.
 5. The newer generation of processor, the MIPS R10000, allows the processor to restart after only the needed word has arrived, without having to wait for the complete cache block to arrive. This critical-word restart mechanism reduces the miss latency.

To maintain cache coherence, the SGI Challenge uses the Illinois MESI protocol by default. It also supports update transactions. Interactions of the cache coherence protocol and the split-transaction bus are as described in Section 6.4.

6.5.3 SGI I/O Subsystem

To support the high computing power provided by multiple processors, careful attention needs to be devoted to providing matching I/O capability. The SGI Challenge provides scalable I/O performance by allowing multiple I/O cards to be placed on the system bus, each card providing a local 320-MB/s proprietary HIO I/O bus. Personality ASICs are provided to act as an interface between the I/O bus and standards-conforming (e.g., Ethernet, VME, SCSI, HPPI) and nonstandards-conforming (e.g., SGI Graphics) devices.

Figure 6.16 shows a block-level diagram of the SGI Challenge's PowerChannel-2 I/O subsystem. The bus is a 64-bit-wide multiplexed address/data bus that runs off the same clock as the system bus. It supports split read transactions, with up to four outstanding transactions per device. In contrast to the main system bus, it uses centralized arbitration, as latency is much less of a concern. However, arbitration is pipelined so that bus bandwidth is not wasted. Since the HIO bus supports several different transaction lengths (it does not require every transaction to handle a full cache block of data), transactions are required to indicate their length at time of request, and the arbiter uses this information to ensure more efficient utilization of the bus. The narrower HIO bus allows the personality ASICs to be cheaper than if they were to directly interface to the very wide system bus. In addition, common functionality needed to interface to the system bus is in this way shared by a number of personality ASICs.

HIO interface chips can request read or write DMA transfers to or from any locations in system memory using the full 40-bit system address, make a request for address translation using the mapping resource in the system interface (e.g., for 32-bit VME), request interrupting the processor, or respond to processor I/O (PIO) reads. The system bus interface provides DMA read responses and the results of address translation to the I/O devices and passes on PIO reads to them.

To the rest of the system (processor boards and main memory boards), the system bus interface on the I/O board provides a clean interface; it essentially acts like another processor board. Thus, when a DMA read request makes it through the I/O board's system bus interface onto the system bus, it becomes a Powerpath-2 read, just like one that a processor would issue. Similarly, when a full-cache-block DMA write goes out, it becomes a special block write transaction on the bus that invalidates copies of the block in all processors' caches (in addition to updating main memory). A special transaction is needed because even if a processor has the block dirty in its local cache, we do not want it to write it back in this case.

To support partial-block DMA writes, special care is needed because data must be merged coherently into main memory. To support these partial DMA writes, the system bus interface includes a fully associative, four-block cache that snoops on the Powerpath-2 system bus in the usual fashion. The cache blocks can be in one of only

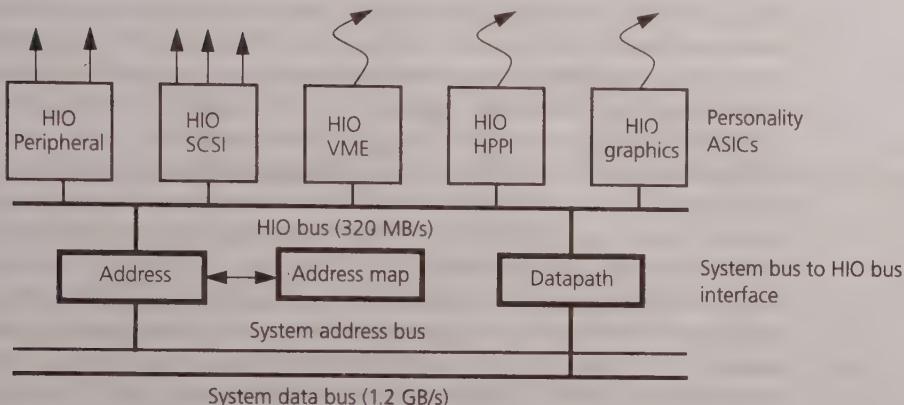


FIGURE 6.16 High-level organization of the SGI Challenge PowerChannel-2 I/O subsystem. Each I/O board provides an interface to the Powerpath-2 system bus and an internal 64-bit-wide HIO I/O bus with peak bandwidth of 320 MB/s. The narrower HIO bus lowers the cost of interfacing to it, and it supports a number of personality ASICs, which in turn support standard buses and peripherals.

two states: invalid or modified. When a partial DMA write is first issued, the block is brought into the four-block cache on the I/O board in modified state, invalidating the copies in all other processors' caches. Subsequent partial DMA writes need not go to the system bus if they hit in this cache, thus increasing the system bus efficiency. This modified block goes to the invalid state and supplies its contents on the system bus (1) on any system bus transaction that accesses this block; (2) when another partial DMA write causes the block to be replaced from the four-block cache; and (3) on any HIO bus read transaction that accesses the block. While DMA reads could have also used this four-block cache, the designers felt that partial DMA reads were rare, and the gains from such an optimization would have been minimal.

The address map RAM in the system bus interface provides general-purpose address translation for I/O devices to access main memory. For example, it may be used to map small address spaces such as VME-24 or VME-32 into the 40-bit physical address space of the Powerpath-2 bus. Two types of mapping are supported: one-level and two-level. One-level mappings simply return one of the 8-K entries in the mapping RAM, where by convention each entry maps 2 MB of physical memory. In the two-level scheme, the map entry points to the page tables in main memory. Each 4-KB page has its own entry in the second-level table, so virtual pages can be arbitrarily mapped to physical pages. Note that PIO requests (from the processors) face a similar translation problem when going down to the I/O devices. Such translation is not done using the mapping RAM but is directly handled by the personality ASIC interface chips.

The final issue that we examine for I/O is flow control. All requests proceeding from the I/O interfaces to the Powerpath-2 system bus are implicitly flow controlled;

that is, the HIO interface will not issue a read on the system bus unless it has buffer space reserved for the response. Similarly, the HIO arbiter will not grant the HIO bus to a requestor unless the system interface has room to accept the transaction. In the other direction, from the processors to the I/O devices, however, PIOs can arrive unsolicited, and they need to be explicitly flow controlled.

The explicit flow control solution used in the Challenge system is to make the PIOs appear to the HIO interface ASICs as if they were solicited. After reset, HIO interface chips (e.g., HIO-VME, HIO-HPPI) transmit their available PIO buffer space to the system bus interface using special requests called IncPIO requests. The system bus interface maintains this information in a separate counter for each HIO device. Every time a PIO is sent to a particular device, the corresponding count is decremented. Every time that device retires a PIO, it issues another IncPIO request to increment its counter. If the system bus interface receives a PIO for a device that has no buffer space available, it rejects (NACKs) that request on the system bus, and the request must be retried later.

6.5.4 SGI Challenge Memory System Performance

The access time for various levels of the SGI Challenge memory system can be determined using the simple read microbenchmark from Chapter 4. Recall that the microbenchmark measures the average access time in reading elements of an array of a given size with a certain stride. Figure 6.17 shows the read access time for a range of sizes and strides. Each curve shows the average access time for a given size as a function of the stride. Arrays smaller than 32 KB fit entirely in the first-level cache. Second-level cache accesses have an access time of roughly 75 ns, and the inflection point at 16-byte stride shows that the transfer size between the L₂ and L₁ caches is 16 bytes. The second bump shows the additional penalty of roughly 140 ns for a TLB miss and reveals that the page size is 8 KB. (Can you think why the time per miss drops back as the stride increases further?) Starting with a 2-MB array, accesses miss in the 1-MB L₂ cache, and we see that the combination of the L₂ controller, the Powerpath bus protocol, and DRAM access results in an access time of roughly 1,150 ns. The minimum bus protocol of 13 cycles from request to reply accounts for a little under 300 ns of this time, as discussed earlier. TLB misses add roughly 200 ns to this 1,150-ns figure. A simple ping-pong microbenchmark, in which a pair of nodes each spins on a flag until the flag indicates their turn and then sets the flag to signal the other, shows a round-trip time of 6.2 μ s.

6.5.5 Sun Gigaplane System Bus

The Sun Gigaplane is also a nonmultiplexed, split-transaction bus with 256-bit data lines and 41-bit physical addresses but is clocked at 83.5 MHz. It is a *centerplane design*, a bus wiring and connection assembly that allows cards to plug into both sides, rather than a single-sided backplane. The total length of the bus is 18 inches, so eight boards can plug into each side with 2 inches of cooling space between boards and 1-inch spacing between connectors. In sharp contrast to the SGI Chal-

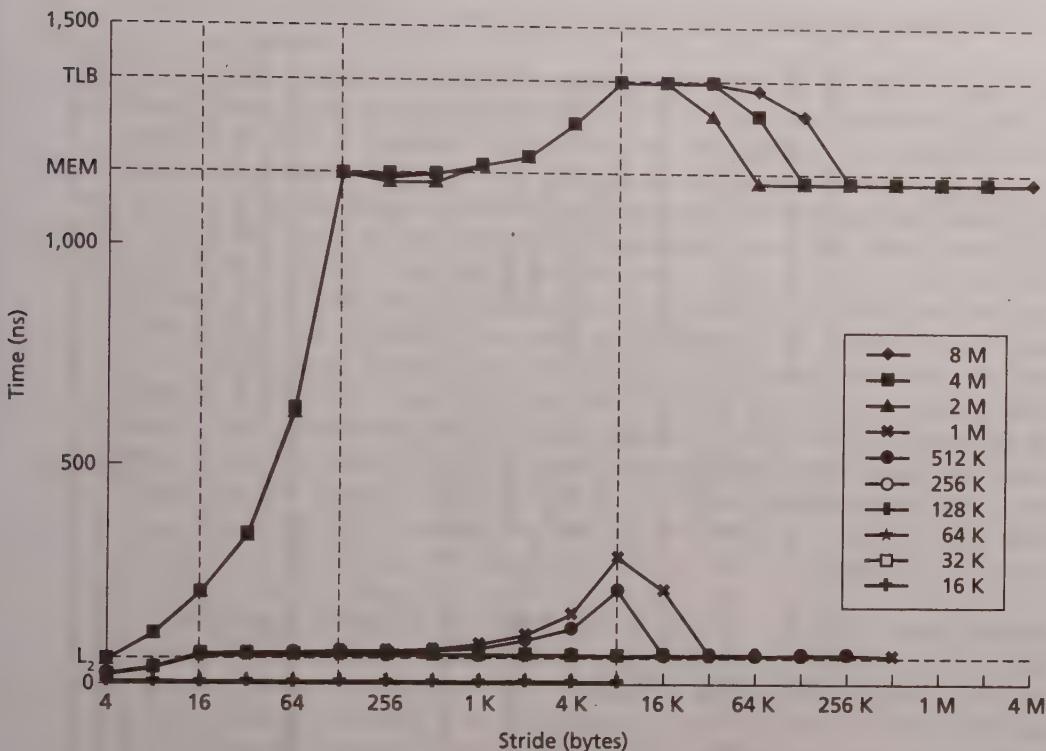


FIGURE 6.17 Read microbenchmark results for the SGI Challenge. Each curve is for an array of the size shown in the legend. The datapoints for array sizes 32 K to 256 K are so similar that they cannot be easily distinguished.

lenger Powerpath-2 bus, the bus can support up to 112 outstanding transactions, including up to 7 from each board, so it is designed for devices that can sustain multiple outstanding transactions, such as lockup-free caches. The electrical and mechanical design allows for live insertion (hot plugging) of processing and I/O modules.

The bus consists of a total of 388 signals: 256 data, 32 ECC, 43 address (with parity), 7 ID tag, 18 arbitration, and a number of configuration signals. The electrical design allows for turnaround between data transfers with no dead cycles. Emphasis is placed on minimizing the latency of operations, and the protocol (illustrated in Figure 6.18) is quite different from that on the SGI Challenge. A novel collision-based speculative arbitration technique is used to avoid the cost of bus arbitration. When a requestor arbitrates for the address bus, if the address bus is not scheduled to be in use from the previous cycle, it speculatively drives its request on the address bus in the arbitration cycle itself. If no other requestors are in that cycle, it wins arbitration and has already passed the address, so it can continue with the remainder of the transaction. If a request collision occurs, the requestor that wins

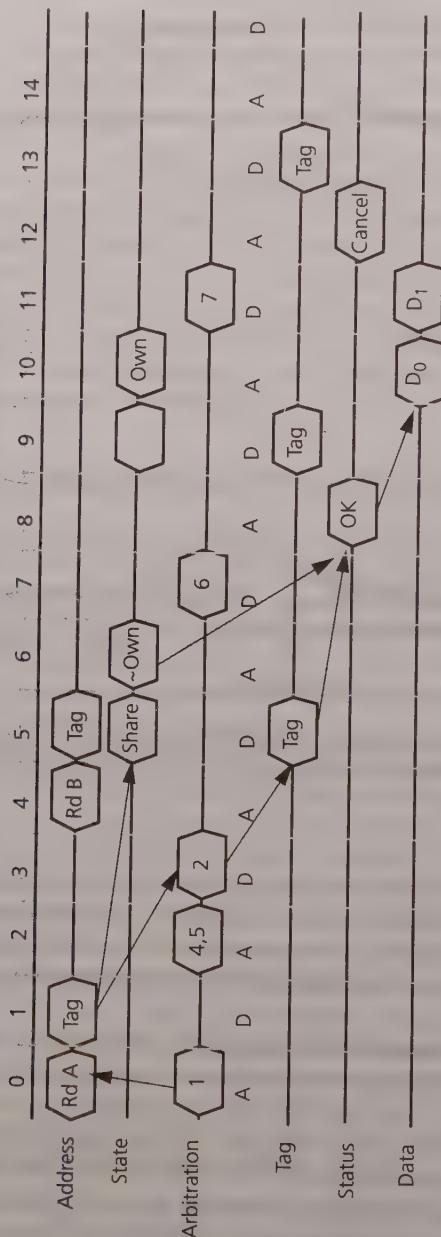


FIGURE 6.18 Sun Gigaplane signal timing for BusRd with fast address arbitration. The pipeline slots occupied by two BusRd transactions are shown, one unshaded and one shaded. The horizontal bold lines show the different components of the bus, and the vertical dotted lines delineate bus cycles. “A” and “D” under the arbitration component of the bus indicate address and data bus arbitration cycles. The arrows indicate the path of the first BusRd transaction. Board 1 initiates a read transaction with fast arbitration (so the address is successfully driven in the same cycle), which is responded to by home board 2. The snoop result indicates that no cache is the owner of the block, so the home board drives the response on the data lines. For the second BusRd transaction, boards 4 and 5 collide during address bus arbitration; board 4 wins and initiates a read transaction. Home board 6 arbitrates for the data bus and then cancels its response because the snoop result indicates ownership by a cache. Eventually the owning cache, on board 7, responds with the data. Board 5's retired transaction is not shown.

arbitration simply drives the address again in the next cycle, as it would with conventional arbitration.

The 7-bit tag associated with the request is presented on the address bus in the cycle following the address (see Figure 6.18). The snoop state is associated with the address phase, not the data phase. Five cycles after the address, all boards assert their snoop signals (shared, owned, mapped, and ignore) on the “state” bus lines. In the meantime, the board responsible for the memory address (the home board) can request the data bus three cycles after the address, before the snoop result. The DRAM access can be started speculatively as well. When the home board wins arbitration, it must assert the tag bus lines two cycles later, informing all devices of the approaching data transfer. Three cycles after driving the tag and two cycles before the data, the home board drives a status signal, which may indicate that the data transfer is canceled if some cache owns the block (as detected in the snoop state). The owner places the data on the bus by arbitrating for the data bus, driving the tag, and driving the data. Figure 6.18 shows a second (gray) read transaction, which experiences a collision in arbitration, so the address is supplied in the conventional slot. The snoop for this transaction indicates ownership by a cache, so the home board cancels its data transfer. Later, that owning cache arbitrates for the data bus and drives the data response.

Like the SGI Challenge, invalidations are ordered by the BusRdX requests appearing on the address bus and are handled in FIFO fashion by the cache subsystems; thus, no explicit acknowledgment of invalidation completion is required. To maintain sequential consistency, it is still necessary to gain arbitration for the address bus before allowing the writing processor to proceed with memory operations past the write.⁶

6.5.6 Sun Processor and Memory Subsystem

In the Sun Enterprise, each processing board has two processors, each with external L₂ caches, and two banks of memory connected through a crossbar, as shown in Figure 6.19. Data lines within the UltraSparc module are buffered to drive an internal bus, called the UPA (universal port architecture) with an internal bandwidth of 1.3 GB/s. A very wide path to memory is provided so that a full 64-byte cache block can be read in a single memory cycle, which is two bus cycles in length. The address controller adapts the UPA protocol to the Gigaplane protocol, realizes the cache coherence protocol, provides buffering, and tracks the potentially large number of outstanding transactions. It maintains a set of duplicate tags, called D-tags, for the L₂ cache. To ensure cache coherence, even accesses to the local memory module from a processor go through the address controller.

Although the UltraSparc implements a five-state MOESI protocol in the L₂ caches, the D-tags maintain an approximation using only three states: owned, shared, and invalid. They essentially combine states that are handled identically at the Gigaplane

6. The Sparc V9 specification weakens the consistency model in this respect to allow the processor to employ write buffers, which we discuss in more depth in Chapter 9.

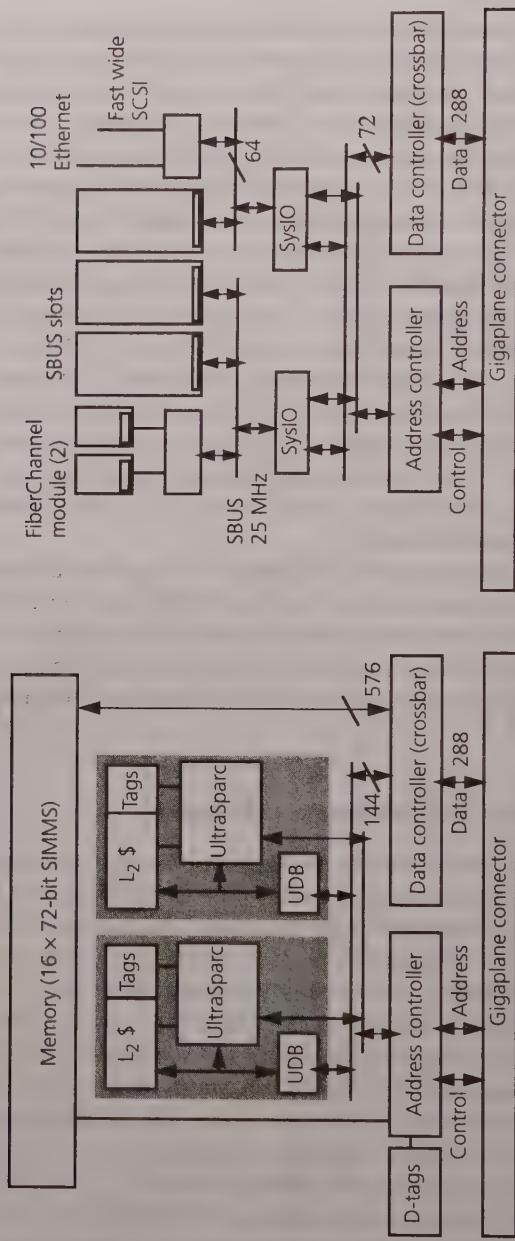


FIGURE 6.19 Organization of the Sun Enterprise processing and I/O boards. The processing board (left) contains two UltraSparc modules with L₂ caches on an internal bus and two wide memory banks interfaced to the system bus through two ASICs. The address controller adapts between the two bus protocols and implements the cache coherence protocol. The data controller is essentially a crossbar. The I/O board (right) uses the same two ASICs to interface to two I/O controllers. The SysIO ASICs appear to the bus as a one-block cache that follows the coherence protocol. On the other side, they support independent I/O buses and interfaces to FiberChannel, Ethernet, and SCSI.

level. In particular, the address controller needs to know if the L₂ cache has a copy of a block and if it is an exclusive copy. It does not need to know if that block is clean or dirty. For example, on a BusRd the block will need to be flushed onto the bus if it is in the L₂ cache in any of the following three states: modified, owned (flushed since last modified), or exclusive (not shared when read and not modified since); thus, the D-tags represent only owned. This has the advantage that the address controller need not be informed when the UltraSparc elevates a block from exclusive to modified. It will be informed of a transition from invalid, shared, or owned to modified because in these cases it needs to initiate a bus transaction.

6.5.7 Sun I/O Subsystem

An Enterprise I/O board uses the same bus interface ASICs as the processing board, but the internal bus is only half as wide and there is no memory path. Externally, the I/O boards only do cache-block-sized transactions, just like the processing boards, in order to simplify the design of the main system bus. The SysIO ASICs implement a single-block cache, which follows the coherence protocol, on behalf of the I/O devices. Internally, two independent 64-bit 25-MHz SBUSs are supported. One of these supports two dedicated FiberChannel modules providing a redundant, high-bandwidth interconnect to large disk storage arrays. The other provides dedicated Ethernet and fast wide SCSI connections. In addition, three SBUS interface cards can be plugged into the two buses to support arbitrary peripherals, including a 622-MB/s ATM interface. The I/O bandwidth, the connectivity to peripherals, and the cost of the I/O subsystem scales with the number of I/O cards.

6.5.8 Sun Enterprise Memory System Performance

The access time for various levels of the Sun Enterprise via the read microbenchmark is shown in Figure 6.20. Arrays of 16 KB or less fit entirely in the first-level cache. Level 2 cache accesses have an access time of roughly 40 ns, and the inflection point shows that the transfer size between these levels is 16 bytes. With a 1-MB array, accesses miss in the L₂ cache, and we see that the combination of the L₂ controller, bus protocol, and DRAM access result in an access time of roughly 300 ns. The minimum bus protocol of 11 cycles at 83.5 MHz accounts for 130 ns of this time. TLB misses add roughly 340 ns to the miss penalty since the machine has a software TLB handler. The simple ping-pong microbenchmark, in which a pair of nodes each spins on a flag until it indicates their turn and then sets the flag to signal the other, shows a round-trip time of 1.7 μ s, roughly five memory accesses.

6.5.9 Application Performance

Now that we have an understanding of the machines and their microbenchmark performance, let us examine the performance obtained on our parallel applications. Absolute application performance for commercial machines is not presented in this book; instead, the focus is on performance improvements due to parallelism. Let us

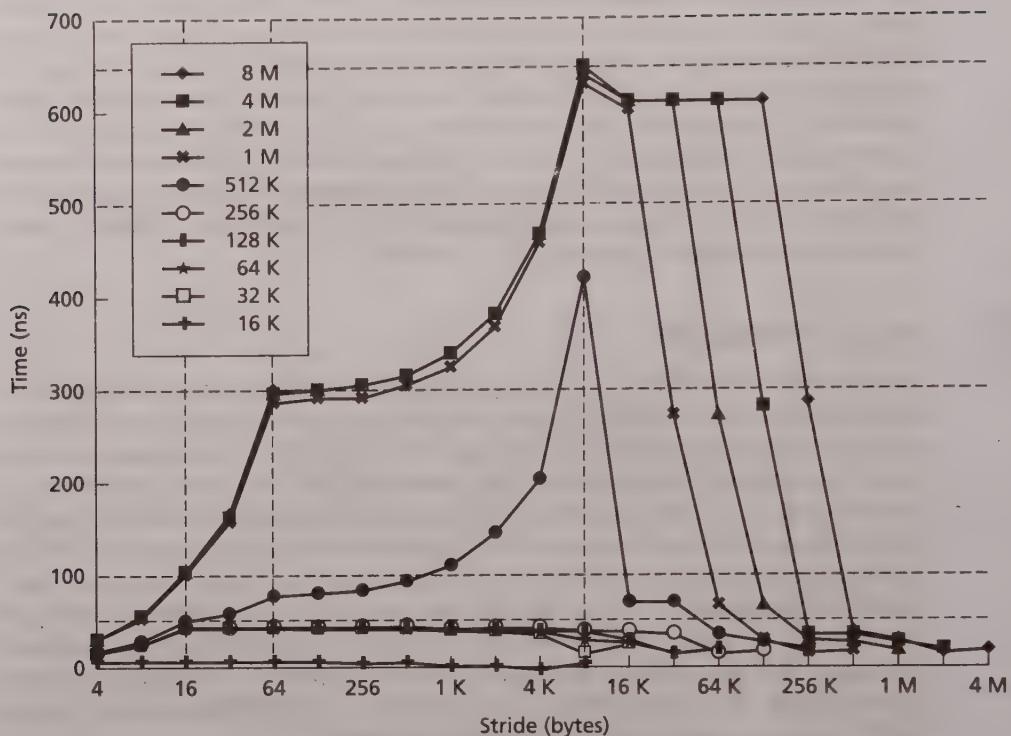


FIGURE 6.20 Read microbenchmark results for the Sun Enterprise. Each curve is for an array of the size shown in the legend.

first look at application speedups and then at scaling, using only the SGI Challenge for illustration.

Application Speedups

Figure 6.21 shows the speedups obtained on our six parallel programs for two data set sizes each. We can see that the speedups are quite good for most of the programs, with the exception of the Radix sorting kernel. Examining the breakdown of execution time for the sorting kernel shows that the vast majority of the time is spent stalled on data access. The shared bus simply gets swamped with the data and coherence traffic due to the permutation phase of the sort, and the resulting contention destroys performance. The contention also leads to severe load imbalances in data access time and, hence, time spent waiting at global barriers, even though busy time is well balanced. Contention is unfortunately not alleviated much by increasing the problem size since the communication-to-computation ratio, and hence the bandwidth demand, in the permutation phase is independent of problem size (see Section 4.4.1). The results shown are for a radix value of 256, which delivers the

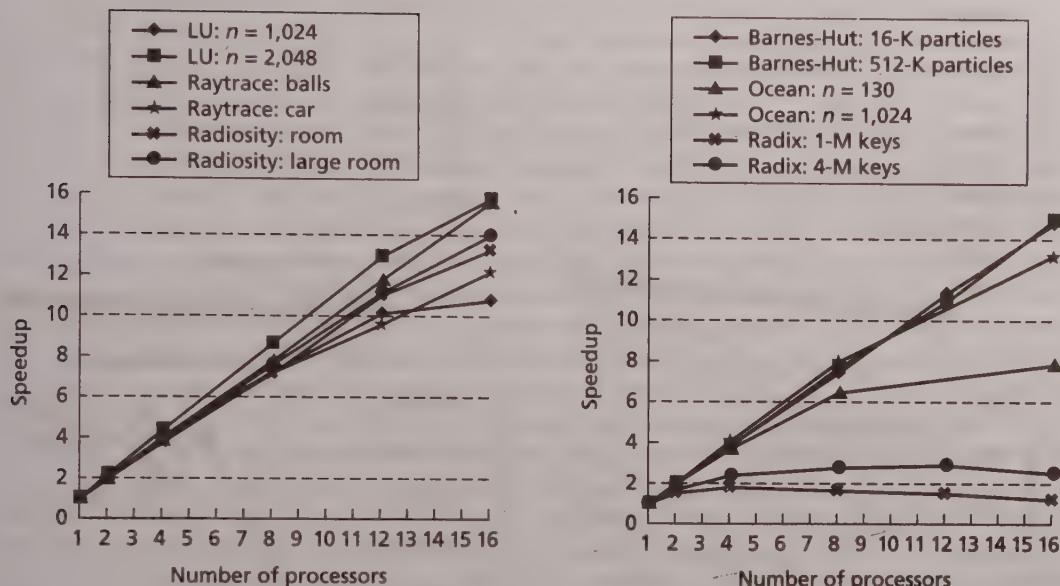


FIGURE 6.21 Speedups for the six parallel applications on the SGI Challenge. The block size for the blocked LU factorization is 32×32 .

best performance over the range of processor counts for both problem sizes. Barnes-Hut, Raytrace, and Radiosity speed up very well even for the relatively small input problems used. LU does too, and the bottleneck for the smaller problem at 16 processors is primarily load imbalance as the factorization proceeds along the matrix. Finally, the bottleneck for the small Ocean problem size is both the high communication-to-computation ratio and the imbalance this generates since some partitions have fewer neighbors than others. Both problems are alleviated by running larger data sets.

Scaling

Let us now examine the impact of scaling for a few of the programs. According to the discussion in Chapter 4, we look at the speedups under the different scaling models as well as at how the work done and the data set size used change. Figure 6.22 shows the results for the Barnes-Hut and Ocean applications. Naive TC (time-constrained) or MC (memory-constrained) scaling refers to scaling only the parameter that chiefly governs the data set size—the number of bodies or grid points (n)—without changing the other application parameters (accuracy or the number of time-steps). It is clear that the work done under realistic MC scaling grows much faster than linearly in the number of processors in both applications, so the parallel execution time grows very quickly. The number of bodies or grid points that can be

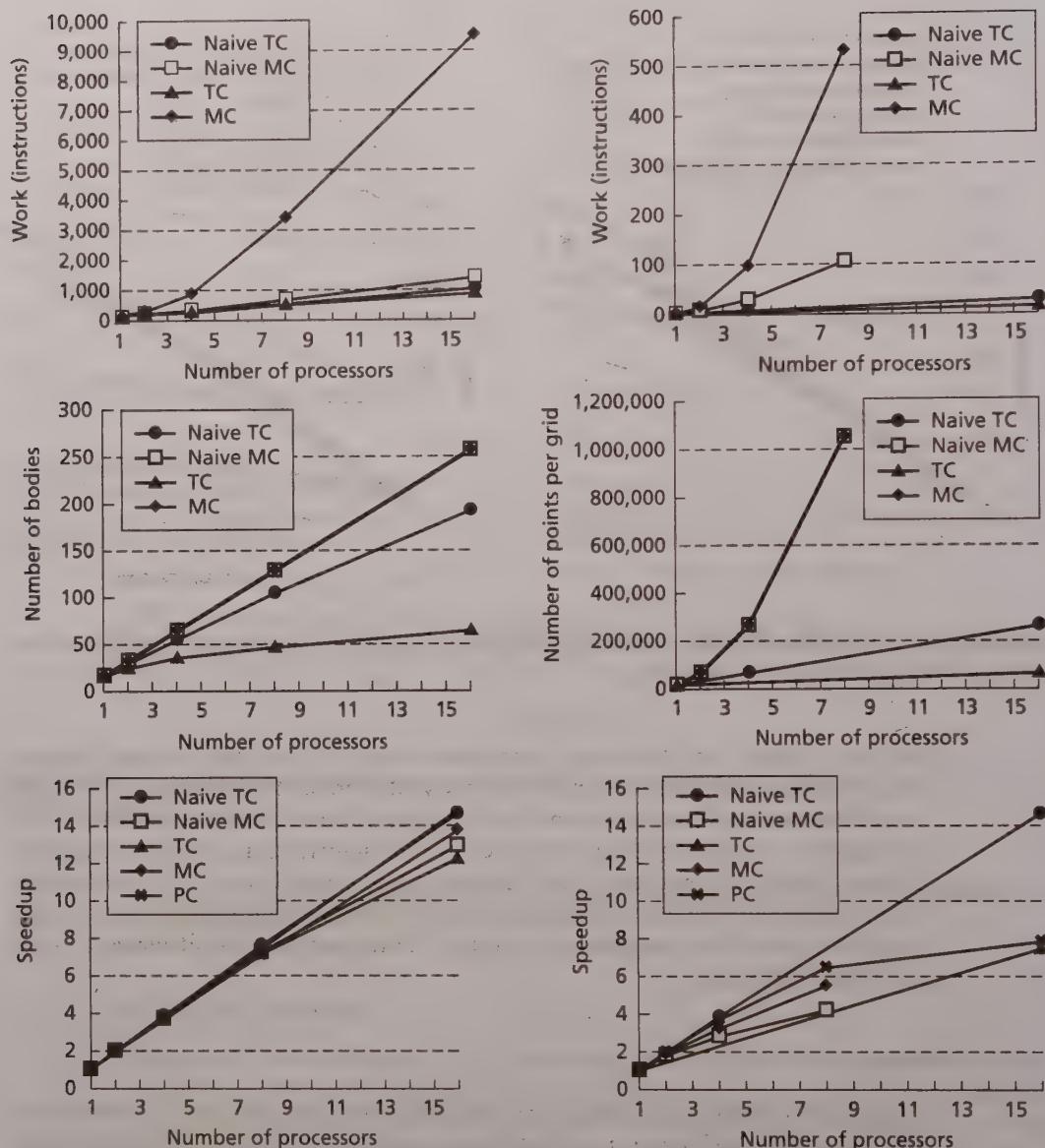


FIGURE 6.22 Scaling results for Barnes-Hut (left) and Ocean (right) on the SGI Challenge. The graphs show the scaling of work done, data set size (measured in number of bodies or grid points), and speedups under different scaling models. PC, TC, and MC refer to problem-constrained, time-constrained, and memory-constrained scaling, respectively. The baseline problem sizes are 16-K bodies for Barnes-Hut and 130×130 grids for Ocean. The top set of graphs shows that the work needed to solve the problem grows very quickly under realistic MC scaling for both applications. The middle set of graphs shows that the data set size that can be run grows much more quickly under MC or naive TC scaling than under realistic TC scaling. The impact of the scaling model on speedup is much larger for Ocean than for Barnes-Hut, primarily because the communication-to-computation ratio is much more strongly dependent on data set size and number of processors in Ocean.

simulated under TC scaling grows much more slowly than under MC and also much more slowly than under naive TC scaling, where it is the only application parameter scaled. Scaling the other application parameters causes the work done and execution time to increase, leaving much less room to grow n .

The speedups under different scaling models are measured as described in Chapter 4. Consider the Barnes-Hut galaxy simulation, where the speedups are quite good for this size of machine under all scaling models. The differences can be explained by examining the major performance factors. The communication-to-computation ratio in the force calculation phase depends primarily on the number of bodies. Another important factor that affects performance is the ratio of work done in the force calculation phase, which speeds up well, to that done in the tree-building phase, which does not. This ratio tends to increase with greater accuracy in force computation, that is, smaller θ . However, smaller θ (and to a lesser extent greater n) increase the working set size per processor (Singh, Hennessy, and Gupta 1993). The important working set continues to fit in the large second-level cache even under scaling, but the scaled problem that changes θ may have worse first-level cache behavior than the baseline problem does on a uniprocessor. These factors explain why naive TC scaling yields better speedups than realistic TC scaling: the working sets behave better, and the communication-to-computation ratio is more favorable since n grows more quickly when θ and Δt are not scaled.

The speedups for Ocean are quite different under different models. Here too, the major controlling factors are the communication-to-computation ratio, the working set size, and the time spent in different phases. However, all the effects are much more strongly dependent on the grid size relative to the number of processors. Under MC scaling, the communication-to-computation ratio does not change with the number of processors used, so we might expect the best speedups. However, two effects become visible as we scale. First, conflict misses across different grids increase as a processor's partitions of the grids become further apart in the address space. Second, more time is spent in the higher levels of the multigrid hierarchy in the solver, which have worse parallel performance. The latter effect turns out to be alleviated when the accuracy and time-step interval are refined as well (at least, this is beneficial for parallel speedup), so realistic MC scales a little better than naive MC. Under naive TC scaling, the growth in grid size is not fast enough to cause major conflict problems but is fast enough that the communication-to-computation ratio does not increase significantly, so speedups are very good. Realistic TC scaling causes a slower growth of grid size and hence a greater increase in the communication-to-computation ratio, resulting in lower speedups. Clearly, many effects play important roles in determining parallel performance under scaling, and which scaling model is most appropriate for an application affects the results of evaluating a machine.

6.6

EXTENDING CACHE COHERENCE

The snooping-based techniques for achieving cache coherence described in this and the previous chapter extend in many directions. This section examines a few important ones: scaling down with shared caches, scaling in functionality with virtually

indexed caches and translation lookaside buffers (TLBs), and scaling up with non-bus interconnects.

6.6.1 Shared Cache Designs

Grouping processors together to share a level of the memory hierarchy (e.g., the first- or the second-level cache) is a potentially attractive option for shared memory multiprocessors, especially due to packaging considerations such as placing multiple processors on a chip. Compared with each processor having its own cache or memory at that level of the hierarchy, grouping processors together has several potential benefits. The benefits—like the drawbacks to be discussed later—are encountered when sharing at any level of the hierarchy but are most extreme when it is the first-level cache that is shared among processors. The benefits of sharing a cache among a group of processors at a level are as follows:

- It eliminates the need for a cache coherence protocol at this level. In particular, if the first-level cache is shared by all processors, then there are no multiple copies of a cache block and hence no coherence problem at all.
- It reduces the latency of communication within the group. The latency of communication between processors is closely related to the level in the memory hierarchy where they meet. When sharing the first-level cache, communication latency can be as low as 2–10 processor clock cycles. The corresponding latency when processors meet at the main memory level is usually many times larger (see the Challenge and Enterprise case studies). The reduced latency enables finer-grained sharing of data between tasks executed on the different processors.
- Once one processor misses on a piece of data and brings it into the shared cache, other processors in the group that need the data may find it already there and will not have to miss on it at that level of the hierarchy. This is called *prefetching* data across processors. With private caches, each processor would have to incur a miss separately. The reduced number of misses reduces the bandwidth requirements at the next level of the memory and interconnect hierarchy.
- It allows more effective use of long cache blocks. Spatial locality is exploited even when different words on a cache block are accessed by different processors in a group. In addition, since there is no cache coherence protocol within a group at this level, there is no false sharing either. For example, consider a situation where two processors P_1 and P_2 write every alternate word of a large array, and think about the differences when they share a first-level cache and when they have private first-level caches.
- The working sets (code or data) of the processors in a group may overlap significantly, allowing the size of the shared cache to be smaller than the combined size of the private caches if each had to hold its processor's entire working set. This reduction of cache size is especially useful for a multiprocessor on a chip, where silicon area is a significant constraint.

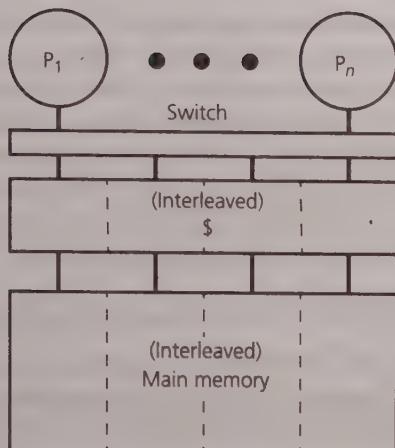


FIGURE 6.23 Generic architecture for a multiprocessor with a first-level cache shared among all processors. The interconnect is placed between the processors and the first-level cache. The shared cache is backed up directly by main memory in this design, and both the cache and the memory system may be interleaved to provide high bandwidth. Of course, such a design is appropriate only for a small number of processors due to bandwidth limitations of the shared cache or switch and the latency imposed by large switches.

- It increases the utilization of the cache hardware. The shared cache does not sit idle because one processor is stalled but, rather, services other references from other processors in the group.
- The grouping fits well with the hierarchy in packaging technologies (cabinets, boards, multichip modules, and chips) and allows us to effectively use emerging packaging technologies to achieve higher computational densities (computation power per unit area).

When sharing a first-level cache, processors are connected to the shared cache by a switch as in Figure 6.23. The switch could be a bus but is more likely a crossbar to allow cache accesses from different processors to proceed in parallel. Similarly, to support the high bandwidth demands imposed by multiple processors, both the cache and the main memory system are interleaved. An early example of such a shared cache architecture is the Alliant FX-8 machine, designed in the early 1980s. An Alliant FX-8 contained up to eight custom processors. Each processor was a pipelined implementation of the 68020 instruction set, augmented with vector instructions, and had a clock cycle of 170 ns. All eight processors were connected using a crossbar to a 512-KB four-way interleaved cache. The cache had 32-byte blocks and was write back, direct mapped, and lockup-free, allowing each processor to have two outstanding misses. The cache-to-processor bandwidth was eight 64-bit words per instruction cycle.

A somewhat different early use of the shared cache approach was exemplified by the Encore Multimax, a contemporary of the FX-8. The Multimax was a snoopy cache-coherent multiprocessor, but each cache supported two processors instead of one (with no need for coherence within a pair). The motivation for Encore at the time was to lower the cost of snooping hardware and to increase the utilization of the cache given the very slow, multiple-CPI processors.

Today, shared first-level caches are being investigated for single-chip multiprocessors, in which four to eight multiprocessors share an on-chip first-level cache. These

can be used in themselves as multiprocessors or as the building blocks for larger systems that maintain coherence across the single-chip shared cache groups. As technology advances and the number of transistors on a chip reaches several tens or hundreds of millions, this approach becomes increasingly attractive. Since interprocessor communication and synchronization within a chip can be quite inexpensive, workstations using such chips will be able to offer very high performance for workloads requiring either fine-grained or coarse-grained parallelism. The question is whether this is a more effective approach or one that uses the available transistors to build more complex processors.

Unfortunately, sharing caches also has several disadvantages and challenges:

- The shared cache has to satisfy the bandwidth requirements from multiple processors, restricting the size of a group. The problem is particularly acute for shared first-level caches, which are therefore limited to very small numbers of processors. Providing the bandwidth needed is one of the biggest challenges of the single-chip multiprocessor approach.
- The hit latency to a shared cache is usually higher than to a private cache at the same level due to the interconnect in between. For shared first-level caches, the imposition of a switch between the processor and the first-level cache means that either the machine clock cycle is elongated or that additional delay slots are added for load instructions in the processor pipeline. The slowdown due to the former is obvious. While compilers have some capability to schedule independent instructions in load delay slots, the success depends on the application. Particularly for programs that don't have a lot of instruction-level parallelism, some slowdown is inevitable. The increased hit latency is aggravated by contention at the shared cache and, correspondingly, the miss latency is also increased by sharing.
- For the preceding reasons, the design complexity for building an effective shared cache is higher than for a private cache.
- Although a shared cache need not be as large as the sum of the private caches it replaces, it is still much larger and, hence, slower than each individual private cache. For first-level caches, this too will either elongate the machine clock cycle or lead to cache access times of multiple processor cycles.
- The converse of overlapping working sets (or constructive interference) is the performance of the shared cache being hurt because of cache conflicts across reference streams from different processors (destructive interference). When a shared cache multiprocessor is used to run workloads with little data sharing (e.g., a parallel compilation or a database or transaction processing workload), the interference in the cache between the data sets needed by the different processors can hurt performance substantially. In scientific computing, where performance is paramount, many programs try to manage their use of the per-processor cache very carefully so that the many arrays they access do not interfere in the cache. All this effort by the programmer or compiler can easily be undone in a shared cache system. Shared caches may require higher associativity than private caches, which may increase their access time as well.

- Finally, at the current time, shared first-level caches do not meet the trend toward using commodity microprocessor technology to build cost-effective parallel machines.

Since many microprocessors already provide snooping support for first-level caches, an attractive approach may be to have private first-level caches and a shared second-level cache among groups of processors. This approach softens both the benefits and the drawbacks of shared first-level caches but may be a good trade-off overall. The shared cache is likely to be large to reduce destructive interference. In practice, packaging considerations also have a very large impact on decisions to share caches.

6.6.2 Coherence for Virtually Indexed Caches

Recall from uniprocessor architecture the trade-offs between physically and virtually indexed caches, that is, caches that are indexed using a physical or virtual address. With physically indexed first-level caches, allowing cache indexing to proceed in parallel with address translation requires that the cache be either very small or very highly associative. This ensures that the bits that do not change under translation— $\log_2(\text{page_size})$ bits or a few more if page coloring is used—are sufficient to index into the cache (Hennessy and Patterson 1996). As on-chip first-level caches become larger, virtually indexed caches become more attractive. However, these have their own problems. First, different processors may use the same virtual address to refer to unrelated data in different address spaces. This can be handled by flushing the whole cache on a context switch or by associating address space identifier (ASID) tags with cache blocks in addition to virtual address tags. The more serious problem for cache coherence is synonyms: distinct virtual pages, from the same or different processes, that point to the same physical page for sharing purposes. With virtually addressed caches, the same (shared) physical memory block can be fetched into two distinct blocks at different indices in the cache. This is a problem for uniprocessors, as we know, but the problem extends to cache coherence in multiprocessors as well. If one processor writes the block using one virtual address synonym and another reads it using a different synonym, then by simply putting virtual addresses on the bus and snooping them, the write to the shared physical page will not become visible to the latter processor. Putting virtual addresses on the bus also has another drawback: it requires I/O devices and memory to do virtual-to-physical translation since they deal with physical addresses. However, putting physical addresses on the bus seems to require reverse translation to look up the virtually indexed caches during a snoop, and this does not solve the synonym coherence problem by itself anyway.

The synonym problem can be avoided in software by restricting the use of synonyms. For example, synonyms may be forced to have the same page color, that is, to be the same in the bits used to index the cache if these are more than $\log_2(\text{page_size})$ bits. Alternatively, processes may be required to use the same shared virtual address when referring to the same page, as in the SPUR research project (Hill et al. 1986).

Sophisticated cache designs have also been proposed to solve the synonym coherence problem in hardware (Goodman 1987). The idea is to use virtual addresses to look up the cache on processor accesses but to put physical addresses on the bus for other caches and devices to snoop. This requires mechanisms to be provided for the following: (1) to look up the cache with the physical address if a lookup with the virtual address fails (by which time the physical address is available) or if it is detected that the block was brought into the cache by a synonym access; (2) to ensure that the same physical block is never in the same cache under two different virtual addresses at the same time; and (3) to convert a snooped physical address to an effective virtual address to look up the snooping cache. One way to accomplish these goals is for caches to maintain both virtual and physical tags (and states) for their cached blocks, indexed by virtual and physical addresses, respectively, and for the two tags for a block to point to each other (i.e., to store the corresponding physical and virtual indexes, respectively; see Figure 6.24). The cache data array itself is indexed using the virtual index (or the pointer from the physical tag entry, which is the same, in the case of a snoop). Let's see at a high level how this organization provides the needed mechanisms.

A processor looks up the cache with its virtual address, and at the same time, the virtual-to-physical translation is done by the memory management unit in case it is needed. If the lookup with the virtual address succeeds, all is well. If it fails, the translated physical address is used to look up the physical tags; if this hits, the block is found through the pointer in the physical tag. This achieves the first goal as follows. A virtual miss but physical hit detects the possibility of a synonym since the physical block may have been brought in via a different virtual address. In a direct-mapped cache, it must have been brought in by a different virtual address, so let us assume a direct-mapped cache for simplicity. The pointer contained in the physical tags now points to a different block in the cache array (the synonym virtual index) than the current virtual index. We need to make the current virtual index point to this physical data and reconcile the virtual and physical tags to remove the synonym. The physical block, which is currently at the synonym virtual index, is copied over to replace the block at the current virtual index (which is written back if necessary), so references to the current virtual index will hereafter hit right away. The block at the synonym virtual index is rendered invalid or inaccessible, so the data is now accessible only through the current virtual index (or through the physical address via the pointer in the physical tag in the case of a snoop) but not through the synonym virtual index. A subsequent access to the synonym will miss on its virtual address lookup and will have to go through the same procedure. Thus, a given physical block is valid only in one (virtually indexed) location in the cache at any given time, accomplishing the second goal. Note that if both the virtual and physical address lookups fail (a true cache miss), up to two write backs may be needed. The new block brought into the cache will be placed at the index determined from the current virtual (not physical) address, and the virtual and physical tags and states will be suitably updated to point to each other.

The address put on the bus is always a physical address, whether for a write back, a read miss, or a read exclusive or upgrade. Snooping with physical addresses from

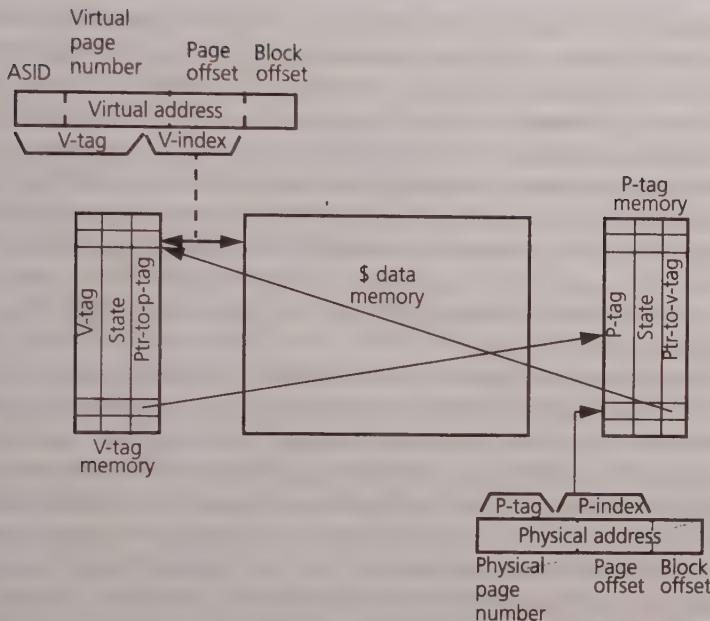


FIGURE 6.24 Organization of a dual-tagged virtually addressed cache. The v-tag memory on the left services the CPU and is indexed by virtual addresses. The p-tag memory on the right is used for bus snooping and is indexed by physical addresses. The contents of the memory block are stored based on the index of the v-tag. Corresponding p-tag and v-tag entries point to each other for handling updates to the cache.

the bus is easy. Explicit reverse translation is not required since the information needed is already there. The physical tags are looked up to check for the presence of the block, and the data is found from the pointer (corresponding virtual index) it contains. If action must be taken, the state in the virtual tag pointed to by the physical tag entry is updated as well. Further details of how such a cache system operates can be found in (Goodman 1987). This approach has also been extended to multi-level caches, where it is even more attractive: the L₁ cache is virtually tagged to speed cache access and the L₂ cache is physically tagged to facilitate snooping and avoid synonyms across processors (Wang, Baer, and Levy 1989).

6.6.3 Translation Lookaside Buffer Coherence

A processor's translation lookaside buffer (TLB) is simply a cache on the page table entries (PTEs) used for virtual-to-physical address translation. A PTE can come to reside in the TLBs of multiple processors due to either actual sharing of data or process migration. PTEs may be modified—for example, when the page is swapped out or its protection is changed—leading to direct analog of the cache coherence problem.

A variety of solutions have been used for TLB coherence. Software solutions, through the operating system, are popular since TLB coherence operations are much less frequent than cache coherence operations. The exact solutions used depend on whether PTEs are loaded into the TLB directly by hardware or under software control as well as on several other variables in how TLBs and operating systems are implemented. Hardware solutions are also used by some systems, particularly when TLB operations are not visible to software. This section provides a brief overview of four approaches to TLB coherence: virtually addressed caches, software TLB shootdown, address space identifiers (ASIDs), and hardware TLB coherence. Further details can be found in (Thompson et al. 1988; Rosenberg 1989; Teller 1990) and the papers referenced therein.

TLBs, and hence the TLB coherence problem, can be avoided entirely by using virtually addressed caches. Address translation is now needed only on cache misses, so particularly if the cache miss rate is small, we can use the page tables directly. Page table entries are brought into the regular data cache when they are accessed and are therefore kept coherent by the cache coherence mechanism. However, when a physical page is swapped out of memory or its protection changed, this is not visible to the cache coherence hardware, so the PTE must be flushed from the virtually addressed caches of all processors by the operating system. In addition, the coherence problem for virtually addressed caches must be solved. This approach was explored in the SPUR research project (Hill et al. 1986; Wood et al. 1986).

A second approach is called TLB shootdown. There are many variants that rely on different (but small) amounts of hardware support, usually including support for interprocessor interrupts and invalidation of individual TLB entries. The TLB coherence procedure is invoked by a processor, called the *initiator*, when it makes changes to PTEs that may be cached by other TLBs. Since changes to PTEs must be made by the operating system, the OS knows which PTEs are being changed and it may also know which other processors might be caching them in their TLBs (conservatively, since entries may have been replaced). The OS kernel locks the PTEs being changed (or the relevant page table sections, depending on the granularity of locking) and sends interrupts to other processors that it thinks have copies. On being interrupted, the recipients disable interrupts, look at the list of page table entries being modified (which is in shared memory), and locally invalidate those entries from their TLBs. The initiator waits for them to finish, perhaps by polling shared memory locations, and then unlocks the page table sections. A different, somewhat more complex shootdown algorithm is used in the Mach operating system (Black et al. 1989).

Some processor families, most notably the MIPS family from Silicon Graphics, use software-loaded rather than hardware-loaded TLBs, which means that the OS is involved not only in PTE modifications but also in loading a PTE into the TLB on a miss. In these cases, the coherence problem for process-private pages due to process migration can be solved using a third approach, that of ASIDs, which avoids interrupts and TLB shootdown. Every TLB entry has an ASID field associated with it to avoid flushing the entire TLB on a context switch (just as the process identifier is used in virtually addressed caches). In the case of TLBs, however, ASIDs are like tags allocated dynamically by the OS on a per-processor basis, using a free pool to which

they are returned when TLB entries are replaced; they are not associated with processes for their lifetime. One way to use the ASIDs, as was done in the IRIX 5.2 operating system, follows. The OS maintains an array for each process that tracks the ASID assigned to that process on each of the processors in the system. When a process modifies a PTE, the ASID of that process for all other processors is set to zero. This ensures that when the process is migrated to another processor, it will find its ASID to be zero there; the kernel will therefore allocate it a new one, thus preventing use of stale TLB entries. TLB coherence for pages truly shared by processes is performed using TLB shootdown.

Finally, some processor families provide hardware instructions to invalidate other processors' TLBs. In the PowerPC family (Weiss and Smith 1994), the "TLB invalidate entry" instruction (`tlbie`) broadcasts the page address on the bus so that the snooping hardware on other processors can automatically invalidate the corresponding TLB entries without interrupting the processor. The algorithm for handling changes to PTEs is simple: the operating system first makes changes to the page table and then issues a `tlbie` instruction for the changed PTEs. If the TLB is hardware loaded (as it is in the PowerPC), the OS does not know which other TLBs might be caching the PTE, so the invalidation must be broadcast to all processors. Broadcast is well suited to a bus but undesirable for the more scalable systems with distributed networks that will be discussed in subsequent chapters.

6.6.4 Snoop-Based Cache Coherence on Rings

Since the scale of bus-based cache-coherent multiprocessors is limited by the bus, it is natural to ask how snoop-based coherence may be extended to other, less limited interconnects. One straightforward extension of a bus is a ring. Instead of a single set of wires onto which all modules are attached, each module is attached to two neighboring modules. A ring is an interesting interconnection network from the perspective of coherence since, like a bus, it inherently supports broadcast-based communication. A transaction from one node to another traverses link by link down the ring, and since the average distance of the destination node is half the length of the ring, it is simple and natural to let the acknowledgment simply propagate around the rest of the ring and return to the sender. In fact, a natural way to structure the communication in hardware is to have the sender place the transaction on the ring and have other nodes inspect (snoop) it as it goes by to see if it is relevant to them. Given this broadcast and snooping infrastructure, we can provide snooping cache coherence on a ring even if memory is physically distributed among the nodes on the ring. Serialization and sequential consistency on a ring are a bit more complicated than on a bus since multiple transactions may be in progress around the ring simultaneously and the modules see the transactions at different times and potentially in different order.

The potential advantage of rings over buses, other than the use of distributed memory, is that the short, point-to-point nature of the links allows them to be driven at very high clock rates. For example, the IEEE scalable coherent interface (SCI) transport standard (Gustavson 1992; IEEE 1993) is based on 500-MHz 16-bit-wide

point-to-point links. The linear point-to-point nature also allows the links to be extensively pipelined, that is, new bits can be pumped onto the wire by the source before the previous bits have reached the destination. This latter feature allows the links to be made long without affecting their throughput. A disadvantage of rings is that the communication latency is high, typically higher than that of buses, and grows linearly with the number of processors in the ring (on average, $p/2$ hops need to be traversed before getting to the destination on a unidirectional ring and half that on a bidirectional ring).

Since rings are a broadcast media, snooping cache coherence protocols can be implemented quite naturally on them. An early ring-based snooping cache-coherent machine was the KSR1 sold by Kendall Square Research (Frank, Burkhardt, and Rothnie 1993). More recent commercial offerings, such as the Sequent NUMA-Q and Convex's Exemplar family (Convex 1993; Thekkath et al. 1997), use rings as the second-level interconnect to connect together multiprocessor nodes. (Both of these systems use a directory protocol rather than snooping on the ring interconnect, so we defer discussion of them until Chapter 8 when these protocols are introduced. In the NUMA-Q, the interconnect within a node is a bus; in the Exemplar, it is a richly connected low-latency crossbar.) The University of Toronto's Hector system (Vranesic et al. 1991; Farkas, Vranesic, and Stumm 1992) is a ring-based research prototype.

Figure 6.25 illustrates the organization of a ring-connected multiprocessor. Typically, rings are used with physically distributed memory, but the memory may still be logically shared. Each node consists of a processor, its private cache, a portion of the global main memory, and a ring interface. The ring interface consists of an input link from the ring, a set of latches organized as a FIFO buffer, and an output link to the ring. At each ring clock cycle, the contents of the latches are shifted forward, so the whole ring acts as a circular pipeline. The main function of the latches is to hold a passing transaction long enough so that the ring interface can decide whether to forward the message to the next node or not. A transaction may be taken out of the ring by storing the contents of the latch in local buffer memory and writing an empty-slot indicator into that latch instead. If a node wants to put something on the ring, it waits for an opportunity to fill a passing empty slot and fills it. Of course, it is desirable to minimize the number of latches in each interface to reduce the latency of transactions going around the ring.

The mechanism that determines when a node can insert a transaction on the ring, called the *ring access control mechanism*, is complicated by the fact that the datapath of the ring is usually much narrower than the size of the transactions being transferred on it. As a result, transactions need multiple consecutive slots on the ring. Furthermore, transactions (messages) on the ring can themselves have different sizes. For example, *request* messages are short and contain only the command and address whereas *data reply* messages contain the contents of the memory block and are longer. The final complicating factor is that arbitration for access to the ring must be done in a distributed manner since, unlike in a bus, there are no globally visible wires.

Three main options have been used for ring access control (i.e., arbitration): token-passing rings, register insertion rings, and slotted rings. In *token-passing rings*,

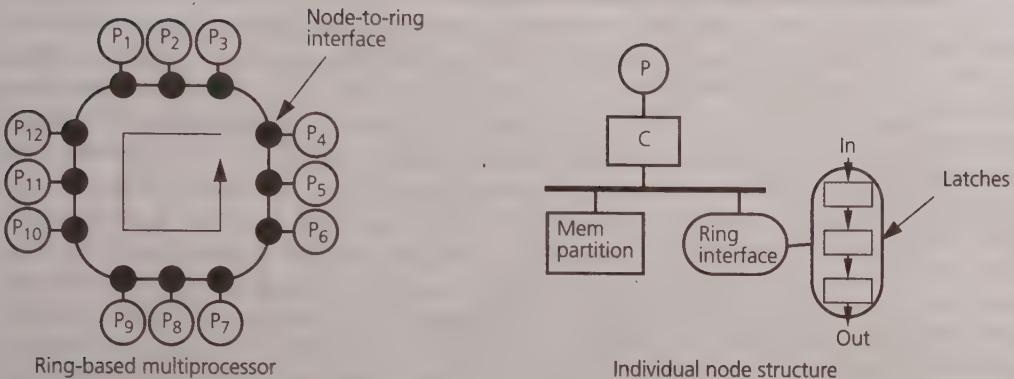


FIGURE 6.25 Organization of a single-ring multiprocessor

a special bit pattern, called a token, is passed around the ring, and only the node currently possessing the token is allowed to transmit on the ring. Arbitration is easy, but the disadvantage is that only one node may initiate a transaction at a time even though empty slots may be passing by other nodes on the ring, resulting in wasted bandwidth. Register insertion rings were chosen for the IEEE SCI standard. Here, a bypass FIFO between the input and output stages of the ring interface is used to buffer incoming transactions (with backward flow control to avoid overloading) while the local node is transmitting. When the local node finishes, the contents of the bypass FIFO are forwarded to the output link, and the local node is not allowed to transmit again until the bypass FIFO is empty. Multiple nodes may be transmitting at a time, and parts of the ring will stall when their bypass FIFOs are overloaded. Finally, in slotted rings, the ring is divided into transaction slots with labeled types (for different-sized transactions, such as requests and data replies), and these slots keep circulating around the ring. A processor ready to transmit a transaction waits until an empty slot of the required type comes by (indicated by a bit in the slot header), and then it inserts its message. A “slot” here really means a sequence of empty time slots, the length of the sequence depending on the type of message. The slotted ring can restrict the utilization of the ring bandwidth by hardwiring the mixture of available slots of different types, which may not match the actual traffic pattern for a given workload. However, for a given coherence protocol, the mix of message types is reasonably well known and little bandwidth is wasted in practice (Barroso and Dubois 1993, 1995).

While it may seem at first that broadcast and snooping waste bandwidth on a point-to-point interconnect such as a ring, in reality it is not necessarily so. A broadcast takes only twice as much bandwidth on a ring as the average random point-to-point message since the latter will, on average, traverse half the ring between two randomly chosen nodes. In addition, broadcast is needed only for request messages (read miss, write miss, upgrade requests), which are all short; data reply messages are put on the ring by the source of the data and stop at the requesting node.

Consider a cache read miss in a ring-based snooping protocol. If the main memory unit in which the block is allocated (called the *home* memory) is not on the local node, the read request is placed on the ring. If the home is local, we must determine if the block is dirty in some other node, in which case the local memory should not respond and a request should be placed on the ring. A simple solution is to place all misses on the ring as is done in the Sun Enterprise, which is a bus-based design with physically distributed memory. Alternatively, a dirty bit can be maintained for each block in home memory. This bit is turned ON if a block is cached in dirty state in some node other than the home node. If the bit is on, the request goes on the ring. The read request now circles the ring. It is snooped by all nodes, and either the home or the dirty node will respond (again, if the home were not local, the home node uses the dirty bit to decide whether or not it should respond to the request). Read-exclusive and upgrade transactions also appear on the ring, and other nodes snoop these requests and invalidate their blocks if necessary. The request and response transactions are removed from the ring when they arrive back at the requestor. The return of the request to the requesting node serves as an acknowledgment. When multiple nodes attempt to write to the same block concurrently, the winner is the one that reaches the current owner of the block first (i.e., the home node if the block is valid in main memory or the dirty node otherwise); the other nodes are implicitly or explicitly sent negative acknowledgments (NACKs), and they must retry.

From an implementation perspective, a key difficulty with snooping protocols on rings is the real-time constraint imposed: the snooper in the ring interface must examine and react to all passing messages without excessive delay or internal queuing. This can be difficult for register insertion rings since many short request messages may be adjacent to each other in the ring. With rings operating at high speeds, the requests can be too close together for the snooper to respond to in a fixed time. The problem is simplified in slotted rings, where careful choice and placement of short request messages and long data response messages (the latter are point-to-point and do not need cache lookup) can ensure that request-type messages are not too close together (Barroso and Dubois 1995). For example, slots can be grouped together in frames, and each frame can be organized to have request slots followed by response slots. Nonetheless, as with buses, bandwidth on rings is ultimately limited by snoop bandwidth at the controllers or caches rather than raw data transfer bandwidth on the interconnect.

Serialization and sequential consistency in rings are a bit trickier than on buses since the possibility exists that processors at different points on the ring will see a pair of transactions on the ring in different order (depending on where they are in the ring relative to the originators of those transactions). Using invalidation protocols simplifies this problem because writes only cause read-exclusive transactions to be placed on the ring, not the data itself, and all nodes but the home node will respond simply by invalidating their copy. The home node can determine when conflicting transactions are on the ring and take special action, but this does increase the number of transient states in the protocol substantially.

6.6.5 Scaling Data and Snoop Bandwidth in Bus-Based Systems

Several alternative methods are available to increase the bandwidth of SMP designs while preserving much of the simplicity of bus-based approaches. With split-transaction buses, we have seen that the arbitration, the address phase, and the data phase are pipelined, so each of them can go on simultaneously. Scaling data bandwidth is, in fact, the easier part; the real challenge is scaling the snoop bandwidth.

Let's consider first scaling data bandwidth. Cache blocks are large compared to the address that describes them. The most straightforward way to increase the data bandwidth is simply to make the data bus wider. We see this, for example, in the Sun Enterprise and SGI Challenge designs, both of which use 256-bit-wide data buses. In the Enterprise, a 64-byte cache block is transferred in only two cycles. The downside of this approach is cost: as the bus gets wider, it uses a larger connector, occupies more space on the board, and draws more power. It certainly pushes the limit of this style of design since an efficient, uniform pipeline demands that a snoop operation, which needs to be observed by all the caches and acknowledged, must complete in only two cycles. A more radical alternative replaces the data portion of the bus with a point-to-point crossbar, directly connecting each processor-memory module to every other one. The approach recognizes that it is only the address portion of the transaction that needs to be broadcast to all the nodes in order to determine the coherence operation and the data source (i.e., memory or cache). This approach is followed in the IBM PowerPC-based RS6000 G30 multiprocessor. A bus is used for addresses and snoop results, but a crossbar is used to move the actual data. The individual paths in the crossbar need not be extremely wide since multiple transfers can occur simultaneously.

A brute-force way to scale bandwidth in a bus-based system is simply to use multiple buses, including address buses, as mentioned in Section 6.4.7. In fact, this approach offers a fundamental contribution since it allows snoop bandwidth to be scaled as well. In order to scale the snoop bandwidth beyond one coherence result per address cycle, there must be multiple simultaneous snoop operations. Once there are multiple address buses, the data bus issues can be handled by multiple data buses, crossbars, or any other mechanism. Coherence is easy. Different portions of the address space use different buses; typically, each bus will serve specific memory banks so that a given address always uses the same bus. Multiple address buses would seem to violate the critical mechanism used to ensure sequential consistency: serialized arbitration for the address bus. Remember, however, that sequential consistency requires a logical total order, not a strict chronological order of the address events. A static ordering is logically assigned to the sequence of buses: an address operation i logically precedes j if it occurs before j in time (bus cycles) or if they happen on the same cycle but i takes place on a lower-numbered bus. This multiple bus approach is used in the Sun SparcCenter 2000, which provides two split-transaction buses, each identical to that used in the SparcStation 1000, and scales to 30 processors. The CRAY CS6400 uses four such buses and scales to 64 processors. Each cache controller snoops all of the buses separately and responds according to the cache coherence protocol. The Sun Enterprise 10000, a later machine than the Sun Enterprise 6000 discussed in this chapter, combines the use of multiple address

buses and data crossbars to scale to 64 processors. Each board consists of four 250-MHz processors, four banks of memory (up to 1 GB each), and two independent SBUS I/O buses. Sixteen of these boards are connected by a 16×16 data crossbar with paths 144 bits wide as well as four address buses associated with the four banks on each board. Collectively, this provides 12.6 GB/s of data bandwidth and a high snoop rate of 250 MHz.

6.7 CONCLUDING REMARKS

The design issues that we have explored in this chapter are fundamental and will remain important at moderate levels of parallelism. Of course, the optimal design choices may change. For example, although not currently very popular, it is possible that sharing caches at some level of the hierarchy may become quite attractive when multichip-module packaging technology becomes cheap or when multiple processors appear on a single chip.

Although it is a powerful mechanism, a shared bus interconnect clearly has bandwidth limitations as the number of processors or the processor speed increases. Architects will surely continue to find innovative ways to squeeze more data bandwidth and more snoop bandwidth out of these designs and will continue to exploit the simplicity of a broadcast-based approach, at least at small scale. However, the general solution in building scalable cache-coherent machines is to distribute memory physically among nodes and use a scalable interconnect, together with coherence protocols that do not rely on snooping. This direction is the subject of the subsequent chapters. It is likely to find its way down to the smaller scale as processors become faster relative to bus and snoop bandwidth. It is difficult to predict what the future holds for buses and the scale at which they will be used, although they are likely to have an important role for some time to come. Regardless of that evolution, the issues discussed in this and the previous chapter in the context of buses—the placement of the interconnect within the memory hierarchy, the cache coherence problem and the various coherence protocols at the state transition level, and the key correctness and implementation issues that arise when dealing with many concurrent transactions—are all largely independent of technology and are crucial to the design of all cache-coherent shared memory architectures, regardless of the interconnect used. The specific machinery used to address the correctness and implementation issues may change, but the issues, trade-offs, and basic approaches are fundamental and can be extrapolated. Moreover, these bus-based designs provide the basic building block for larger-scale design presented in the remainder of the book.

6.8 EXERCISES

- 6.1 Consider two machines M_1 and M_2 . M_1 is a four-processor shared L_1 cache machine whereas M_2 is a four-processor bus-based snooping cache machine. M_1 has a single shared 1-MB two-way set-associative cache with 64-byte blocks whereas each pro-

cessor in M_2 has a 256-KB direct-mapped cache with 64-byte blocks. M_2 uses the Illinois MESI coherence protocol. Consider the following piece of code:

```

double A[1024,1024]; /* row-major; 8-byte elems */
double C[4096];
double B[1024,1024];

for (i=0; i<1024; i+=1) /* loop-1 */
    for (j=myPID; j<1024; j+=numPEs)
    {
        B[i,j] = (A[i+i,j] + A[i-1,j] +
                   A[i,j+1] + A[i,j-1]) / 4.0;
    }
for (i=myPID; i<1024; i+=numPEs) /* loop-2 */
    for (j=0; j<1024; j+=1)
    {
        A[i,j] = (B[i+i,j] + B[i-1,j] +
                   B[i,j+1] + B[i,j-1]) / 4.0;
    }
}

```

- a. Assume that the array A starts at hexadecimal address (0x) 0, array C at 0x 300,000, and array B at 0x 308,000. All caches are initially empty. Each processor executes the preceding code, and $myPID$ varies from 0 to 3 for the four processors. Compute misses for M_1 , separately for the two loop nests. Do the same for M_2 , stating any assumptions that you make.
 - b. Briefly comment on how your answer to part (a) would change if array C were not present. State any other assumptions that you make.
 - c. What can be learned about advantages and disadvantages of shared cache architecture from this exercise?
- 6.2 Given your knowledge about the Barnes-Hut, Ocean, Raytrace, and Multiprog workloads from previous chapters and data in Section 5.4, comment on how each of the applications would do on a four-processor shared cache machine with a 4-MB cache versus a four-processor snoopy bus-based machine with 1-MB caches. It might be useful to verify your intuition using simulation. State any relevant assumptions.
- 6.3 Compared to a shared first-level cache, what are the advantages and disadvantages of having private first-level caches but a shared second-level cache? Comment on how modern microprocessors, for example, MIPS R10000 and IBM/Motorola PowerPC 620, encourage or discourage this trend. What would be the impact of packaging technology on such designs?
- 6.4 Using the terminology from Section 6.3 on cache inclusion, assume both L_1 and L_2 are two-way set associative, $n_2 > n_1$, $b_1 = b_2$, and the replacement policy is FIFO instead of LRU. Does inclusion hold naturally? What if replacement is random or based on a ring counter?

- 6.5 Give an example reference stream showing cache inclusion violation for the following situations:
- L_1 cache is 32 bytes, two-way set associative, 8-byte cache blocks, and LRU replacement. L_2 cache is 128 bytes, four-way set associative, 8-byte cache blocks, and LRU replacement.
 - L_1 cache is 32 bytes, two-way set associative, 8-byte cache blocks, and LRU replacement. L_2 cache is 128 bytes, two-way set associative, 16-byte cache blocks, and LRU replacement.
- 6.6 For the following systems, state whether or not the caches provide inclusion naturally; if not, state the problem or give an example that violates inclusion.
- L_1 : 8-KB direct-mapped primary instruction cache, 32-byte line size
 L_2 : 8-KB direct-mapped primary data cache, write through, 32-byte line size
 - L_1 : 4-MB four-way set-associative unified secondary cache, 32-byte line size
 L_2 : 16-KB direct-mapped unified primary cache, write through, 32-byte line size
 - L_1 : 4-MB four-way set-associative unified secondary cache, 64-byte line size
 L_2 : 16-KB direct-mapped unified primary cache, write through, 64-byte line size
- 6.7 Recall the discussion of the cache inclusion property in Section 6.3.
- The discussion stated that in a common case inclusion is satisfied quite naturally. The case is when the L_1 cache is direct mapped ($a_1 = 1$), L_2 can be direct mapped or set associative ($a_2 \geq 1$) with any replacement policy (e.g., LRU, FIFO, random) as long as the new block brought in is put in both L_1 and L_2 caches, the block size is the same ($b_1 = b_2$), and the number of sets in the L_1 cache is equal to or smaller than in the L_2 cache ($n_1 \leq n_2$). Show or argue why this is true.
 - The discussion claimed that the problem with multiple caches at a level being backed up by a unified cache is not solved by making the unified cache associative. Show that this is true for a simple example with direct-mapped instruction and data caches backed up by a unified, two-way set-associative cache.
- 6.8 Assume that each processor has separate instruction and data caches and that there are no instruction misses. Further assume that, when active, the processor issues a data cache request every 3 clock cycles, the miss rate is 1%, and miss latency is 30 cycles. Assume that tag reads take 1 clock cycle but modifications to the tag take 2 clock cycles.
- Quantify the performance lost to cache tag contention if a single-level data cache with only one set of cache tags is used. Assume that the bus transactions requiring snoop occur every 5 clock cycles and that 10% of these invalidate a block in the cache. Further assume that snoops are given preference over processor accesses to tags. Do back-of-the-envelope calculations. Then check the accuracy of your answer by building a queuing model or writing a simple simulator.

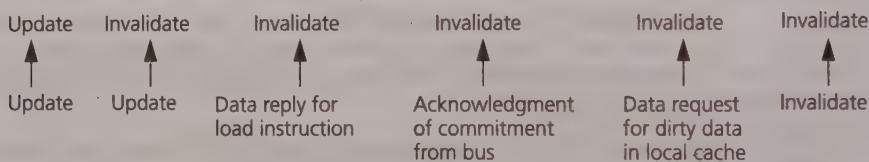
- b. What is the performance lost to tag contention if separate sets of tags for processor and snooping are used?
 - c. In general, would you give priority in accessing tags to processor references or bus snoops?
- 6.9 The designers of the SGI Challenge multiprocessor considered the following bus controller optimization to make better use of interleaved memory and bus bandwidth. If the controller finds that a request is already outstanding for a given memory bank (which can be determined from the request table), it does not issue that request until the previous one for that bank is satisfied. Discuss potential problems with this optimization and what features in the Challenge design allow this optimization.
- 6.10 a. Although the Challenge supports the MESI protocol states, it does not support the cache-to-cache transfer feature of the original Illinois MESI protocol.
 - (i) Discuss the possible reasons for this choice.
 - (ii) Extend the Challenge implementation to support cache-to-cache transfers. Describe the extra signals needed on the bus, if any, and keep in mind the issue of fairness.
- b. Although the Challenge MESI protocol has four states, the tags stored with the cache controller chip keep track of only three states (I, S, and E+M). Explain why this still works correctly. Why do you think they made this optimization?
- c. Discuss the cost, performance, implementation, and scalability trade-offs between the multiple bus architecture of the SparcCenter and the single fast wide bus architecture of the SGI Challenge, as well as any implications for program semantics and deadlock.
- 6.11 a. The main memory on the Challenge speculatively initiates fetching the data for a read request even before it is determined if it is dirty in some processor's cache. Using data from Table 5.1, estimate the fraction of useless main memory accesses. Based on the data, are you in favor of the optimization? Is this data methodologically adequate? Explain.
- b. The bus interfaces on the Challenge support request merging. Thus, if multiple processors are stalled waiting for the same memory block, then when the data appears on the bus, all of them can grab that data off the bus. This feature is particularly useful for implementing spin-lock-based synchronization primitives. For a test-and-test&|set lock, compute the minimum traffic on the bus with and without this optimization. Assume that there are four processors, each acquiring the lock once and then doing an unlock, and that initially no processor had the memory block containing the lock variable in its cache.
- 6.12 The SGI Challenge bus allows for eight outstanding transactions. How did the designers arrive at that decision? Suggest a general formula to indicate how many outstanding transactions should be supported given the parameters of the bus. Use the following parameters:

P	Number of processors
M	Number of memory banks
L	Average memory latency (cycles)
B	Cache block size (bytes)
W	Data bus width (bytes)

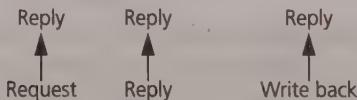
Define any other parameters you think are essential. Keep your formula simple, clearly state any assumptions, and justify your decisions.

- 6.13 Consider incoming transactions (requests and responses) from the bus into the cache hierarchy in a system with two-level caches and outgoing transactions from the cache hierarchy to the bus. To ensure sequential consistency (SC) when invalidations are acknowledged early (as soon as they appear on the bus), what ordering constraints must be maintained in each direction among the ones described in the following display, and which ones can be reordered? Answer the same question for the incoming transactions at the first-level cache. Assume that each processor has only one outstanding request at a time, but the bus is split transaction.

Orderings in the upward direction (from bus toward the caches and the processor):



Orderings in the downward direction (from the processor and caches toward the bus):



- 6.14 In the split-transaction solution discussed in Section 6.4, depending on the processor-to-cache interface, it is possible that an invalidation request comes immediately after the data response so that the block is invalidated before the processor has had a chance to actually access that cache block and satisfy its request. Why might this be a problem and how can you solve it?
- 6.15 When supporting lockup-free caches, a designer suggests that we also add more entries to the request table sitting on the bus interface of the split-transaction bus. Is this a good idea and do you expect the benefits to be large?
- 6.16 Apply the different techniques described in Section 6.4.6 to preserve SC with multiple outstanding bus transactions to Example 6.3 and convince yourself that they work. Under what conditions is one solution better than the other?

- 6.17 Assume a system bus similar to the Powerpath-2 discussed in Section 6.5. Assuming 200-MIPS/200-MFLOPS processors with 1-MB caches and 64-byte cache blocks, for each of the applications in Table 5.1, compute the bus bandwidth when using

- the Illinois MESI protocol
- the Dragon protocol
- the Illinois MESI protocol, assuming 256-byte cache blocks

For each of parts, (a), (b), and (c) compute the utilization of the address+ command bus separately from the utilization of the data bus. State all assumptions clearly.

- Complete parts (a) and (b) for a single SparcCenter XDBus, which has 64-bit-wide multiplexed address and data signals. Assume that the bus runs at 100 MHz, that transmitting address information takes 2 cycles on the bus, and that 64 bytes of data take 9 cycles on the bus.

- 6.18 One deadlock solution proposed for multilevel caches in Section 6.4.8 is to make all queues deep enough to accommodate all incoming requests and responses. Can the queues be smaller? If so, why? Discuss why it may be beneficial to have deeper queues than the size required by deadlock considerations.

- 6.19 Section 6.3 presented coherence protocols assuming two-level caches. What if there are three or more levels in the cache hierarchy? Extend the Illinois MESI protocol for the middle cache in a three-level hierarchy: list any additional states or actions needed, and present the state transition diagram.

- 6.20 Figure 6.26 shows the details of the TLB shootdown algorithm used in the Mach operating system (Black et al. 1989). For each processor, the following basic data structures are maintained: an “active” flag, indicating whether the processor is actively using any page tables; a queue of TLB flush notices, indicating the range of virtual addresses whose mappings are to be changed; and a list indicating currently active page tables (i.e., processes whose PTEs may currently be cached in the processor’s TLB). For every page table, there is a spin-lock that the processor must hold while making changes to that page table and a set of processors on which the page table is currently active. While the basic shootdown approach is simple, practical implementations require careful sequencing of steps and locking of data structures.

- Why are page table entries modified before sending interrupts or invalidate messages to other processors in TLB coherence?
- Why must the initiator of the shootdown in Figure 6.26 mask out interprocessor interrupts (IPIs) before acquiring the page table lock and clear its own active flag before acquiring the page table lock? Can you think of any deadlock conditions that exist in the figure, and if so, how would you solve them?
- A problem with the Mach algorithm is that it makes all responders busy-wait while the initiator makes changes to the page table. The reason is that it was designed for use with microprocessors that autonomously wrote back the

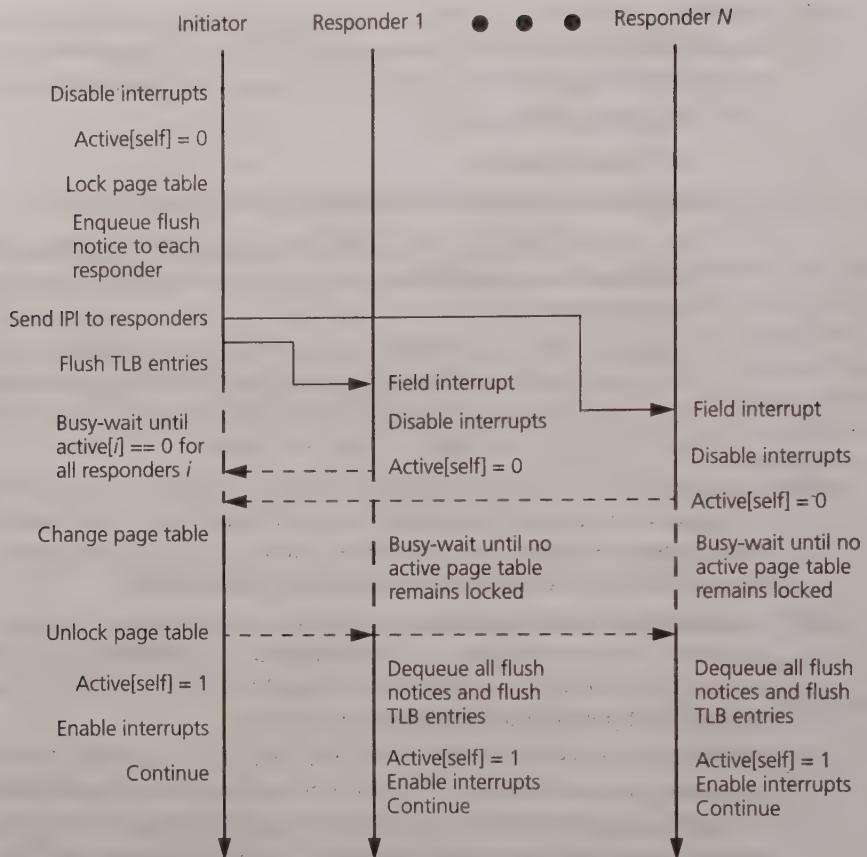


FIGURE 6.26 The Mach TLB shootdown algorithm. The initiator is the processor making changes to a page table whereas the responders are all other processors that may have entries from that page table cached in their TLBs.

entire TLB entry into the corresponding PTE whenever the dirty bit was set on a TLB replacement. Thus, for example, if other processors were allowed to use the page table while the initiator was modifying it, an autonomous write back from those processors could overwrite the new changes. How would you design the TLB hardware and/or algorithm so that responders do not have to busy-wait?

- Under what circumstances would it be better to flush the whole TLB versus selectively trying to invalidate TLB entries?

Scalable Multiprocessors

In this chapter, we begin our study of the design of machines that can be scaled in a practical manner to hundreds or even thousands of processors. Scalability has profound implications at all levels of the design. For starters, it must be physically possible and technically feasible to construct a large configuration. Adding processors clearly increases the potential computational capacity of the system, but to realize this potential, all aspects of the system must scale. In particular, the memory bandwidth must scale with the number of processors. A natural solution is to distribute the memory with the processors, as in our generic multiprocessor, so that each processor has direct access to local memory. However, the communication network connecting these nodes must provide scalable bandwidth at reasonable latency. In addition, the protocols used in transferring data within the system must scale, and so must the techniques used for synchronization. With scalable protocols on a scalable network, a very large number of transactions can take place in the system simultaneously, and we cannot rely on global information to establish ordering or to arbitrate for resources. Thus, to achieve scalable application performance, scalability must be addressed as a “vertical” problem throughout each of the layers of the system design. Let us consider a couple of familiar design points to make these scalability issues more concrete.

The small-scale shared memory machines described in Chapters 5 and 6 can be viewed as one extreme point. A shared bus typically has a maximum length of a foot or two, a fixed number of slots, and a fixed maximum bandwidth, so it is fundamentally limited in scale. The interface to the communication medium is an extension of the memory interface, with additional control lines and controller states to support the coherence protocol. A global order is established by arbitration for the bus, and a limited number of transactions can be outstanding at any time. Protection is enforced on all communication operations through the standard virtual-to-physical address translation mechanism. There is total trust between processors in the system, which are viewed as under the control of a single operating system that runs on all of the processors, with common system data structures. The communication medium is contained within the physical structure of the box and is thereby completely secure. Typically, if any processor fails, the system is rebooted. Little or no software intervention takes place between the programming model and the hardware primitives. Thus, at each level of the system design, decisions are grounded in scaling limitations at layers below and assumptions of close coupling between the

components. Scalable machines are fundamentally less closely coupled than bus-based shared memory multiprocessors, and we are forced to rethink how processors interact with other processors and with memories.

At the opposite extreme, we might consider conventional workstations on a local area or even a wide area network. Here, there is no clear limit to physical scaling and very little trust between processors in the system. The interface to the communication medium is typically a standard peripheral interface at the hardware level, with the operating system interposed between the user-level primitives and the network to enforce protection and control access. Each processing element is controlled by its own operating system, which treats the others with suspicion. No global order of operations is present, and consensus is difficult to achieve. The communication medium is external to the individual nodes and potentially insecure. Individual workstations can fail and restart independently, except perhaps where one is providing services to another. There is typically a substantial layer of software between the hardware primitives and any user-level communication operations, regardless of programming model, so communication latency tends to be quite high and communication bandwidth low. Since communication operations are handled in software, no clear limit is placed on the number of outstanding transactions or even the meaning of the transactions. At each level of the system design, it is assumed that communication with other nodes is slow and inherently unreliable. Thus, even when multiple processors are working together on a single problem, it is difficult to exploit the inherent coupling and trust within the application to obtain greater performance from the system.

Between these extremes is a spectrum of reasonable and interesting design alternatives, several of which are illustrated by current commercial large-scale parallel computers and emerging parallel computing clusters. Many of the *massively parallel processors* (MPPs) employ sophisticated packaging and a fast dedicated proprietary network so that a very large number of processors can be located in a confined space with high-bandwidth and low-latency communication. Other scalable machines use essentially conventional computers as nodes with more or less standard interfaces to fast networks. In either case, there is a great deal of physical security and the option of either a high degree of trust or of substantial autonomy.

The generic multiprocessor of Chapter 1 provides a useful framework for understanding scalable designs: the machine is organized as essentially complete computational nodes, each with a memory subsystem and one or more processors, connected by a scalable network. The nature of the *node-to-network interface* is one of the most critical issues in scalable system design. It allows a wide scope of possibilities, differing in how tightly coupled the processor and memory subsystems are to the network and in the processing power within the network interface itself. These issues affect the degree of trust between the nodes and the performance characteristics of the communication primitives, which in turn determine the efficiency with which various programming models can be realized on the machine.

Our goal in this chapter is to understand the design trade-offs across the spectrum of communication architectures for scalable machines. We want to understand, for example, how the decision to pursue a more specialized or a more commodity-

oriented approach impacts the capabilities of the node-to-network interface, the ability to support various programming models efficiently, and the limits on scale. We begin in Section 7.1 with a general discussion of scalability, examining the requirements it places on system design in terms of bandwidth, latency, cost, and physical construction. This discussion provides a nuts-and-bolts introduction to a range of recent large-scale machines but also allows us to develop an abstract view of the scalable network on the “other side” of the node-to-network interface. We return to an in-depth study of the design of scalable networks later in Chapter 10, after having fully developed the requirements on the network in this and the following two chapters.

We focus in Section 7.2 on the question of how programming models are realized in terms of the communication primitives provided on large-scale parallel machines. The key concept is that of a *network transaction*, which is the analog for scalable networks of the bus transaction studied in the previous chapter. Working with a fairly abstract concept of a network transaction, we look at how shared address space and message-passing models are realized through protocols built out of network transactions.

The remainder of the chapter examines a series of important design points with increasing levels of direct hardware interpretation of the information in the network transaction. In a general sense, the interpretation of the network transaction is akin to the interpretation of an instruction set. Very modest interpretation suffices in principle, but more extensive interpretation is important for performance in practice. Section 7.3 investigates the case where there is essentially no interpretation of the message transaction; it is viewed as a sequence of bits and transferred blindly into memory via a physical direct memory access (DMA) operation under operating system control. This is an important design point, as it represents many early MPPs and most current local area network (LAN) interfaces.

Section 7.4 considers more aggressive designs where messages can be sent from user level to user level without operating system intervention. At the very least, this requires that the network transaction carry a user/system identifier, which is generated by the source communication assist and interpreted by the destination. This small change gives rise to the concept of a user virtual network, which, like virtual memory, must present a protection model and offer a framework for sharing the underlying physical resources. A particularly critical issue is user-level message reception since message arrival is inherently asynchronous to the user thread for which it is destined.

Section 7.5 focuses on designs that provide a global virtual address space. This requires substantial interpretation at the destination since it needs to perform the virtual-to-physical translation, carry out the desired data transfer, and provide notification. Typically, these designs use a dedicated message or communication processor (CP) so that extensive interpretation of the network transaction can be performed without the specifics of the interpretation being bound at machine design time. Section 7.6 considers more specialized support for a global physical address space. In this case, the communication assist is closely integrated with the memory subsystem and, typically, it is a specialized device supporting a limited set of network

transactions. The support of a global physical address space brings us full circle to designs that are close in spirit to the small-scale shared memory machines studied in the previous chapter. However, automatic replication of shared data through coherent caches in a scalable fashion is considerably more involved than in the bus-based setting, and we devote Chapters 8 and 9 to that topic.

7.1 SCALABILITY

What does it mean for a design to “scale”? Almost all computers allow the capability of the system to be increased in some form, for example by adding memory, I/O cards, disks, or upgraded processor(s), but the increase typically has hard limits. A *scalable system* attempts to avoid inherent design limits on the extent to which resources can be added to the system. In practice, a system can be quite scalable even if it is not possible to assemble an arbitrarily large configuration because, at any point in time, crude limits are imposed by economics. If a “sufficiently large” configuration can be built, the scalability question has really to do with the incremental cost of increasing the capacity of the system and the resultant increase in performance delivered on applications. In practice, no design scales perfectly, so our goal is to understand how to design systems that scale up to a large number of processors effectively. In particular, we look at four aspects of scalability in a more or less top-down order. First, how does the bandwidth or throughput of the system increase with additional processors? Ideally, throughput should be proportional to the number of processors. Second, how does the latency or time per operation increase? Ideally, this should be constant. Third, how does the cost of the system increase, and finally, how do we actually package the systems and put them together?

It is easy to see that the bus-based multiprocessors of Chapter 6 fail to scale well in all four aspects, and the reasons are quite interrelated. In those designs, several processors and memory modules were connected via a single set of wires—the bus. When one module is driving a wire, no other module can drive it. Thus, the bandwidth of a bus does not increase as more processors are added to it; at some point, it will saturate. Even accepting this defect, we could consider constructing machines with many processors on a bus, perhaps under the belief that the bandwidth requirements per processor might decrease with added processors. Unfortunately, the clock period of the bus is determined by the time to drive a value onto the wires and have it sampled by every module on the bus, which increases with the number of modules on the bus and with wire length. Thus, a bigger bus would have longer latency and less aggregate bandwidth. In fact, the signal quality on the wire degrades with length and number of connectors, so for any bus technology there is a hard limit on the number of slots into which modules can be plugged and on the maximum wire length. Accepting this limit, it would seem that the bus-based designs have good cost scaling since processors and memory can be added at the cost of the new modules. Unfortunately, this simple analysis overlooks that even the minimum configuration is burdened by a large fixed cost for the infrastructure needed to support the maximum configuration; the bus, the cabinet, the power supplies, and other compo-

nents must be sized for the full configuration. At the very least, a scalable design must overcome these limitations. The aggregate bandwidth must increase with the number of processors, the time to perform an operation should not increase substantially with the size of the machine, and a large configuration must be practical and cost-effective to build. It is also valuable if the design scales down well, so small configurations are cost-effective.

7.1.1 Bandwidth Scaling

Fundamentally, if a large number of processors are to exchange information simultaneously with many other processors or memories, a large number of independent wires must connect them. Thus, scalable machines must be organized in the manner illustrated abstractly by Figure 7.1; a large number of processor modules and memory modules connected together by *independent wires* (or links) through a large number of switches. We use the term *switch* in a very general sense to mean a device connecting a limited number of inputs to a limited number of outputs. Internally, such a switch may be realized by a bus, a crossbar, or even an ad hoc collection of multiplexers. We call the number of outputs (or inputs) the *degree* of the switch. With a bus, the physical and electrical constraints discussed previously determine its degree. Only one of the inputs can transfer information to the outputs at a time. A crossbar allows every input to be connected to a distinct output, but the degree is constrained by the cost and complexity of the internal array of cross-points. The cost of multiplexers increases rapidly with the number of ports, and latency increases as well. Thus, switches are limited in scale but may be interconnected to form large configurations, that is, *networks*. In addition to the physical interconnect between inputs and outputs, there must also be some form of controller to determine which inputs are to be connected to which outputs at each instant in time. In essence, a scalable network is like a roadway system with wires for streets, switches for intersections, and a simple way of determining which cars proceed at each intersection. If done right, a large number of vehicles may make progress to their destinations simultaneously and get there quickly.

By our definition, a basic bus-based SMP contains a single switch connecting the processors and the memories, and a simple hierarchy of bus-based switches connects these components to the peripherals. The control path in a bus is rather specialized in that the address associated with a transaction at one of the inputs is broadcast to all of the outputs and the acknowledgment determines which output is to participate. A *network switch* is a more general-purpose device, in which the information presented at the input is enough for the switch controller to determine the proper output without consulting all the nodes. Pairs of modules are connected by *routes* through network switches.

The most common structure for scalable machines is illustrated by our generic architecture of Figure 7.2, in which one or more processors are packaged together with one or more memory modules and a communication assist as an easily replicated unit, which we will call a *node*. The “intranode” switch is typically a

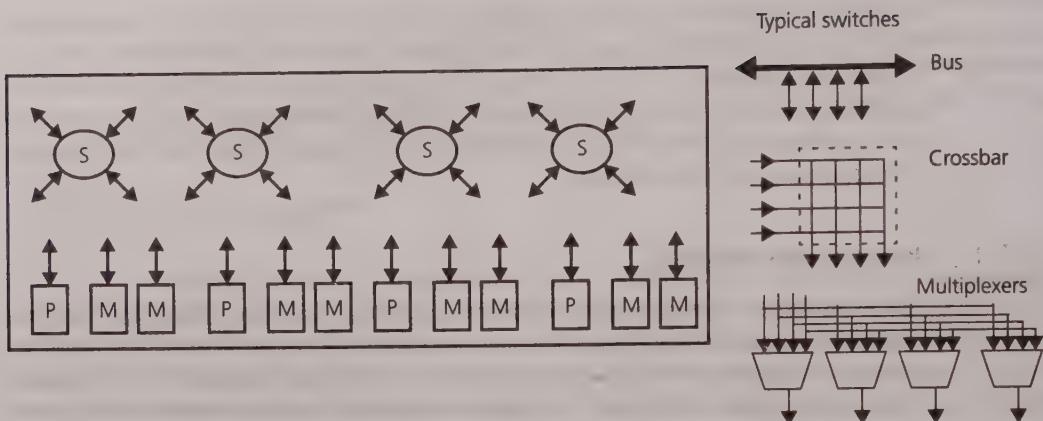


FIGURE 7.1 Abstract view of a scalable machine. A large number of processor (P) and memory (M) modules are connected by independent wires (or links) through a large number of switch modules (S), each with some limited number of degree. An individual switch may be formed by a bus, a crossbar, multiplexers, or some other controlled connection between inputs and outputs.

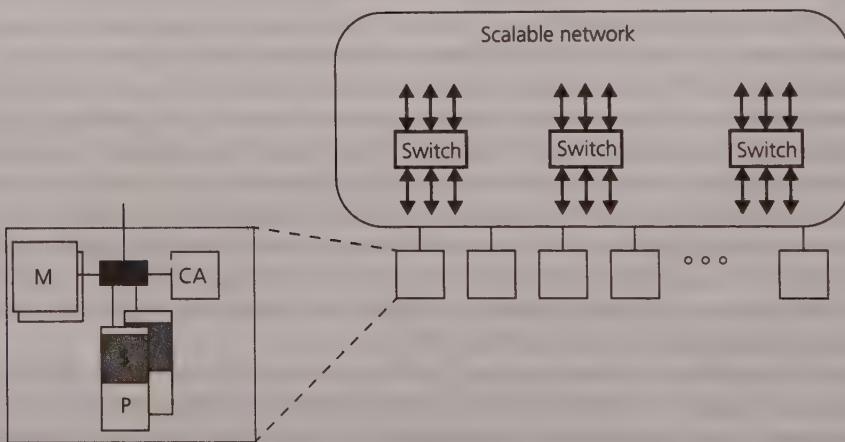


FIGURE 7.2 Generic distributed-memory multiprocessor organization. A collection of essentially complete computers, including processor and memory, that communicate through a general-purpose, high-performance, scalable interconnection network. Typically, each node contains a controller that assists in initiating and receiving communication operations.

high-performance bus. Alternatively, systems may be constructed in a dancehall configuration, in which processing nodes are separated from memory nodes by the network, as in Figure 7.3. In either case, there is a vast variety of potential switch designs, interconnection network topologies, and routing algorithms, which we will study in Chapter 10. The key property of a scalable network is that it provide a large

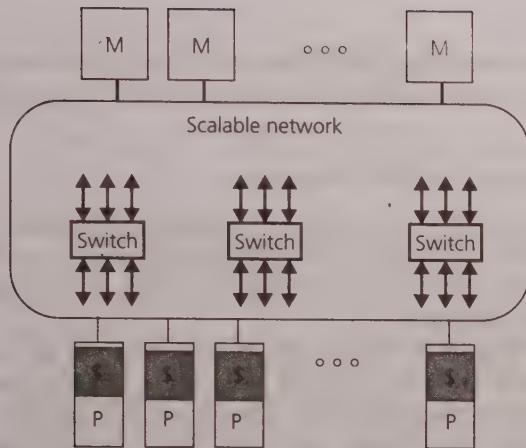


FIGURE 7.3 Dancehall multiprocessor organization. Processors access memory modules across a scalable interconnection network. Even if processes are totally independent with no communication or sharing, the bandwidth requirement on the network increases linearly with the number of processors.

number of independent communication paths between nodes such that the bandwidth increases as nodes are added. Ideally, the latency in transmitting data from one node to another should not increase with the number of nodes, nor should the cost per node, but, as we will discuss, some increase in latency and cost is unavoidable.

If the memory modules are on the opposite side of the interconnect, as in Figure 7.3, the network bandwidth requirement scales linearly with the number of processors, even when no communication occurs between processes. Providing adequate bandwidth scaling may not be enough for the computational performance to scale perfectly since the access latency increases with the number of processors. By distributing the memories across the processors, all processes can access local memory with fixed latency, independent of the number of processors; thus, the computational performance of the system can scale perfectly, at least in this simple case. The network needs to meet the demands associated with actual communication and sharing of information. How computational performance scales in the more interesting case where processes do communicate depends on how the network itself scales, how efficient the communication architecture is, and how the program communicates.

To achieve scalable bandwidth, we must abandon several key assumptions employed in bus-based designs; namely, that there are a limited number of concurrent transactions and that these are globally ordered via central arbitration and globally visible. Instead, it must be possible to have a very large number of concurrent transactions using different wires. They are initiated independently and without global arbitration. The effects of a transaction (such as changes of state) are directly visible only by the nodes involved in the transaction. (The effects may eventually become visible to other nodes as they are propagated by additional transactions.) Although it is possible to broadcast information to all the nodes, *broadcast bandwidth* (i.e., the rate at which broadcasts can be performed) does not increase with the number of nodes. Thus, in a large system broadcasts can be used only infrequently.

7.1.2 Latency Scaling

We may extend our abstract model of scalable networks to capture the primary aspects of communication latency. In general, the time to transfer n bytes between two nodes is given by

$$T(n) = \text{Overhead} + \text{Channel Time} + \text{Routing Delay} \quad (7.1)$$

where Overhead is the processing time in initiating or completing the transfer, Channel Time is n/B (where B is the bandwidth of the “thinnest channel”), and Routing Delay is a function $f(H,n)$ of the number of routing steps, or hops, in the transfer and possibly the number of bytes transferred.

The processing overhead may be fixed or it may increase with n if the processor must copy data for the transfer. For most networks used in parallel machines, there is a fixed delay per router hop, independent of the transfer size, because the message cuts through several switches.¹ In contrast, traditional data communication networks “store and forward” the data at each stage, incurring a delay per hop proportional to the transfer size.² Store-and-forward routing is impractical in large-scale parallel machines. Since network switches have a fixed degree, the average routing distance between nodes must increase as the number of nodes increases. Thus, communication latency increases with scale. However, the increase may be small compared to the overhead and transfer time if the switches are fast and the interconnection topology is reasonable (see Example 7.1).

EXAMPLE 7.1 Many classic networks are constructed out of fixed-degree switches in a configuration, or *topology*, such that for n nodes the distance from any network input to any network output is $\log_2 n$ and the total number of switches is $\alpha n \log n$ for some small constant α . Assuming the overhead is 1 μs per message, the link bandwidth is 64 MB/s and the router delay is 200 ns per hop. How much does the time for a 128-byte transfer increase as the machine is scaled from 64 to 1,024 nodes?

Answer At 64 nodes, six hops are required so

$$T_{64}(128) = 1.0 \mu\text{s} + \frac{128 \text{ B}}{64 \text{ B}/\mu\text{s}} + 6 \times 0.200 \mu\text{s} = 4.2 \mu\text{s}$$

This increases to 5 μs on a 1,024-node configuration. Thus, the latency increases by less than 20% with a 16-fold increase in machine size. Even with this small transfer size, a store-and-forward delay would add 2 μs (the time to buffer 128 bytes) to the routing delay per hop. Thus, the latency would be

-
1. The message may be viewed as a train, where the locomotive makes a choice at each switch in the track and all the cars behind follow, even though some may still be crossing a previous switch when the locomotive makes a new turn.
 2. The *store-and-forward* approach is like what the train does when it reaches the station. The entire train must come to a stop at a station before it can resume travel, presumably after exchanging goods or passengers. Thus, the time a given car spends in the station is linear in the length of the train.

$$T_{sf,64}(128) = 1.0 \mu s + \left(\frac{128 B}{64 B/\mu s} + 0.200 \mu s \right) \times 6 = 14.2 \mu s \text{ at 64 nodes and}$$

$$T_{sf,1,024}(128) = 1.0 \mu s + \left(\frac{128 B}{64 B/\mu s} + 0.200 \mu s \right) \times 10 = 23 \mu s \text{ at 1,024 nodes. ■}$$

In practice, an important connection exists between bandwidth and latency that is overlooked in Example 7.1. If two transfers involve the same node or utilize the same wires within the network, one may delay the other due to contention for the shared resource. As more of the available bandwidth is utilized, the probability of contention increases and the expected latency increases. In addition, queues may build up within the network, further increasing the expected latency. This basic saturation phenomenon occurs with any shared resource. One of the goals in designing a good network is to ensure that these load-related delays are not too large on the communication patterns that commonly occur in practice. However, if a large number of processors transfer data to the same destination node at once, there is no escaping contention. The problem must be resolved at a higher level by balancing the communication load using the techniques described in Chapter 3.

7.1.3 Cost Scaling

For large machines, scaling the cost of the system is quite important. In general, we may view this as a fixed cost for the system infrastructure plus an incremental cost of adding processors and memory to the system:

$$\text{Cost}(p,m) = \text{Fixed Cost} + \text{Incremental Cost } (p,m)$$

The fixed and incremental costs are both important. For example, the fixed cost in bus-based machines typically covers the cabinet, the power supply, and the bus supporting a full configuration. This puts small configurations at a disadvantage relative to the uniprocessor competition but encourages expansion as the incremental cost of adding processors is constant and often much less than the cost of a stand-alone processor. (Interestingly, for most commercial bus-based machines the typical configuration is about one-half of the maximum configuration. This is sufficiently large to amortize the fixed cost of the infrastructure yet leaves “headroom” on the bus and allows for expansion. Vendors who supply large SMPs, say, 20 or more processors, usually offer a smaller model providing a low-end sweet spot.) We have highlighted the incremental cost of memory in our cost equation because memory often accounts for a sizable fraction of the total system cost, and a parallel machine need not necessarily have p times the memory of a uniprocessor.

Experience has shown that scalable machines must support a wide range of configurations, not just large and extra-large sizes. Thus, the “pay-up-front” model of bus-based machines is impractical for scalable machines. Instead, the infrastructure must be modular so that, as more processors are added, more power supplies and

more cabinets are added, as well as more network. For networks with good bandwidth scaling and good latency scaling, the cost of the network grows more than linearly with the number of nodes, but in practice the growth rate is not too burdensome, as illustrated in Example 7.2.

EXAMPLE 7.2 In many networks the number of network switches scales as $n \log n$ for n nodes. Assuming that at 64 nodes the cost of the system is equally balanced between processors, memory, and network, what fraction of the cost of a 1,024-node system is devoted to the network (assuming the same amount of memory per processor)?

Answer We may normalize the cost of the system to the per-processor cost of the 64-node system. The large configuration will have $10/6$ as many routers per processor as the small system. Thus, assuming the cost of the network is proportional to the number of routers, the normalized cost per processor of the 1,024-node system is $1 \text{ processor} \times 0.33 + 1 \text{ memory} \times 0.33 + 10/6 \text{ routers} \times 0.33 = 1.22$. As the system is scaled up by 16-fold, the share of cost in the network increases from 33% to 45%. (In practice, additional factors such as increased wire length may cause network cost to increase somewhat faster than the number of switches.) ■

Network designs differ in how the bandwidth increases with the number of ports, in how the cost increases, and in how the delay through the network increases, but invariably, all three do increase. There are many subtle trade-offs among these three factors, but for the most part, the greater the increase in bandwidth, the smaller the increase in latency and the greater the increase in cost. Good design involves trade-offs and compromises. Ultimately, these will be rooted in the application requirements of the target workload.

Finally, when looking at issues of cost, it is natural to ask whether a large-scale parallel machine can be cost-effective or if it is only a means of achieving greater performance. The standard definition of efficiency ($\text{Efficiency}(p) = \text{Speedup}(p)/p$) reflects the view that a parallel machine is effective only if all of its processors are effectively utilized all the time. This processor-centric view neglects to recognize that much of the cost of the system is elsewhere, especially in the memory system (Wood and Hill 1995). If we define the cost scaling of a system, costup , in a manner analogous to speedup ($\text{Costup}(p) = \text{cost}(p)/\text{cost}(1)$), then we can see that parallel computing is cost-effective, that is, it has a smaller cost-performance ratio, whenever $\text{Speedup}(p) > \text{Costup}(p)$. Thus, in a real application scenario, we need to consider the entire cost of the system required to run the problem of interest.

7.1.4 Physical Scaling

While it is generally agreed that modular construction is essential for large-scale machines, little consensus emerges on the specific requirements of physical scale, such as how compact the nodes need to be, how long the wires can be, the clocking strategy, and so on. In some commercial machines, the individual nodes occupy scarcely more than the microprocessor footprint whereas, in others, a node is a large fraction of a board or a complete workstation chassis. In some machines, no wire is

longer than a few inches; in others, the wires are several feet long. In some machines, the links are 1 bit wide; in others, 8 or 16. Generally speaking, links tend to get slower with length, and each specific link technology has an upper limit on length due to power requirements and signal-to-noise ratio. Technologies that support very long distances, such as optical fiber, tend to have a much larger fixed cost per link for the transceivers and connectors. Thus, there are scaling advantages to a dense packing of nodes in physical space. On the other hand, a looser packing tends to reduce the engineering effort by allowing greater use of commodity components, which also reduces the time lag from availability of new microprocessor and memory technology to its availability in a large parallel machine. Thus, loose packing can have better technology scaling. The complex set of trade-offs between physical packaging strategies has given rise to a broad spectrum of designs, so it is best to look at some concrete examples. These examples also help make the other aspects of scaling more concrete.

Chip-Level Integration

A modest number of designs have integrated the communications architecture directly into the processor chip. The nCUBE/2 is a good representative of this densely packed node approach, even though the machine itself is rather old. The highly integrated node approach is also used in the MIT J-machine (Dally, Keen, and Noakes 1993) and a number of other research machines and embedded systems. The design style may gain wider popularity as chip density continues to increase.

In the nCUBE, each node had the processor, memory controller, network interface, and network router integrated in a single chip. The node chip connected directly to DRAM chips and 14 bidirectional network links on a small card occupying a few square inches, shown in actual size in Figure 7.4. The network links formed bit-serial channels connecting directly to other nodes³ and one bidirectional channel to the I/O system. Each of the 28 wires had a dedicated DMA device on the node chip. The nodes were socketed 64 to a board, and the boards plugged into a passive wiring backplane, forming direct node-to-node wire pairs between each processor chip and $\log n$ other processors. The I/O links, one per processor, were brought outside the main rack to I/O nodes containing a node chip (connecting to eight processors) and an I/O device. The maximum configuration was 8,096 processors, and machines with 2,048 nodes were built in 1991. The system ran on a single 40-MHz clock in all configurations. Since some of the wires reached across the full width of the machine, dense packing was critical to limiting the maximum length.

The nCUBE/2 should be understood as a design at a point in time. The node chip contained roughly 500,000 transistors, which was large for its time. The processor was something of a reduced VAX running at 20 MHz with 64-bit integer operation

3. The nCUBE nodes were connected in a hypercube configuration. A hypercube, or n -cube, is a graph generalizing the cube shown in Figure 7.4, where each node connects directly to $\log n$ other nodes in an n -node configuration. Thus, 13 links could support a design of up to 8,096 nodes.

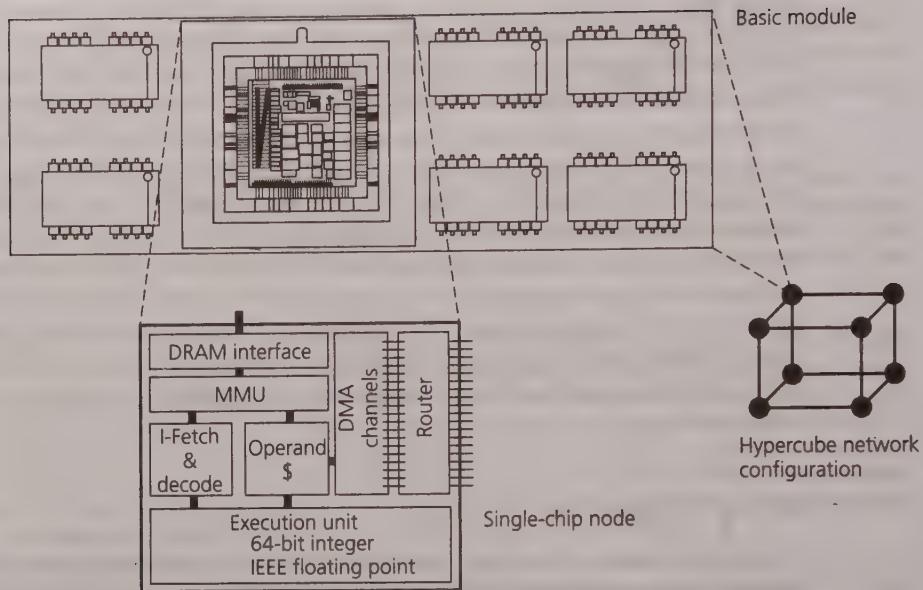


FIGURE 7.4 nCUBE/2 machine organization. The design is based on a compact module comprising a single-chip node, containing the processor, memory interface, network switch, and network interface, directly connected to DRAM chips and to other nodes.

and 64-bit IEEE floating point, with a peak of 7.5 MIPS and 2.4 MFLOPS double precision. The communication support essentially occupied the silicon area that would have been devoted to cache in a uniprocessor design of the same generation; the instruction cache was only 128 bytes, and the data cache held eight 64-bit operands. Network links were 1 bit wide, and the DMA channels operated at 2.22 MB/s each.

In terms of latency scaling, the nCUBE/2 is a little more complicated than our cut-through model. A message may be an arbitrary number of 32-bit words to an arbitrary destination, with the first word being the address of the destination node. The message is routed to the destination as a sequence of 36-bit chunks—32 data plus 4 bits of parity—with a routing delay of 44 cycles ($2.2 \mu\text{s}$) per hop and a transfer time of 36 cycles per word. The maximum number of hops with n nodes is $\log n$ and the average distance is half that amount. In contrast, the J-machine organized the nodes in a three-dimensional grid (actually a 3D torus) so that each node was connected to six neighbors with very short wires. Individual links were relatively wide (8 bits). The maximum number of hops in this organization is roughly $\frac{3}{2}\sqrt[3]{n}$, and the average is half this many.

Board-Level Integration

The most common hardware design strategy for large-scale machines obtains a moderately dense packing by using standard microprocessor components and integrating them at the board level. Representatives of this approach include the CalTech hypercube machines (Seitz 1985) and the Intel iPSC and iPSC/2, which essentially placed the core of an early personal computer on each node. The Thinking Machines CM-5 replicated the core of a Sun SparcStation 1 workstation, the CM-500 replicated the SparcStation 10, and the CRAY T3D and T3E essentially replicated the core of a DEC Alpha workstation. Most recent machines place a few processors on a board, and in some cases each board contains a bus-based multiprocessor. For example, the Intel ASCI Red machine has more than 4,000 Pentium Pro two-way multiprocessors.

The Thinking Machines CM-5 is a good representative of the board-level approach, circa 1993. The basic hardware organization of the CM-5 is shown in Figure 7.5. The node comprised essentially the components of a contemporary workstation, in this case a 33-MHz Sparc microprocessor, its external floating-point unit, and a cache controller connected to an MBUS-based memory system.⁴ The network interface was an additional ASIC on the Sparc MBUS. Each node connected to two data networks, a control network, and a diagnostic network. The network was structured as a 4-ary tree with the processing nodes at the leaves. A board contained four nodes and a network switch that connected these nodes together at the first level of the tree. In order to provide scalable bandwidth, the CM-5 used a kind of multirooted tree, called a fat-tree (discussed in Chapter 10), which has the same number of network switches at each level. Each board contained one of the four network switches forming the second level of the network for 16 nodes. Higher levels of the network tree resided on additional network boards, which were cabled together. Several boards fit in a rack, but for configurations on the scale of a thousand nodes several racks were cabled together using large wiring bundles. In addition, racks of routers were used to complete the interconnection network. The links in the network were 4 bits wide, clocked at 40 MHz, delivering a peak bandwidth of about 12 MB/s. The routing delay was 10 cycles per hop, with at most $2 \log_4 n$ hops.

The CM-5 network provided a kind of scalable backplane, supporting multiple independent user partitions, as well as a collection of I/O devices. Although memory was distributed over the processors, a dancehall approach was adopted for I/O. A collection of dedicated I/O nodes were accessed uniformly across the network from the processing nodes. Other machines employing similar board-level integration, such as the Intel Paragon and the CRAY T3D and T3E, connect the boards in a grid-like fashion to keep the wires short and use wider, faster links (Dally 1990b). I/O nodes are typically on the faces of the grid or occupy internal planes of the cube.

4. The CM-5 used custom memory controllers that contained a dedicated, memory-mapped vector accelerator. This aspect of the design grew out of the CM-2 SIMD heritage of the machine and is incidental to the physical machine scaling.

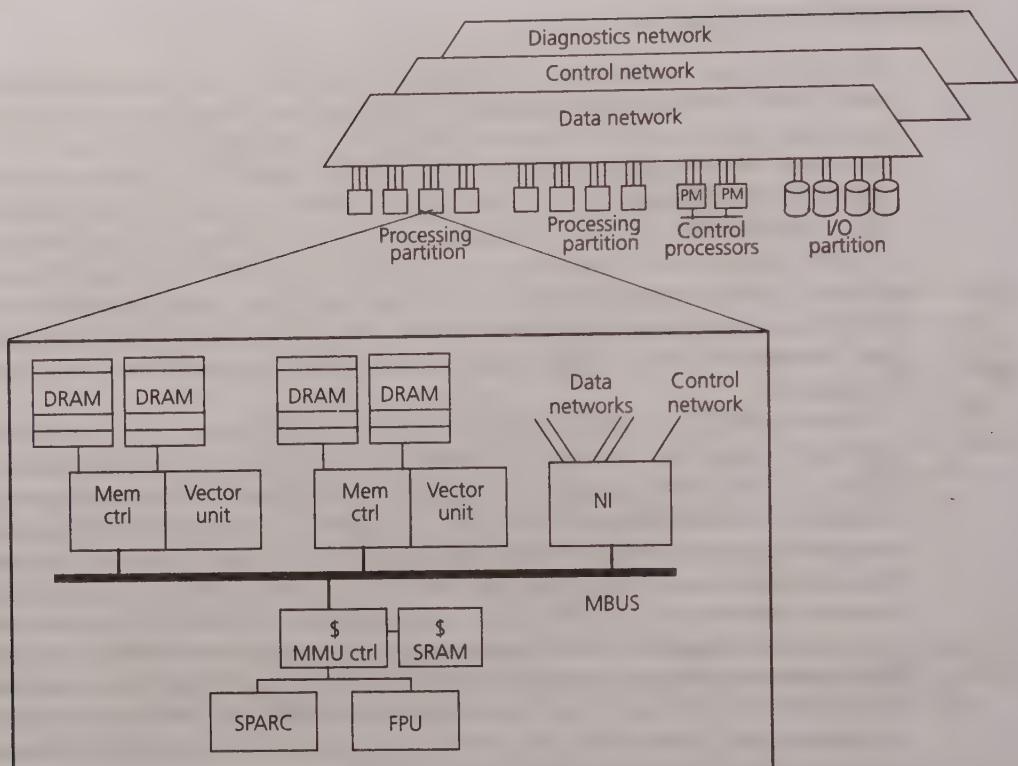


FIGURE 7.5 CM-5 machine organization. Each node is a repackaged SparcStation chip set (processor, FPU, MMU, cache, memory controller, and DRAM) with a network interface chip on the MBUS. The networks (data, control, and diagnostics) form a “scalable backplane” connecting computational partitions with I/O nodes.

System-Level Integration

Some recent large-scale machines employ less dense packaging in order to reduce the engineering time to utilize new microprocessor and operating system technology. The IBM Scalable Parallel system design (SP-1 and SP-2) is a good representative; it puts several almost complete RS6000 workstations into a rack. Since a complete, standard system is used for the node, the communication assist and network interface are part of a card that plugs into the system. For the IBM SPs, this is a Micro-Channel adapter card connecting to a switch in the base of the rack. The individual network links operate at 40 MB/s. Eight to sixteen nodes fill a rack, so large configurations are built by cabling together a number of racks, including additional switch racks. With a complete system at every node, there is the option of distributed disks and other I/O devices over the entire machine. In the SP systems, the disks on compute nodes are typically used only for swap space and temporary storage. Most of the I/O devices are concentrated on dedicated I/O nodes. This style of design allows

for a degree of heterogeneity among the nodes since all that is required is that the network interface card can be inserted. For example, the SP systems support several models of workstations as nodes, including SMP nodes.

The high-speed networking technology developed for large-scale parallel machines has migrated into a number of widely used local area networks (LANs). For example, ATM (asynchronous transfer mode) is a scalable, switch-based network supporting 155-Mb/s and 622-Mb/s links. FDDI switches connecting many 100-Mb/s rings are available, along with switch-based FiberChannels and HPPI. Many vendors support switch-based 100-Mb/s Ethernet, and switch-based gigabit Ethernet is emerging. In addition, a number of higher-bandwidth, lower-latency system area networks (SANs), which operate over shorter physical distances, have been commercialized, including Myrinet, SCI, and ServerNet. These networks are very similar to traditional large-scale multiprocessor networks and allow conventional computer systems to be integrated into a “cluster” with a scalable, low-latency interconnect. In many cases, the individual machines are small scale multiprocessors. Because the nodes are complete, independent systems, this approach is widely used to provide high-availability services, such as databases, where, if one node fails, its job “fails over” to the others.

7.1.5 Scaling in a Generic Parallel Architecture

The engineering challenges of large-scale parallel machines are well understood today, with several companies routinely producing systems of several hundred to a thousand high-performance microprocessors. The tension between tighter physical integration and the engineering time to incorporate new microprocessor technology gives rise to a wide spectrum of packaging solutions, all of which have the conceptual organization of our generic parallel architecture shown in Figure 7.2. Scalable interconnection network design is an important and interesting subarea of parallel architecture that has advanced dramatically over recent years, as we will see in Chapter 10. Several good networks have been produced commercially, which offer high per-link bandwidth and reasonably low latency that increases slowly with the size of the system. Furthermore, these networks provide scalable aggregate bandwidth, allowing a very large number of transfers to occur simultaneously. They are also robust in that the hardware error rates are very low, in some cases as low as modern buses. Each of these designs has some natural scaling limit as a result of bandwidth, latency, and cost factors, but from an engineering viewpoint it is practical today to build machines on a scale that is limited primarily by financial concerns.

In practice, the target maximum scale is quite important in assessing design trade-offs. It determines the level of design and engineering effort warranted by each aspect of the system. For example, the engineering required to achieve the high packaging density and degree of modularity needed to construct very large systems may not be cost-effective at the moderate scale where less sophisticated solutions suffice. A practical design seeks a balance between computational performance, communication performance, and cost at the time the machine is produced. For example, better communication performance or better physical density might be

achieved by integrating the network more closely with the processor. However, this may increase cost or compromise performance, either by increasing the latency to memory or increasing design time, and thus might not be the most effective choice. With processor performance improving rapidly over time, a more rudimentary design starting later on the technology curve might be produced at the same time with higher computational performance but perhaps lower communication performance. As with all aspects of design, it is a question of balance.

What the entire spectrum of large-scale parallel machines have in common is that a very large number of transfers can be ongoing simultaneously, there is essentially no instantaneous global information or global arbitration, and the bulk of the communication time is attributable to the node-to-network interface. These are the issues that dominate our thinking from an architectural viewpoint. It is not enough that the hardware capability scales; the entire system solution must scale, including the protocols used to realize programming models and the capabilities provided by the operating system, such as process scheduling, storage management, and I/O. Serialization due to contention for locks and shared resources within applications or even the operating system may limit the useful scaling of the system even if the hardware scales well in isolation. Given that we have met the engineering requirements to physically scale the system to the size of interest in a cost-effective manner, we must also ensure that the communication and synchronization operations required to support the target programming models scale and have a sufficiently small fixed cost to be effective.

7.2

REALIZING PROGRAMMING MODELS

In this section, we examine what is required to implement programming models on large distributed-memory machines. Historically, these machines have been most strongly associated with message-passing programming models, but shared address space programming models have become increasingly important and well represented. Chapter 1 introduced the concept of a communication abstraction, which defined the set of communication primitives provided to the user. These could be realized directly in the hardware, via system software, or through some combination of the two, as illustrated by the now familiar Figure 7.6. This perspective focuses our attention on the aspects of the node architecture that support communication. In small-scale shared memory machines, the communication abstraction is supported directly in hardware as an extension of the memory interface. The load and store operations in the coherent shared memory abstraction are implemented by a sequence of primitive bus transactions according to a specific protocol defined by a collection of state machines.

In large-scale parallel machines, the programming model is realized in a similar manner, except that the primitive events are transactions across the network, that is, network transactions rather than bus transactions. A *network transaction* is a one-way transfer of information from an output buffer at the source to an input buffer at the destination that causes some kind of action at the destination, the occurrence of which is not directly visible at the source. This is illustrated in Figure 7.7. The

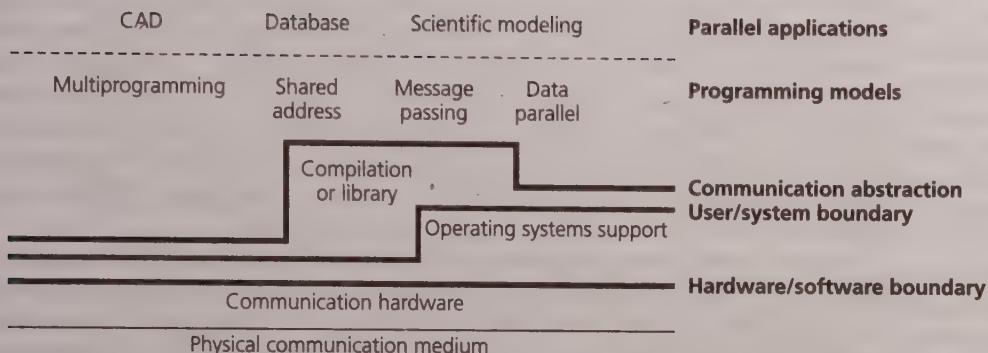


FIGURE 7.6 Layers of parallel architecture. The figure illustrates the critical layers of abstractions and the aspects of the system design that realize each of the layers.

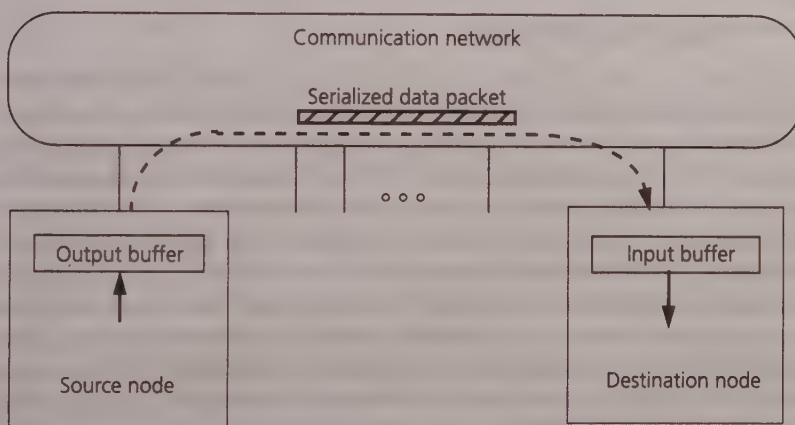


FIGURE 7.7 Network transaction primitive. A one-way transfer for information from a source output buffer to an input buffer of a designated destination, causing some action to take place at the destination.

action may be quite simple (e.g., depositing the data into an accessible location or making a transition in a finite state machine) or it can be more general (e.g., the execution of a message handler routine). The effects of a network transaction are observable only through additional transactions. Traditionally, the communication abstraction supported directly by the hardware in large-scale machines was hidden below the vendor's message-passing library, but increasingly the lower-level abstraction is accessible to user applications.

The differences between bus and network transactions have far-reaching ramifications. The potential design space is even larger than what we saw in Chapter 5, where the bus provided serialization and broadcast and the primitive events were small variations on conventional bus transactions. In large-scale machines, there is

tremendous variation in the primitive network transaction itself, as well as in how these transactions are driven and interpreted at the endpoints. They may be presented to the processor as I/O operations and driven entirely by software, or they may be integrated into the memory system and driven by a dedicated hardware controller. A very wide spectrum of large-scale machines has been designed and implemented, emphasizing a range of programming models, employing a range of primitives at the hardware/software boundary, and providing widely differing degrees of direct hardware support and system software intervention.

To make sense of this great diversity, we proceed step by step. This section first defines a network transaction more precisely and contrasts it more carefully with a bus transaction. It then covers what is involved in realizing shared memory and message-passing abstractions out of this primitive without getting encumbered by the myriad of ways that a network transaction itself might be realized. In later sections, we will work systematically through the space of design options for realizing network transactions and the programming models built upon them.

7.2.1 Primitive Network Transactions

To understand what is involved in a network transaction, let us first reexamine what is involved in a bus transaction, since similar issues arise. Before starting a bus transaction, a protection check has been performed as part of the virtual-to-physical address translation. The *format* of information in a bus transaction is determined by the physical wires of the bus, that is, the data lines, address lines, and command lines. The information to be transferred onto the bus is held in special *output registers* (namely, address, command, and data registers) until it can be driven onto the bus. A bus transaction begins with *arbitration* for the medium. Most buses employ a global arbitration scheme where a processor requesting a transaction asserts a bus request line and waits for the corresponding bus grant. The *destination* of the transaction is implicit in the address. Each module on the bus is configured to respond to a set of physical addresses. All modules examine the address and one responds to the transaction. (If none responds, the bus controller detects the time-out and aborts the transaction.) Each module includes a set of *input registers*, capable of buffering any request to which it might respond. Each bus transaction involves a *request* followed by a *response*. In the case of a read, the response is the data and an associated completion signal; for a write it is just the completion acknowledgment. In either case, both the source and destination are informed of the *completion* of the transaction. In many buses, each transaction is guaranteed to complete according to a well-defined schedule. The primary variation is the length of time that it takes for the destination to turn around the response. In split-transaction buses, the response phase of the transaction may require rearbitration and may be performed in a different order than the requests.

Care is required to avoid deadlock with split transactions (and coherence protocols involving multiple bus transactions per operation) because a module on the bus may be both requesting and servicing transactions. The module must continue

servicing bus requests and accept replies while it is attempting to present its own request. The bus design ensures that, for any transaction that might be placed on the bus, sufficient input buffering exists to accept the transaction at the destination. This can be accomplished in a conservative fashion by providing enough resources for the worst case or in an optimistic fashion by adding a negative acknowledgment signal (NACK). In either case, the solution is relatively straightforward because few concurrent communication operations can be in progress on a bus, and the source and destination are directly coupled by wires.

The discussion of buses raises the issues present in a network transaction as well. These issues include protection, format, output buffering, media arbitration, destination name and routing, input buffering, action, completion detection, transaction ordering, deadlock avoidance, and delivery guarantees. The fundamental difference in the network transaction as compared to the bus is that the source and destination of the transaction are *uncoupled*; that is, there may be no direct wires between them, and there is no global arbitration for the resources in the system. No global information is available to all modules at the same instant, and a huge number of transactions may be in progress simultaneously. These basic differences give the preceding issues a very different character than in a bus. Let us consider each issue in turn.

- **Protection.** As the number of components becomes larger, the coupling between components looses, and the individual components more complex, it may be worthwhile to limit how much each component trusts the others to operate correctly. Whereas in a bus-based system all protection checks are performed by the processor before placing a transaction on the bus, in a scalable system, individual components will often perform checks on the network transaction so that an errant program or faulty hardware component cannot corrupt other components of the system.
- **Format.** Most network links are narrow, so the information associated with a transaction is transferred as a serial stream. Typical links are a few (1 to 16) bits wide. The format of the transaction is dictated by how the information is serialized onto the link, unlike in a bus where it is a parallel transfer whose format is determined by the physical wires. Thus, there is a great deal of flexibility in this aspect of design. We can think of the information in a network transaction as an envelope with more information inside. The envelope includes information germane to the physical network to get the packet from its source to its destination port. This is very much like the command and address portion of a bus transaction, which tells all parties involved what to do with the transaction. Some networks are designed to deliver only fixed-size packets; others can deliver variable-size packets. Very often the envelope contains additional envelopes within it. The communication assist may wrap up the user information in an envelope germane to the remote communication assist and put this within the physical network envelope. This notion of placing information packets within larger envelopes is provides *encapsulation* as it does in traditional networking stacks. It provides a means of abstracting the layers of the communication subsystem.

- *Output buffering.* The source must provide storage to hold information that is to be serialized onto the link, either in registers, FIFOs, or memory. If the transaction is of fixed format, this may be as simple as the output buffer for a bus. Since network transactions are one-way and can potentially be pipelined, it may be desirable to provide a queue of such output registers. If the packet format is variable up to some moderate size, a similar approach may be adopted where each entry in the output buffer is of variable size. If a packet can be quite long, then typically the output controller contains a buffer of descriptors, pointing to the data in memory. It then stages portions of the packet from memory into small output buffers and onto the link, often through DMA transfer.
- *Media arbitration.* There is no global arbitration for access to the network, and many network transactions can be initiated simultaneously. (In buslike networks, such as Ethernet, there is distributed arbitration for the single or small number of transactions that can occur simultaneously.) Initiation of the network transaction places an implicit claim on resources in the communication path from the source to the destination, as well as on resources at the destination. These resources are potentially shared with other transactions. Local arbitration is performed at the source to determine whether or not to initiate the transaction. However, this usually does not imply that all necessary resources are reserved to the destination; the resources are allocated incrementally as the message moves forward.
- *Destination name and routing.* The source must be able to specify enough information to cause the transaction to be routed to the appropriate destination. This is in contrast to the bus, where the source simply places the address on the wire and the destination chooses whether it should accept the request. There are many variations in how routing is specified and performed, but basically the source performs a translation from some logical name for the destination to some form of physical address.
- *Input buffering.* At the destination, the information in the network transaction must be transferred from the physical link into some storage element. As with the output buffer, this may be simple registers or a queue, or it may be delivered directly into memory. The key difference is that transactions may arrive from many sources; in contrast, the source has complete control over how many transactions it initiates. The input buffer is in some sense a shared resource used by many remote processors; how this is managed and what happens when it fills up is a critical issue that we will examine later.
- *Action.* The action taken at the destination may be very simple, say, a memory access, or it may be complex. In either case, it may involve initiating a response.
- *Completion detection.* The source has an indication that the transaction has been delivered into the network but usually no indication that it has arrived at its destination. This completion must be inferred from a response, an acknowledgment, or some additional transaction.

- *Transaction ordering.* Whereas a bus provides strong ordering properties among transactions, in a network the ordering is quite weak. Even on a split-transaction bus with multiple outstanding transactions, we could rely on the serial arbitration for the address bus to provide a global order. Some networks ensure that a sequence of transactions from a given source to a single destination will be seen in order at the destination; others will not even provide this limited assurance. In either case, no node can perceive the global order. In realizing programming models on large-scale machines, ordering constraints must be imposed through network transactions.
- *Deadlock avoidance.* Most modern networks are deadlock-free as long as the modules on the network continue to accept transactions. Within the network, this may require restrictions on permissible routes or other special precautions, as we discuss in Chapter 10. Still, we need to be careful that our use of network transactions to realize programming models does not introduce deadlock. In particular, while we are waiting, unable to source a transaction, we usually will need to continue accepting incoming transactions. This situation is very much like that with split-transaction buses, except that the number of simultaneous transactions is much larger and there is no global arbitration or immediate feedback.
- *Delivery guarantees.* A fundamental decision in the design of a scalable network is the behavior when the destination buffer is full. This is clearly an issue on an end-to-end basis since it is nontrivial for the source to know whether the destination input buffer is available when it is attempting to initiate a transaction. It is also an issue on a link-by-link basis within the network itself. We have two basic options: discard information if the buffer is full or defer transmission until space is available. The first requires a way to detect the situation and retry; the second requires a flow control mechanism and can cause the transactions to back up. We will examine both options later in this chapter.

In summary, a network transaction is a one-way transfer for information from a source output buffer to an input buffer of a designated destination, causing some action to take place at the destination. Let us consider what is involved in realizing the communication abstractions found in common programming models in terms of this primitive.

7.2.2 Shared Address Space

Realizing the shared address space communication abstraction fundamentally requires a two-way request-response protocol, as illustrated abstractly in Figure 7.8. A global address is decomposed into a module number and a local address. For a read operation, a request is sent to the designated module requesting a load of the desired address and specifying enough information to allow the result to be returned to the requestor through a response network transaction. A write is similar, except that the data is conveyed with the address and command to the designated module

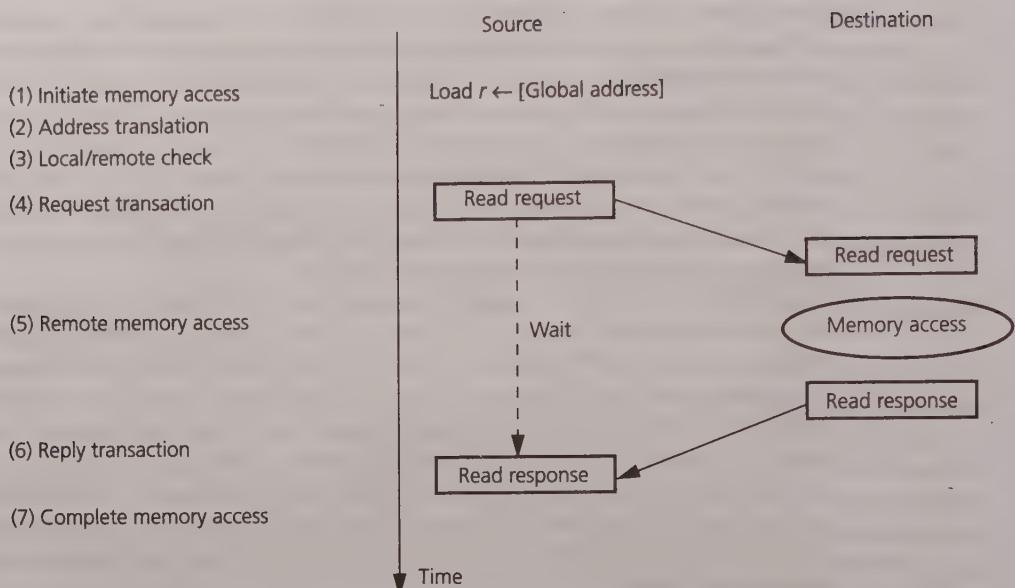


FIGURE 7.8 Shared address space communication abstraction. The figure illustrates the anatomy of a read operation in a large-scale machine in terms of primitive network transactions. (1) The source processor initiates the memory access on global address. (2) The global address is translated into a node number or route and a local address on that node. (3) A check is performed to determine if the address is local to the issuing processor. (4) If not, a read request transaction is performed to deliver the request to the designated processor, which (5) accesses the specified address, and (6) returns the value in a reply transaction to the original node, which (7) completes the memory access.

and the response is merely an acknowledgment to the requestor that the write has been performed. The response informs the source that the request has been received or serviced, depending on whether it is generated before or after the remote action. The response is essential to enforce proper ordering of transactions.

A read request typically has a simple fixed format, describing the address to read and the return information. The write acknowledgment is also simple. If only fixed-size transfers are supported (that is, a word or a cache block), then the read response and write request are also of simple fixed format. This is easily extended to support partial word transfers, say, by including byte enables; however, transfers of arbitrary length require a more general format. For fixed-format transfers, the output buffering is typically as with a bus. The address, data, and command are staged in an output register and serialized onto the link.

The destination name is generally determined as a result of the address translation process, which converts a global address to a module name (or possibly a route to the module) and an address local to that module. Succeeding with the translation usually implies authorization to access the designated destination module; however, the source must still gain access to the physical network and the input

buffer at the remote end. Since a large number of nodes may issue requests to the same destination and there is no global arbitration or direct coupling between the source and destination, the combined storage requirement of the requests may exceed the input buffering at the node. The rate at which the requests can be processed is merely that of a single node, so the requests may back up through the network, perhaps even to the sources. Alternatively, the requests might be dropped in this situation, requiring some mechanism for retrying. Since the network may not be able to accept a request when the node attempts to issue it, each node must be able to accept replies and requests, even while unable to inject its own request, so that the packets in the network can move forward. This is the more general form of the fetch deadlock issue observed in the previous chapter. This input buffer problem and the fetch deadlock problem arise with many different communication abstractions, so they will be addressed in more detail after looking at the corresponding protocols for message-passing abstractions.

When supporting a shared address space abstraction, we need to ask whether it is coherent and what memory consistency model it supports. In this chapter, we consider designs that do not replicate data automatically through caches; Chapters 8 and 9 are devoted to that topic. Thus, each remote read and write goes to the node hosting the address and accesses the location, so coherence is met by the natural serialization involved in going through the network and accessing memory. One important subtlety is that the accesses from remote nodes need to be coherent with accesses from the local node. Thus, if shared data is cached locally, processing the remote reference needs to be cache coherent within the node.

Achieving sequential consistency in scalable machines is more challenging than in bus-based designs because the interconnect does not serialize memory accesses to locations on different nodes. Furthermore, since the latencies of network transactions tend to be large, we are tempted to try to hide it whenever possible. In particular, it is very tempting to issue multiple write transactions without waiting for the completion acknowledgments to come back in between. To see how this can undermine the consistency model, consider our familiar flag-based code fragment executing on a multiprocessor with physically distributed memory but no caches. The variables `A` and `flag` are allocated in two different processing nodes, as shown in Figure 7.9(a). Because of delays in the network, processor P_2 may see the stores to `A` and `flag` in the reverse of the order they are generated. Ensuring point-to-point ordering among packets between each pair of nodes does not remedy the situation because multiple pairs of nodes may be involved. A situation with a possible reordering due to the use of different paths within the network is illustrated in Figure 7.9(b). Overcoming this problem is one of the reasons why writes need to be acknowledged. A correct implementation of this construct will wait for the write of `A` to be completed before issuing the write of `flag`. By using the completion transactions for the write, and the read response, it is straightforward to meet the sufficient conditions for sequential consistency. The deeper design question is how to meet these conditions while minimizing the amount of waiting by determining that the write has been committed and appears to all processors as if it had performed.

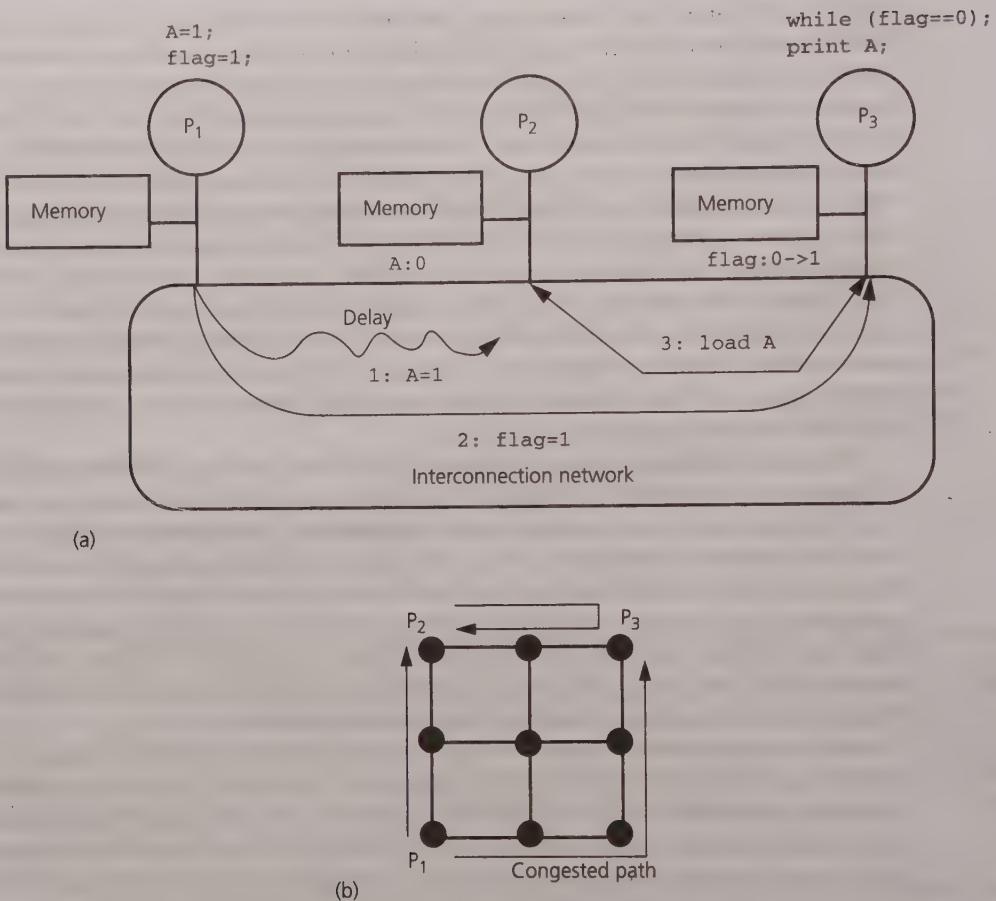


FIGURE 7.9 Possible reordering of memory references for shared flags. The network is assumed to preserve point-to-point order. The processors have no cache, as shown in part (a). The variable A is assumed to be allocated out of P_2 's memory, whereas the variable flag is assumed to be allocated from P_3 's memory. It is assumed that the processors do not stall on a store instruction, as is true for most uniprocessors. It is easy to see that if there is network congestion along the links from P_1 to P_2 , P_3 can get the updated value of flag from P_1 and then read the stale value of A ($A=0$) from P_2 . This situation can easily occur, as indicated in part (b), where messages are always routed first in X -dimension and then in Y -dimension of a 2D mesh.

7.2.3 Message Passing

A send/receive pair in the message-passing model is conceptually a one-way transfer from a source area specified by the source user process to a destination area specified by the destination user process. In addition, it embodies a pairwise synchronization event between the two processes. In Chapter 2, we noted the important semantic variations on the basic message-passing abstractions, such as synchronous and asynchronous message send. User-level message-passing models are implemented in

terms of primitive network transactions, and the different synchronization semantics have quite different implementations (that is, different network transaction protocols). In most early large-scale machines, these protocols were buried within the vendor kernel and library software. In more modern machines, the primitive transactions are exposed to allow a wider set of programming models to be supported.

This chapter uses the concepts and terminology associated with the message-passing interface (MPI). MPI distinguishes the notion of when a call to a send or receive function returns from when a message operation completes. A synchronous send completes once the matching receive has executed, the source data buffer can be reused, and the data is ensured of arriving in the destination receive buffer. A buffered send completes as soon as the source data buffer can be reused, independent of whether the matching receive has been issued; the data may have been transmitted or it may be buffered somewhere in the system.⁵ Buffered send completion is asynchronous with respect to the receiver process. A receive completes when the message data is present in the receive destination buffer. A blocking function, send or receive, returns only after the message operation completes. A nonblocking function returns immediately, regardless of message completion, and additional calls to a probe function are used to detect completion. The protocols are concerned only with message operation and completion, regardless of whether the functions are blocking.

To understand the mapping from user message-passing operations to machine network transaction primitives, let us first consider synchronous messages. The only way for the processor hosting the source process to know whether the matching receive has executed is for that information to be conveyed by an explicit transaction. Thus, the synchronous message operation can be realized with a three-phase protocol of network transactions, as shown in Figure 7.10. This protocol is for a sender-initiated transfer. The send operation causes a “ready-to-send” to be transmitted to the destination, carrying the source process and tag information. The sender then waits until a corresponding “ready-to-receive” has arrived. The remote action is to check a local table to determine if a matching receive has been performed. If not, the ready-to-send information is recorded in the table to await the matching receive. If a matching receive is found, a ready-to-receive response transaction is generated. The receive operation checks the same table. If a matching send is not recorded there, the receive is recorded, including the destination data address.

-
5. The standard MPI mode is a combination of buffered and synchronous modes that gives the implementation substantial freedom and the programmer few guarantees. The implementation is free to choose to buffer data but cannot be assumed to do so. Thus, when the send completes the send buffer can be reused, but it cannot be assumed that the receiver has reached the point of the receive call. Nor can it be assumed that send buffering will break the deadlock associated with two nodes sending to each other and then calling receive. Nonblocking sends can be used to avoid the deadlock, even with synchronous sends. The ready-mode send is a stronger variant of synchronous mode, where it is an error if the receive has not executed by the time the message arrives at the destination. Since the only way to obtain knowledge of the state of the nonlocal processes is through exchange of messages, an explicit message event would need to be used to indicate readiness. The race condition between posting the ready receive and transmitting the synchronization message is very similar to the flags example in the shared address space case.

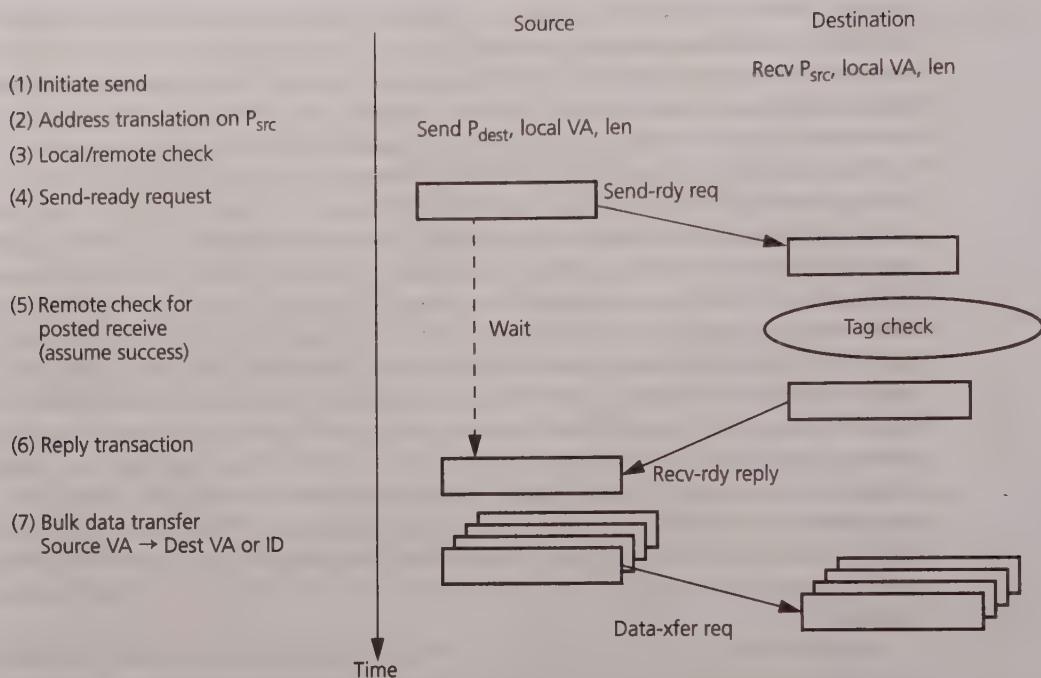


FIGURE 7.10 Synchronous message-passing protocol. The figure illustrates the three-way handshake involved in realizing a synchronous send/receive pair in terms of primitive network transactions.

If a matching send is present, the receive generates a ready-to-receive transaction. When a ready-to-receive arrives at the sender, it can initiate the data transfer. Assuming the network is reliable, the send operation can complete once all the data has been transmitted. The receive will complete once all the data arrives. Note that with this protocol both the source and destination nodes know the local addresses for the source and destination buffers at the time the actual data transfer occurs. The “ready” transactions are small, fixed-format packets whereas the data is a variable-length transfer.

In many message-passing systems, the matching rule associated with a synchronous message is quite restrictive, and the receive specifies the sending process explicitly. This allows for an alternative receiver-initiated protocol in which the match table is maintained at the sender and only two network transactions are required (receive-ready and data transfer).

The buffered send is naively implemented with an optimistic single-phase protocol, as suggested in Figure 7.11. The send operation transfers the source data in a single large transaction with an envelope containing the information used in matching (e.g., source process and tag) as well as length information. The destination strips off the envelope and examines its internal table to determine if a matching receive has been posted. If so, it can deliver the data at the specified receive address.

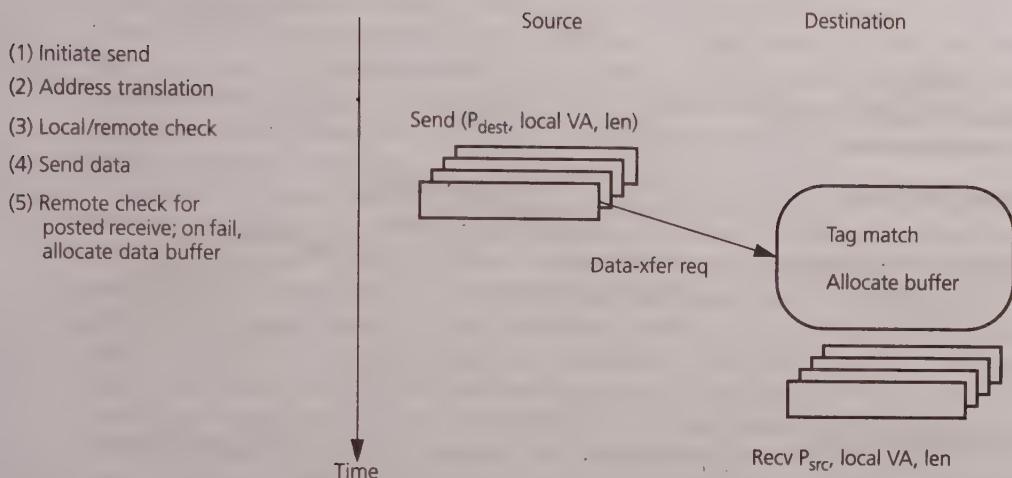


FIGURE 7.11 Asynchronous (optimistic) message-passing protocol. The figure illustrates a naive single-phase optimistic protocol for asynchronous message passing where the source simply delivers that data to the destination, without concern for whether the destination has storage to hold it.

If no matching receive has been posted, the destination allocates storage for the entire message and receives it into the temporary buffer. When the matching receive is posted later, the message is copied to the desired destination area and the buffer is freed.

This simple protocol presents a family of problems. First, the proper destination address of the message data cannot be determined until after examining the process and tag information and consulting the match table. These are fairly expensive operations, typically performed in software. Meanwhile, the message data is streaming in from the network at a high rate. One approach is to always receive the data into a temporary input buffer and then copy it to its proper destination. Of course, this introduces a store-and-forward delay and consumes a fair amount of processing resources.

The second problem with this optimistic approach is analogous to the input buffer problem discussed for a shared address space abstraction since there is no ready-to-receive handshaking before the data is transferred. In fact, the problem is amplified in several respects. First, the transfers are larger, so the total volume of storage needed at the destination is potentially quite large. Second, the amount of buffer storage depends on the program behavior; it is not just a result of the rate mismatch between multiple senders and one receiver, much less a timing mismatch where the data happens to arrive just before the receiver is ready. Several processes may choose to send many messages each to a single process, which happens not to receive them until much later. Conceptually, the asynchronous message-passing model assumes an unbounded amount of storage outside the usual program data structures. Message data is stored until the receives are posted and performed. Fur-

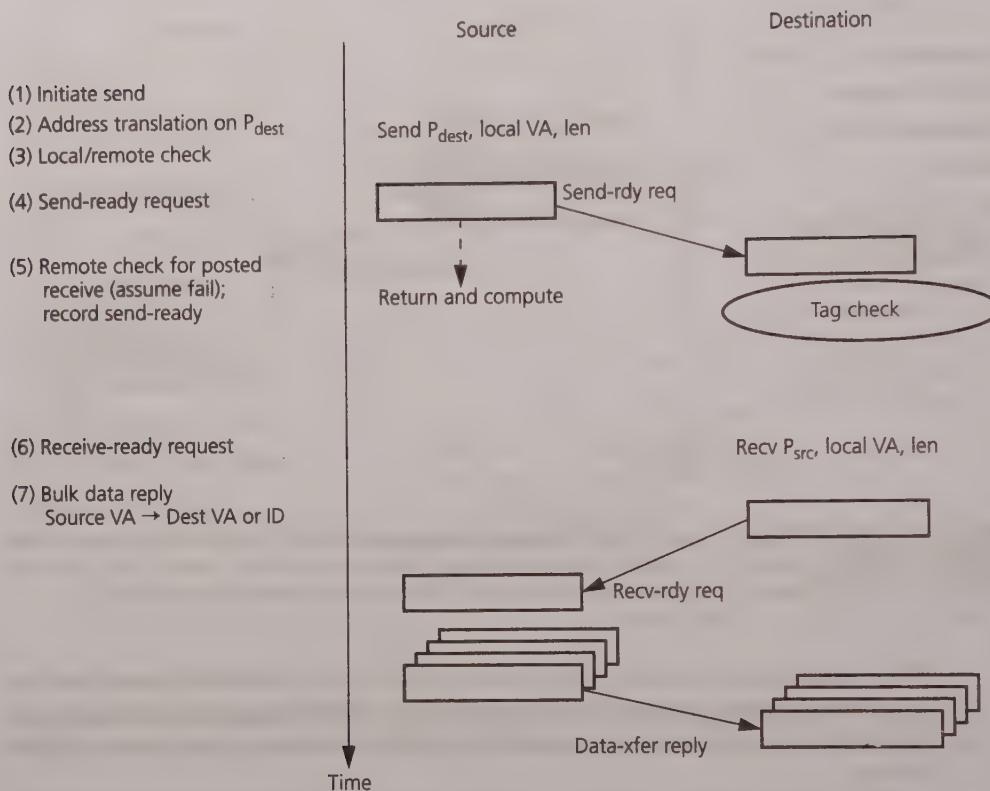


FIGURE 7.12 Asynchronous conservative message-passing protocol. The figure illustrates a one-plus-two-phase conservative protocol for asynchronous message passing. The data is held at the source until the matching receive is executed, making the destination address known before the data is delivered.

thermore, the blocking asynchronous send must be allowed to complete to avoid deadlock in simple communication patterns such as a pairwise exchange. The program needs to continue executing past the send to reach a point where a receive is posted. Our optimistic protocol does not distinguish transient receive-side buffering from the prolonged accumulation of data in the message-passing layer.

More robust message-passing systems use a three-phase protocol for long transfers, as illustrated in Figure 7.12. The send issues a ready-to-send with the envelope but keeps the data buffered at the sender until the destination can accept it. The destination issues a ready-to-receive either when it has sufficient buffer space or when a matching receive has been executed so the transfer can take place to the correct destination area. Note that in this case the source and destination addresses are known at both sides of the transfer before the actual data transfer takes place. For short messages, where the handshake would dominate the actual data transfer cost, a simple credit scheme can be used. Each process sets aside a certain amount of space for

processes that might send short messages to it. When a short message is sent, the sender deducts its destination credit locally, until it receives notification that the short message has been received. In this way, a short message can usually be launched without waiting a round-trip delay for the handshake. The completion acknowledgment is used later to determine when additional short messages can be sent without the handshake.

As with the shared address space, a design issue here concerns whether the source and destination addresses are physical or virtual. The virtual-to-physical mapping at each end can be performed as part of the send and receive calls, allowing the communication assist to exchange physical addresses that can be used for the DMA transfers. Of course, the pages must stay resident during the transfer for the data to stream into memory. However, the residency is limited in time from just before the handshake until the transfer completes. In addition, very long transfers can be segmented at the source so that few resident pages are involved. Alternatively, temporary buffers can be kept resident and the processor relied upon to copy the data from and to the source and destination areas. The resolution of this issue depends heavily on the capability of the communication assist, as discussed in the following.

In summary, the send/receive message-passing abstraction is logically a one-way transfer where the source and destination addresses are specified independently by the two participating processes and an arbitrary amount of data can be sent before any is received. The realization of this class of abstractions in terms of primitive network transactions typically requires a three-phase protocol to manage the buffer space on the ends, although an optimistic single-phase protocol can be employed safely with some form of flow control.

7.2.4 Active Messages

While shared address space and message passing are the dominant programming models for modern parallel machines, it is also possible to provide a communication abstraction that is very close to the network transactions that underlie these models. The most widely used of these low-level communication abstractions is Active Messages (von Eicken et al. 1992). Active Messages constitute request and response transactions in a form that is essentially a restricted remote procedure call. Each message identifies a handler on the destination node that will be invoked upon arrival to process the transaction. A typical request consists of the destination processor address, an identifier for the message handler on that processor, and a small number of data words in the source processor registers that are passed as arguments to the handler. An optimized instruction sequence issues the message into the network via the communication assist. On the destination processor, an optimized instruction sequence extracts the message from the network and invokes the handler on the message data to perform a simple action and issue a response, which identifies a response handler on the original source processor. Higher-level programming models can then be built upon the Active Message primitives by constructing handlers that implement the appropriate protocol (Tucker and Mainwaring 1994; Shah et al. 1998).

Notification of incoming messages (i.e., invoking handlers) may be provided through interrupts or signaling a thread, but it must also be part of issuing an Active Message, in order to allow such a low-level communication operation to be deadlock-free without arbitrary buffering. When attempting to issue a request, the network may be full and the processor will need to allow handlers for incoming messages to be invoked to make progress. Thus, handler invocation can also be provided through explicitly servicing the network with a null message event, called *polling*. Unlike interrupts and threads, this allows handlers to execute synchronously with respect to the destination process.

Bulk transfers can be incorporated into the Active Messages approach either by associating a data buffer with the transaction (a pointer to a source data buffer is provided as part of the request, the buffer is copied to the destination, and a pointer to the destination buffer is provided to the handler), or a memory-to-memory copy can precede the invocation of the handler (Mainwaring and Culler 1996).

7.2.5 Common Challenges

The inherent challenges in realizing programming models in large-scale systems are that each processor has only a limited knowledge of the state of the system, a very large number of network transactions can be in progress simultaneously, and the source and destination of the transaction are decoupled. Each node must infer the relevant state of the system from its own point-to-point events. In this context, it is possible, even likely, for a collection of sources to seriously overload a destination before any of them observe the problem. Moreover, because the latencies involved are inherently large, we are tempted to use optimistic protocols and large transfers. Both of these increase the potential overcommitment of the destination. Furthermore, the protocols used to realize programming models often require multiple network transactions for an operation. All of these issues must be considered to ensure that forward progress is made in the absence of global arbitration. The issues are very similar to those encountered in bus-based designs, but the solutions cannot rely on the constraints of the bus; namely, a small number of processors, a limited number of outstanding transactions, and total global ordering.

Input Buffer Overflow

Consider the problem of contention for the input buffer on a remote node. To keep the discussion simple, assume for the moment fixed-format transfers on a completely reliable network. The management of the input buffer is simple: a queue will suffice. Each incoming transaction is placed in the next free slot in the queue. However, it is possible for a large number of processors to make a request to the same module at once. If this module has a fixed input buffer capacity, on a large system it may become overcommitted. This situation is similar to the contention for the limited buffering within the network, and it can be handled in a similar fashion. One solution is to make the input buffer large and reserve a portion of it for each source. The source must constrain its own demand when it has exhausted its allotment at

the destination. The question then arises, how does the source determine that space is again available for it? There must be some flow control information transmitted back from the destination to the sender. This is a design issue that can be resolved either by acknowledging each transaction or by coupling the acknowledgment with the protocol at a higher level (for example, the reply indicates completion of processing the request).

An alternative approach, common in reliable networks, is for the destination to simply refuse to accept incoming transactions when its input buffer is full. Of course, the data has no place to go, so it remains stuck in the network for a period of time. The network switch feeding the full buffer will be in a situation where it cannot deliver packets as fast as they are arriving. Given that it also has finite buffering, it will eventually refuse to accept packets on its inputs. This phenomenon is called *back pressure*. If the overload on the destination is sustained long enough, the backlog will build in a tree all the way back to the sources. At this point, the sources feel the back pressure from the overloaded destination and are forced to slow down such that the sum of the rates from all the sources sending to the destination are no more than what the destination can receive.

We might worry that the system would fail to function in such situations. Generally, networks are built so that they are *deadlock-free*—that is, messages will make forward progress as long as messages are removed from the network at the destinations (Dally and Seitz 1987). So forward progress will occur. The problem is that with the network so backed up, messages not headed for the overloaded destination will also get stuck in traffic. Thus, the latency of all communication increases dramatically with the onset of this backlog.

Back pressure with a reliable network establishes an interesting “contract” between the processing nodes and the network. From the source point of view, if the network accepts a transaction it is guaranteed that the transaction will eventually be delivered to the destination. However, a transaction may not be accepted for an arbitrarily long period of time, and during that time the source must continue to accept incoming transactions.

Alternatively, the network may be constructed so that the destination can inform the source of the state of its input buffer. This is typically done by reserving a special acknowledgment path in the reverse direction. When the destination accepts a transaction, it explicitly acknowledges the source; if it discards the transaction, it can deliver a negative acknowledgment, informing the source to try again later. Local area networks such as Ethernet, FDDI, and ATM take more austere measures and simply drop the transaction whenever space is not available to buffer it. The source relies on time-outs to decide that it may have been dropped and tries again.

Fetch Deadlock

The input buffer problem takes on an extra twist in the context of the request-response protocols that are intrinsic to a shared address space and present in message-passing implementations. In a reliable network, when a processor attempts to initiate a request transaction, the network may refuse to accept it as a result of

contention for the destination and/or contention within the network. In order to keep the network deadlock-free, the source is required to continue accepting transactions even while it cannot initiate its own. However, the incoming transaction may be a request, which will generate a response. The response cannot be initiated because the network is full.

A common solution to this *fetch deadlock* problem is to provide two logically independent communication networks for requests and responses. This may be realized as two physical networks or as separate virtual channels within a single network with separate output and input buffering. Although it is necessary to continue accepting responses while stalled on attempting to send a request, responses can be completed without initiating further transactions. Thus, response transactions will eventually make progress. This implies that incoming requests can eventually be serviced, which implies that stalled requests will eventually make progress.

An alternative solution is to ensure that input buffer space is always available at the destination when a transaction is initiated by limiting the number of outstanding transactions. In a request-response protocol it is straightforward to limit the number of requests any processor has outstanding; a counter is maintained and each response decrements the counter, allowing a new request to be issued. Standard blocking reads and writes are realized by simply waiting for the response before completing the current request. Nonetheless, with P processors and a limit of k outstanding requests per processor, it is possible for all kP requests to be directed to the same module. Space needs to be available for the $k(P - 1)$ outstanding requests that might be headed for a single destination and for responses to the requests issued by the destination node. Clearly, the available input buffering ultimately limits the scalability of the system. The request transaction is guaranteed to make progress because the network can always sink transactions into available input buffer space at the destination. The fetch deadlock problem arises when a node attempts to generate a request and its outstanding credit is exhausted. It must service incoming transactions in order to receive its own responses, which enable generation of additional requests. Incoming requests can be serviced because it is guaranteed that the requestor reserved input buffer space for the response. Thus, forward progress is ensured even if the node merely queues and ignores incoming transactions while attempting to deliver a response.

Finally, we could adopt the approach we followed for split-transaction buses and NACK the transaction if the input buffer is full. Of course, the NACK may be arbitrarily delayed. Here we assume that the network reliably delivers transactions and NACKs, but the destination node may elect to drop them in order to free up input buffer space. Responses never need to be NACKed because they will be sinked at the destination node, which is the source of the corresponding request and can be assumed to have set aside input buffering for the response. While stalled attempting to initiate a request, we need to accept and sink responses and accept and NACK requests. We can assume that input buffer space is available at the destination of the NACK because it simply uses the space reserved for the intended response. As long as each node provides some input buffer space for requests, we can ensure that even-

tually some request succeeds and the system does not livelock. Additional precautions are required to minimize the probability of starvation.

7.2.6 Communication Architecture Design Space

In the remainder of this chapter, we will examine the spectrum of important design points for large-scale distributed-memory machines. Recall that our generic large-scale architecture consists of a fairly standard node architecture augmented with a hardware communication assist, as suggested by Figure 7.13. The key design issue is the extent to which the information in a network transaction is interpreted directly by the communication assist, without involvement of the node processor. In order to interpret the incoming information, its format must be specified, just as the format of an instruction set must be defined before we can construct an interpreter (that is, a processor) for it. The formatting of the transaction must be performed in part by the source assist, along with address translation, destination routing, and media arbitration. Thus, the processing performed by the source communication assist in generating the network transaction and that performed at the destination together realize the semantics of the lowest-level hardware communication primitives presented to the node architecture. Any additional processing required to realize the desired programming model is performed by the node processor(s), either at user or system level.

Establishing a position on the nature of the processing performed in the two communication assists involved in a network transaction has far-reaching implications

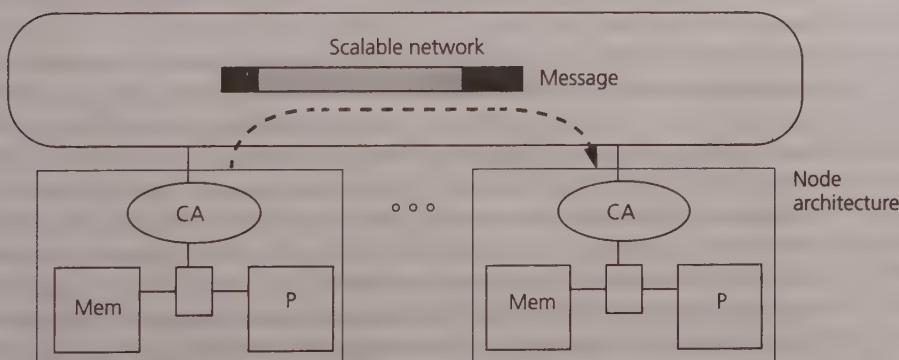


FIGURE 7.13 Processing of a network transaction in the generic large-scale architecture. A network transaction is a one-way transfer of information from an output buffer at the source to an input buffer at the destination that causes some kind of action to occur at the destination, the occurrence of which is not directly visible at the source. The source communication assist (CA) formats the transaction and causes it to be routed through the network. The destination communication assist must interpret the transaction and cause the appropriate actions to take place. The nature of this interpretation is a critical design aspect of scalable multiprocessors.

for the remainder of the design, including how input buffering is performed, how protection is enforced, how many times data is copied within the node, how addresses are translated, and so on. The minimal interpretation of the incoming transaction is not to interpret it at all. It is viewed as a raw physical bit stream and is simply deposited in memory or registers. More specific interpretation provides user-level messages, a global virtual address space, or even a global physical address space. In the following sections, we will examine each of these in turn. We look at important machines that embody the respective design points as case studies.

7.3 PHYSICAL DMA

This section considers designs where no interpretation is placed on the information within a network transaction. This approach is representative of most early message-passing machines, including the nCUBE10 and nCUBE/2, the Intel iPSC, iPSC/2, iPSC860, the Delta, the Ametek, and the IBM SP-1. In addition, most LAN interfaces follow this approach. The hardware can be very simple and the user communication abstraction can be very general, but typical processing overheads are large.

7.3.1 Node-to-Network Interface

The hardware essentially consists of support for physical DMA, as suggested by Figure 7.14. A DMA device or channel typically has associated with it address and length registers, status (e.g., transmit ready, receive ready), and interrupt enables. Either the device is memory mapped or privileged instructions are provided to access the registers. Addresses are physical,⁶ so the network transaction is transferred from a contiguous region of memory. Sending typically requires a trap to the operating system. Privileged software can then provide the source address translation, translate the logical destination node to a physical route, arbitrate for the physical media, and access the physical device. Typically, the data will be copied into a kernel area so that the envelope, including the route and other information, can be constructed. Portions of the envelope, such as the error detection bits, may be generated by the communication assist. The kernel selects the appropriate outgoing channel, sets the channel address to the physical address of the message, and sets the count. (Alternatively, it may build a descriptor containing this information and post it on the transmit queue.) The DMA engine will push the message into the network. When transmission completes, the output channel ready flag is set, and an interrupt is generated, unless it is masked. The message will work its way through the network to the destination, at which point the DMA at the input channel of the destination node must be started to allow the message to continue moving through the network and into the node. (If a delay occurs in starting the input channel or if the message collides in the network with another using the same link, typically the mes-

6. One exception to this is the SBUS used in Sun workstations and servers. It provides virtual DMA, allowing I/O devices to operate on virtual, rather than physical, addresses.

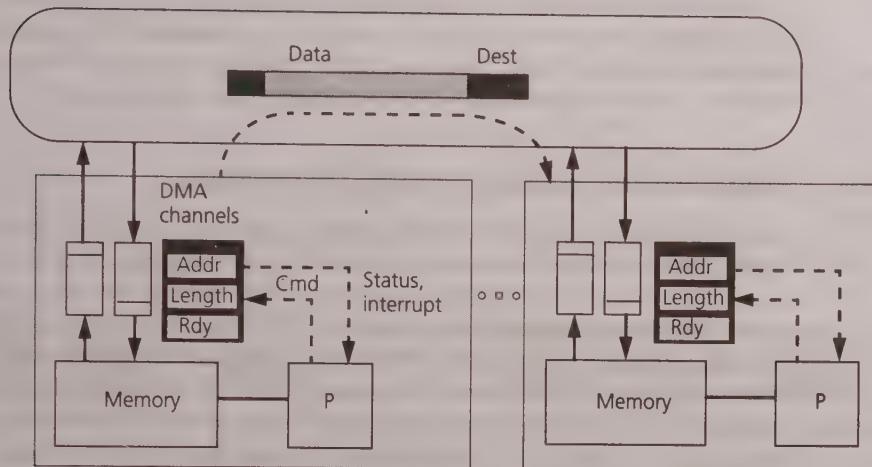


FIGURE 7.14 Hardware support in the communication assist for blind physical DMA. The minimal interpretation is blind physical DMA, which allows the destination communication assist merely to deposit the transaction data into storage, whereupon it will be interpreted by the processor. Since the type of transaction is not determined in advance, kernel buffer storage is used, and processing the transaction will usually involve a context switch and one or more copies.

sage just sits in the network.) Generally, the input channel address register is loaded in advance with the base address where the data is to be deposited. The DMA will transfer words from the network into memory as they arrive. The end-of-message causes the input ready status bit to be set and an interrupt to be generated, unless masked. In order to avoid deadlock, the input channels must be activated to receive messages and drain the network, even if there are output messages that need to be sent on busy output channels.

The key property of this approach is that the destination processor initiates a DMA transfer from the network into a region of memory and the next incoming network transaction is blindly deposited in the specified region of memory. When the system sets up the inbound DMA on the destination side, it cannot determine whether the next message will be a user message or a system message. It will be transferred blindly into the predefined physical region. Message arrival will typically cause an interrupt, so privileged software can inspect the message and either process it or deliver it to the appropriate user process. System software on the node processor interprets the network transaction and (hopefully) provides a clean abstraction to the user.

One potential way to reduce the communication overhead is to allow user-level access to the DMA device. If the DMA device is memory mapped, as most are, this is a matter of setting up the user virtual address space to include the region of the I/O space containing the device control registers. However, with this approach, the protection domain and the level of resource sharing is quite crude. The current user

gets the whole machine. If the user misuses the network, the operating system may not be able to intervene other than to reboot the machine. This approach has certainly been employed in an experimental setting but is not very robust and tends to make the parallel machine into a very expensive personal computer.

7.3.2 Implementing Communication Abstractions

Since the hardware assist in these machines is relatively primitive, the key question becomes how to deliver the newly received network transaction to the user process in a robust, protected fashion. This is where the linkage between programming model and communication primitives occurs. The most common approach is to support the message-passing abstraction directly in the kernel. An interrupt is taken on arrival of a network transaction. The process identifier and tag in the network transaction is parsed, and a protocol action is taken along the lines specified by Figure 7.10 or Figure 7.12. For example, if a matching receive has been posted, the data can be copied directly into the user memory space. If not, the kernel provides buffering or allocates storage in the destination user process to buffer the message until a matching receive is performed. Alternatively, the user process can preallocate communication buffer space and inform the kernel where it wants to receive messages. Some message-passing layers allow the receiver to operate directly out of the buffer rather than receiving the data into its address space.

It is also possible for the kernel software to provide the user-level abstraction of a global virtual address space. In this case, read and write requests are issued either directly through a system call or by trapping on a load or store to a logically remote page. The kernel on the source issues the request and handles the response. The kernel on the destination extracts the user process, command, and destination virtual address from the network transaction and performs the read or write operation (along the lines of Figure 7.8), issuing a response. Of course, the overhead associated with such an implementation of the shared address abstraction is quite large, especially for word-at-a-time operation. Greater efficiency can be gained through bulk data transfers, which make the approach competitive with message passing. Many software shared virtual memory systems have been built along these lines, mostly on clusters of workstations, but the thrust of these efforts is on automatic replication to reduce the amount of communication. They are described in Chapter 9.

Other linkages between the kernel and user are possible. For example, the kernel could provide the abstraction of a user-level input queue and simply append the message to the appropriate queue, following some well-defined policy on queue overflow (Brewer et al. 1995).

7.3.3 A Case Study: nCUBE/2

A representative example of the physical DMA style of machine is the nCUBE/2. The network interface is organized as illustrated in Figure 7.15, where each of the DMA output channels drives an output port and each input DMA channel is associated

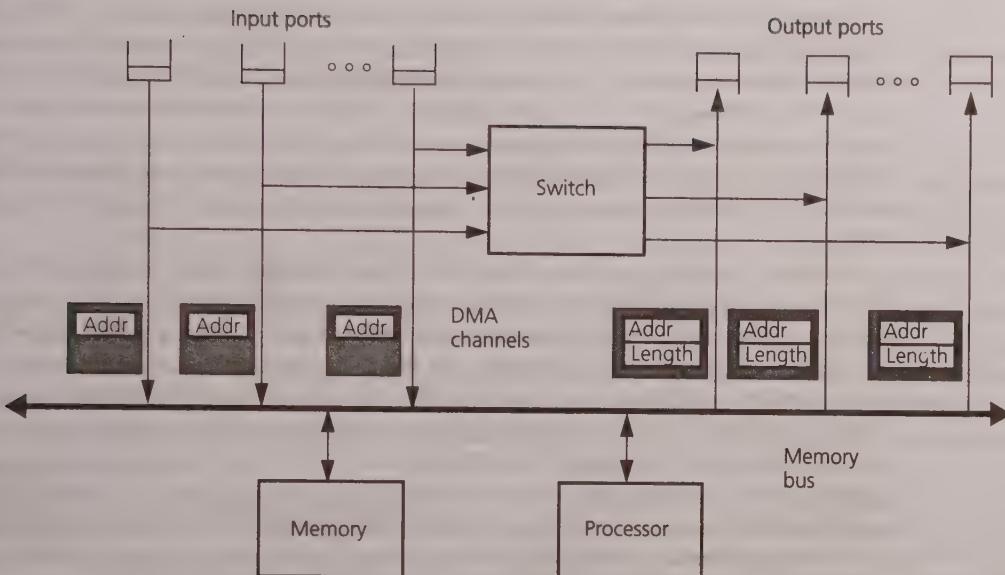


FIGURE 7.15 Network interface organization of the nCUBE/2. Multiple DMA channels drive network transactions directly from memory into the network or from the network into memory. The inbound channels deposit data into memory at a location determined by the processor, independent of the contents. To avoid copying, the machine allows multiple message segments to be transferred as a single unit through the network. A more typical approach is for the processor to provide the communication assist with a queue of inbound DMA descriptors, each containing the address and length of a memory buffer. When a network transaction arrives, a descriptor is popped from the queue and the data is deposited in the associated memory buffer.

with an input port. This machine is an example of a *direct network*, in which data is forwarded from its source to its destination through intermediate nodes. The switch forwards network transactions from input ports to output ports. The network interface inspects the envelope of messages arriving on each input port and determines whether the message is destined for the local node. If so, the input DMA is activated to drain the message into memory. Otherwise, it is forwarded to the proper output port.⁷ Link-by-link flow control ensures that delivery is reliable. User programs are assigned to contiguous subcubes, and the routing is such that the links of the subcube are only used for traffic among the nodes within that subcube; so with space-shared use of the machine, user programs cannot interfere with one another. A peculiarity of the nCUBE/2 is that no count register is associated with the input channels, so the kernels on the source and destination nodes must ensure that incoming messages never overrun the input buffers in memory.

7. This routing step is the primary point where the interconnection network topology, an n -cube, is bound into the design of the node. As we will discuss in Chapter 10, the output port is given by the position of the first bit that differs in the local node address and the message destination address.

To assist in interpreting network transactions and delivering the user data into the desired region of memory without copies, in the nCUBE it is possible to send a series of message segments as a single, contiguous transfer. At the destination, the input DMA will stop at each logical segment. Thus, the destination can take the interrupt and inspect the first segment in order to determine where to direct the remainder, for example, by performing the lookup in the receive table. However, this facility is costly since a start-DMA and interrupt (or busy-wait) is required for each segment.

The best strategy at the kernel level is to keep an input buffer associated with every incoming channel while output buffers are being injected into the output ports (von Eiken et al. 1992). Typically, each message will contain a header, allowing the kernel to dispatch on the message type and take appropriate action to handle it, such as performing the tag match and copying the message into the user data area.

The most efficient and most carefully documented communication abstraction on this platform Active Messages (von Eiken et al. 1992). The first word of the user message contains the address of the user routine that will handle the message. The message arrival causes an interrupt, so the kernel performs a return-from-interrupt to the message handler, with the interrupted user address on the stack. With this approach, a message can be delivered into the network in 13 μ s (16 instructions, including 18 memory references, costing 260 cycles) and extracted from the network in 15 μ s (18 instructions, including 26 memory references, costing 300 cycles). Comparing this with the 150- μ s start-up of the vendor message-passing library reflects the gap between the hardware primitives and the user-level operations in the message-passing programming model. The vendor's message-passing layer uses an optimistic one-way protocol, but matching and buffer management is required.

7.3.4 Typical LAN Interfaces

The simple DMA controllers of the nCUBE/2 are typical of parallel machines and qualitatively different from what is usually found in DMA controllers for peripheral devices and local area networks. Notice that each DMA channel is capable of a single, contiguous transfer. A short instruction sequence sets the channel address and channel limit for the next input or output operation. Traditional DMA controllers provide the ability to chain together a large number of transfers. To initiate an output DMA, a DMA descriptor is chained onto the output DMA queue. The peripheral controller polls this queue, issuing DMA operations and informing the processor as they complete.

Most LAN controllers, including Ethernet LANCE, Sun ATM adapters, and many others, provide a queue of transmit descriptors and a queue of receive descriptors. (There is also a free list of each kind of descriptor. Typically, the queue and its free list are combined into a single ring.) The kernel builds the output message in memory and sets up a transmit descriptor with its address and length, as well as some control information. In some controllers, a single message can be described by a sequence of descriptors, so the controller can gather the envelope and the data from

separate regions of memory. Typically, the controller has a single port into the network, so it pushes the message onto the wire. For Ethernets and rings, each of the controllers inspects the message as it comes by, so a destination address is specified on the transaction rather than a route.

The inbound side is more interesting. Each receive descriptor has a destination buffer address. When a message arrives, a buffer descriptor is popped off the queue, and a DMA transfer is initiated to load the message data into the associated region of memory. If no receive descriptor is available, the message is dropped, and higher-level protocols must retry (just as if the message was garbled in transit). Most devices have configurable interrupt logic, so an interrupt can be generated on every arrival after so many bytes or after a message has waited for too long. The operating system driver manages these input and output queues. The number of instructions required to set up even a small transfer is quite large with such devices, partly because of the formatting of the descriptors and the handshaking with the controller.

7.4

USER-LEVEL ACCESS

The most basic level of hardware interpretation of the incoming network transaction distinguishes user messages from system messages and delivers user messages to the user program without operating system intervention. Each network transaction carries a user/system flag that is examined by the communication assist as the message arrives. In addition, it should be possible to inject a user message into the network at the user level; the communication assist automatically inserts the user flag as it generates the transaction. In effect, this design point provides a *user-level network port*, an access path to the network that can be written and read without system intervention.

7.4.1

Node-to-Network Interface

A typical organization for a parallel machine supporting user-level network access is shown in Figure 7.16. A region of the address space is mapped to the network input and output ports as well as the status register, as indicated in Figure 7.17. The processor can generate a network transaction by writing the destination node number and the data into the output port. The communication assist performs protection check, translates the logical destination node number into a physical address or route, and arbitrates for the medium. It also inserts the message type and any error checking information. Upon arrival, a system message will cause an interrupt so the system can extract it from the network, whereas a user message can sit in the input queue until the user process reads it from the network, popping the queue. If the network backs up, attempts to write messages into the network will fail, and the user process will need to continue to extract messages from the network to make forward progress. Since current microprocessors do not support user-level interrupts, an interrupting user message is treated by the communication assist as a system message, and the system rapidly transfers control to a user-level handler.

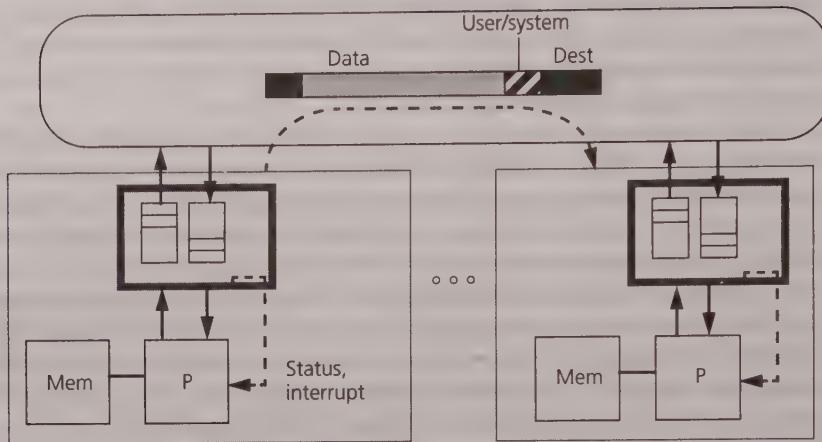


FIGURE 7.16 Hardware support in the communication assist for user-level network ports. The network transaction is distinguished as either system or user. The communication assist provides network input and output FIFOs accessible to the user or system. It marks user messages as they are sent and checks the transaction type as they are received. User messages may be retained in the user input FIFO until extracted by the user application. System transactions cause an interrupt so that they may be handled in a privileged manner by the system. In the absence of user-level interrupt support, interrupting user transactions are treated as special system transactions.

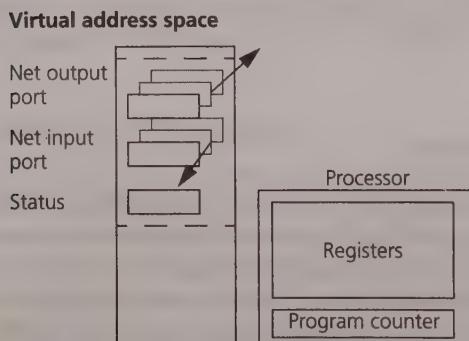


FIGURE 7.17 Typical user-level architecture with network ports. In addition to the storage presented by the instruction set architecture and memory space, a region of the user's virtual address space provides access to the network output port, input port, and status register. Network transactions are initiated and received by writing and reading the ports, plus checking the status register.

One implication of this design point is that the communication primitives allow a portion of the process state to be in the network, having left the source but not arrived at the destination. Thus, if the collection of user processes forming a parallel

program is time-sliced out, the collection of in-flight messages for the program needs to be swapped as well. They will be reinserted into the network or the destination input queues when the program is resumed.

7.4.2 Case Study: Thinking Machines CM-5

The first commercial machine to seriously support a user-level network was the CM-5, introduced in late 1991 by Thinking Machines Corporation. The communication assist is contained in a network interface (NI) chip attached at the memory bus as if it were an additional memory controller, as illustrated in Figure 7.5. The NI provides an input and output FIFO for each of two data networks and a “control network,” which is specialized for global operations such as barrier, broadcast, reduce, and scan. The functionality of the communication assist is made available to the processor by mapping the network input and output ports as well as certain status registers into the address space, as shown in Figure 7.17. The kernel can access all of the FIFOs and registers whereas a user process can only access the user FIFO and status. In either case, communication operations are initiated and completed by reading and writing the communication assist registers using conventional loads and stores. In addition, the communication assist can raise an interrupt. In the CM-5, each network transaction contains a small tag, and the communication assist maintains a table to indicate which tags should raise an interrupt. (All system tags raise an interrupt.)

In the CM-5, it is possible to write a five-word message into the network in $1.5\ \mu s$ (50 cycles) and read one out in $1.6\ \mu s$. In addition, the latency across an unloaded network varies from $3\ \mu s$ for neighboring nodes to $5\ \mu s$ across a 1,024-node machine. An interrupt vectored to user level costs roughly $10\ \mu s$. The user-level handler may process several messages if they arrive in rapid succession. The time to transfer a message into or out of the network interface is dominated by the time spent on the memory bus since these operations are performed as uncached writes and reads. If the message data starts out in registers, it can be written to the network interface as a sequence of buffered stores. However, this must be followed by a load to check if the message was accepted, at which point the write latency is experienced as the write buffer is retired to the NI.

If the message data originates in memory and is to be deposited in memory rather than registers, it is interesting to evaluate whether DMA should be used to transfer data to and from the NI. The critical resource is the memory bus. When using conventional memory operations to access the user-level network port, each word of message data is first loaded into a register and then stored into the NI or memory. If the data is uncacheable, each data word in the network transaction involves four bus transactions. If the memory data is cachable, the transfers between the processor and memory are performed as cache block transfers or are avoided if the data is in the cache. However, the NI stores and loads remain. With DMA transfers, the data is moved only once across the memory bus on each end of the network transaction, using burst mode transfers. However, the DMA descriptor must still be written to the NI. The performance advantages of DMA are lost altogether if the message data

cannot be in cachable regions of memory, so the DMA transfer performed by the NI must be coherent with respect to the processor cache. Thus, it is critical that the node memory architecture support coherent caching. On the receiving side, the DMA must be initiated by the NI based on information in the network transaction and state internal to the NI; otherwise, we are again faced with the problems associated with blind physical DMA. This leads us to place additional interpretation on the network transaction in order to have the communication assist extract address fields. We will consider this approach further in Section 7.5.

The two data networks in the CM-5 provide a simple solution to the fetch deadlock problem: one network can be used for requests and one for responses (Leiserson et al. 1996). When blocking on a request, the node continues to accept incoming replies and requests, which may generate outgoing replies. When blocked on sending a reply, only incoming replies are accepted from the network. Eventually the reply will succeed, allowing the request to proceed. Alternatively, buffering can be provided at each node, with some additional end-to-end flow control to ensure that the buffers do not overflow. Should a user program be interrupted when it is partway through popping a message from the input queue, the system will extract the remainder of the message and push it back into the front of the input queue before resuming the program.

7.4.3 User-Level Handlers

Several experimental architectures have investigated a tighter integration of the user-level network port with the processor, including the Manchester Dataflow Machine (Gurd, Kerkham, and Watson 1985), Sigma-1 (Shimada, Hiraki, and Nishida 1984), iWARP (Borkar et al. 1990), Monsoon (Papadopoulos and Culler 1990), EM-4 (Sakai, Kodama, and Yamaguchi 1991), and the J-machine (Dally, Keen, and Noakes 1993). The key difference is that the network input and output ports are processor registers, as suggested by Figure 7.18, rather than special regions of memory. This substantially changes the engineering of the node since the communication assist is essentially a function unit in the processor. The latency of each of the operations is reduced substantially since data is moved in and out of the network with register-to-register instructions. The bandwidth demands on the memory bus are reduced, and the design of the communication support is divorced from the design of the memory system. However, the processor is involved in every network transaction. Large data transfers consume processor cycles and are likely to pollute the processor cache.

Interestingly, the experimental machines have arrived at a similar design point from vastly different approaches. The iWARP machine (Borkar et al. 1990), developed jointly by CMU and Intel, binds two registers in the main register file to the head of the network input and output ports. The processor may access the message on a word-by-word basis as it streams in from the network. Alternatively, a message can be spooled into memory by a DMA controller. The processor specifies which message it desires to access via the port registers by specifying the message tag, much as in a traditional receive call. Other messages are spooled into memory by the

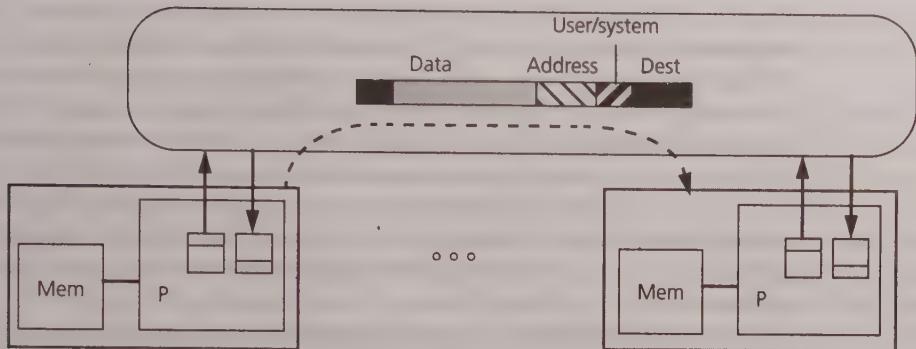


FIGURE 7.18 Hardware support in the communication assist for user-level handlers. The basic level of support required for user-level handlers is that the communication assist can determine that the network transaction is destined for a user process and make it directly available to that process. This either means that each process has a logical set of FIFOs, or a single set of FIFOs is time-shared among user processes.

DMA controller using an input buffer queue to specify the destination address. The extra hardware mechanism to direct one incoming and one outgoing message through the register file was motivated by systolic algorithms where a stream of data is pumped through processors in a highly regular pipeline, doing a small amount of computation as the stream flows through. By interpreting the tag, or virtual channel in iWARP terms, in the network interface, the memory-based message flows are not hindered by the register-based message flow. By contrast, in a CM-5 style of design, all messages are interleaved through a single input buffer.

The *T machine (Nikhil, Papadopoulos, and Arvind 1993), proposed by MIT and Motorola, offered a more general-purpose architecture for user-level message handling. It extended the Motorola 88110 RISC microprocessor to include a network function unit containing a set of registers much like the floating-point unit. In this design, a multiword outgoing message is composed in a set of output registers, and a special instruction causes the network function unit to send it out. There are several output register sets forming a queue and the send advances the queue, exposing the next available set to the user. Function unit status bits indicate whether an output set is available; these can be used directly in branch instructions. There are also several input message register sets, so when a message arrives, it is loaded into an input register set and a status bit is set or an interrupt is generated. Additional hardware support is provided to allow the processor to dispatch rapidly to the address specified by the first word of the message.

The *T design drew heavily on previous efforts supporting message-driven execution and dataflow architectures, especially the J-machine (Dally, Keen, and Noakes 1993), Monsoon (Papadopoulos and Culler 1990), and EM-4 (Sakai, Kodama, and Yamaguchi 1991). These earlier designs employ rather unusual processor architectures, so the communication assist is not clearly articulated. The J-machine design provides two execution contexts, each with a program counter and small register set.

The “system” execution context has priority over the “user” context. The instruction set includes a segmented memory model, with one segment being a special on-chip message input queue. There is also a message output port for each context. The first word of the network transaction is specified as being the address of the handler for the message. Whenever the user context is idle and a message is present in the input queue, the head of the message is automatically loaded into the program counter and an address register is set up to reference the rest of the message. The handler must extract the message from the input buffer before suspending or completing. Arrival of a system-level message preempts the user context and initiates a system handler.

In the Monsoon design, a network transaction is of fixed format and specified to contain a handler address, data frame address, and a 64-bit data value. The processor supports a large queue of such small messages. The basic instruction scheduling mechanism and the message handling mechanism are deeply integrated. In each instruction fetch cycle, a message is popped from the queue, and the instruction specified by the first word of the message is executed. Instructions are in a $1 + x$ address format and specify an offset relative to the frame address where a second operand is located. Each frame location contains *presence bits*, which indicate if the location is full or empty. If the location is empty the data word of the message is stored in the specified location (like a store accumulator instruction). If the location is not empty, its value is fetched, an operation is performed on the two operands, and one or more messages carrying the result are generated, either for the local queue or a queue across the network. In earlier, more traditional dataflow machines, the network transaction carries an instruction address and a tag, which is used in an associative match to locate the second operand, rather than simple frame relative addressing. Later hybrid machines (Nikhil and Arvind 1989; Gafe and Hoch 1990; Culler et al. 1991; Sakai, Kodama, and Yamaguchi 1991) execute a sequence of instructions for each message dequeue-and-match operation.

7.5 DEDICATED MESSAGE PROCESSING

A third important design style for large-scale distributed-memory machines seeks to allow sophisticated processing of the network transaction using dedicated hardware resources but without binding the interpretation in the hardware design. The interpretation is performed by software on a dedicated communication (or message) processor (CP) that operates directly on the network interface. With this capability, it is natural to consider off-loading the protocol processing associated with the message-passing abstraction to the CP. It can perform the buffering, matching, copying, and acknowledgment operations. It is also reasonable to support a global address space where the CP performs the remote read operation on behalf of the requesting node. The CPs can cooperate to provide a general capability to move data from one region of the global address space to another. The CP can provide synchronization operations and even combinations of data movement and synchronization, such as writing data and setting a flag or enqueueing data. This section looks at the

basic organizational properties of machines of this class to understand the key design issues. We will examine in detail two machines as case studies, the Intel Paragon and the Meiko CS-2.

A generic organization for this style of design is shown in Figure 7.19, where the compute processor (P) and communication processor (CP) are symmetric and both reside on the memory bus. This essentially starts with a bus-based SMP as the node (as outlined in Chapter 5), extended with a primitive network interface similar to that described in the previous two sections. One of the processors in the SMP node is specialized in software to function as a dedicated CP. An alternative organization is to have the CP embedded into the network interface, as shown in Figure 7.20. These two organizations have different latency, bandwidth, and cost trade-offs, which we will examine a little later. Conceptually, they are very similar. The CP typically executes at the system privilege level, relieving the machine designer of the issues associated with a user-level network interface discussed previously. The two processors communicate via shared memory, which typically takes the form of a command queue and response area, so the change in privilege level comes essentially for free as part of the hand-off. Since the design assumes that a system-level processor is responsible for managing network transactions, these designs generally allow word-by-word access to the NI FIFOs as well as DMA. The CP can inspect portions of the message and decide what actions to take. The CP can poll the network and the command queues to move the communication process along.

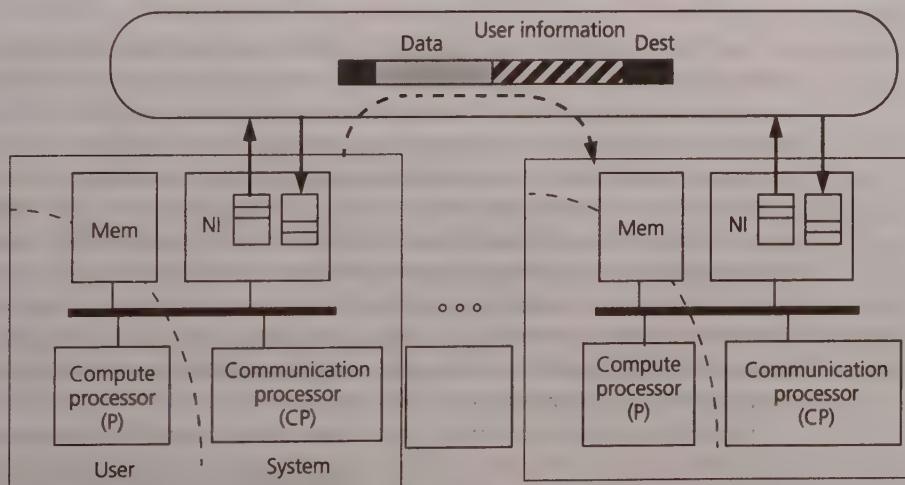


FIGURE 7.19 Machine organization for dedicated message processing with a symmetric processor. Each node has a processor, symmetric with the main processor on a shared memory bus, that is dedicated to initiating and handling network transactions. Being dedicated, it can always run in system mode, so transferring data through memory implicitly crosses the protection boundary. The CP can provide any additional protection checks of the contents of the transactions.

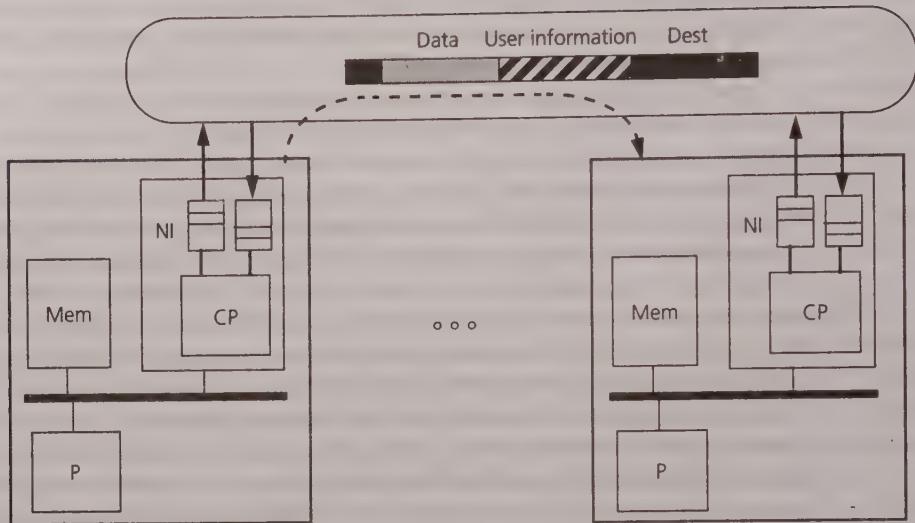


FIGURE 7.20 Machine organization for dedicated message processing with an embedded processor. The communication assist consists of a dedicated, programmable CP embedded in the network interface. It has a direct path to the network that does not utilize the memory bus shared by the main processor.

The CP provides the compute processor with a very clean abstraction of the network interface. All the details of the physical network operation are hidden, such as the hardware input/output buffers, the status registers, and the representation of routes. A message can be sent by simply writing it, or a pointer to it, into shared memory. Control information is exchanged between the processors using familiar shared memory synchronization primitives, such as flags and locks. Incoming messages can be delivered directly into memory by the CP, along with notification to the compute processor via shared variables. With a well-designed user-level abstraction, the data can be deposited directly into the user address space. A simple low-level abstraction provides each user process in a parallel program with a logical input queue and output queue. In this case, the flow of information in a network transaction is as shown in Figure 7.21.

These benefits are not without costs. Since communication between the compute processor and CP is via shared memory within the node, communication performance is strongly influenced by the efficiency of the cache coherency protocol. A review of Chapter 5 reveals that these protocols are primarily designed to avoid unnecessary communication when two processors are operating on mostly distinct portions of a shared data structure. The shared communication queues are a very different situation. The producer writes an entry and sets a flag. The consumer must see the flag update, read the data, and clear the flag. Eventually, the producer will see the cleared flag and rewrite the entry. All the data must be moved from the producer to the consumer with minimal latency. We will see shortly that inefficiencies in tra-

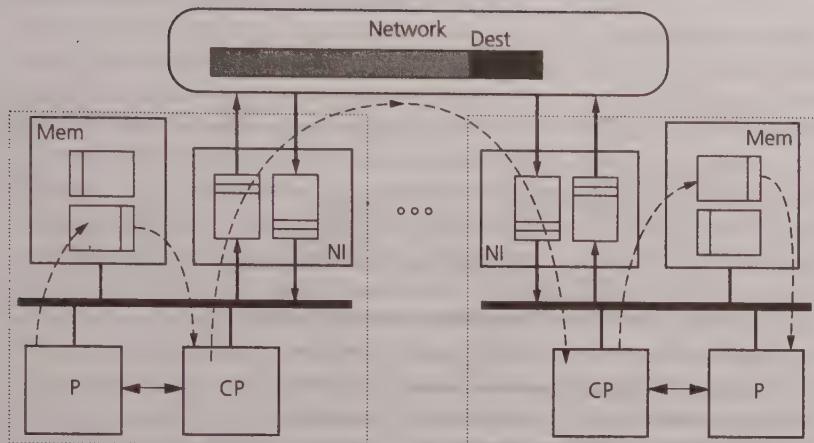


FIGURE 7.21 Flow of a network transaction with a symmetric communication processor. Each network transaction flows through memory or at least across the memory bus in a cache-to-cache transfer between the main processor and the memory processor. It crosses the memory bus again between the CP and the network interface.

ditional coherency protocols make this latency significant. For example, before the producer can write a new entry, the copy of the old one in the consumer's cache must be invalidated. One might imagine that an update protocol or even uncached writes might avoid this situation, but then a bus transaction will occur for every word rather than every cache block.

A second problem is that the function performed by the CP is concurrency intensive. It handles requests from the compute processor, messages arriving from the network, and messages going out and into the network all at once. By folding all these events into a single sequential dispatch loop, they can only be handled one at a time. This can seriously impair the message processing rate of the hardware.

Finally, the ability of the CP to deliver messages directly into memory does not completely eliminate the possibility of fetch deadlock at the user level, although it can ensure that the physical resources are not stalled when an application deadlocks. A user application may need to provide some additional level of flow control.

7.5.1 Case Study: Intel Paragon

To make these general issues concrete, let us examine how they arise in an important machine of this ilk—the Intel Paragon, first shipped in 1992. Each node is a shared memory multiprocessor with two or more 50-MHz i860XP processors, a network interface (NI) chip, and 16 or 32 MB of memory, connected by a 64-bit, 400-MB/s, cache-coherent memory bus, as shown in Figure 7.22. In addition, two DMA engines (one for sending and the other for receiving) are provided to burst data

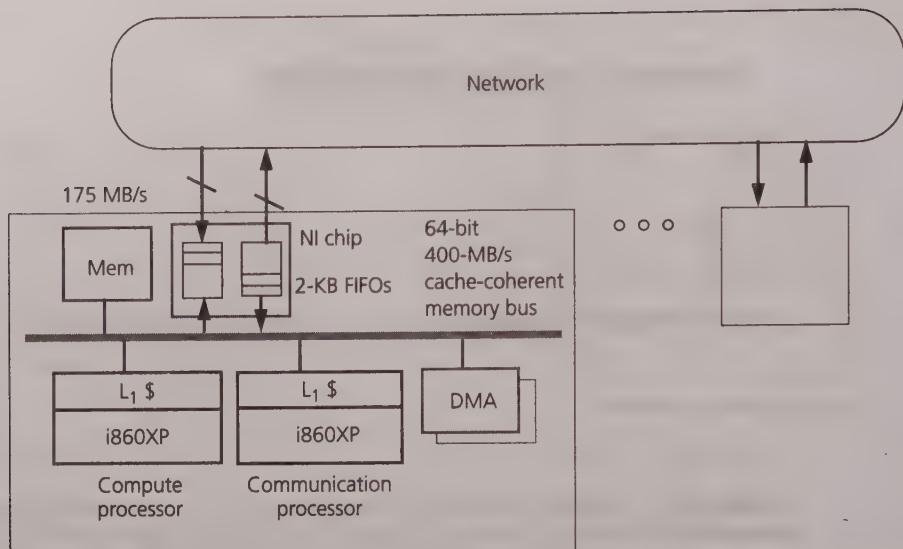


FIGURE 7.22 Intel Paragon machine organization. Each node of the machine includes a dedicated CP, identical to the compute processor on a cache-coherent memory bus, which has a simple, system-level interface to a fast, reliable network and two cache-coherent DMA engines that respect the flow control of the NIC.

between memory and the network. DMA transfers operate within the cache coherency protocol and are throttled by the network interface before buffers are over- or underrun. One of the processors is designated as a CP to handle network transactions and message-passing protocols while the other is used as a compute processor for general computing. “I/O nodes” are formed by adding I/O daughter cards for SCSI, Ethernet, and HPPI connections.

The i860XP uses write-back caching normally, but it can also be configured to use write-through and write-once policies under software or hardware control. The write buffers can hold two successive stores to prevent stalling on write misses. The cache controllers of the i860XP implement a variant of the MESI (modified, exclusive, shared, invalid) cache consistency protocol discussed in Chapter 5. The external bus interface also supports a three-stage address pipeline (i.e., 3 outstanding bus cycles) and burst mode with transfer length of 2 or 4 at 400 MB/s.

The NI chip connects the 64-bit synchronous memory bus to 16-bit asynchronous (self-timed) network links. A 2-KB transmit FIFO (tx) and a 2-KB receive FIFO (rx) are used to provide rate matching between the node and a full-duplex 175-MB/s network link. The head of the rx FIFO and the tail of the tx FIFO are accessible to the node as a memory-mapped NI chip I/O register. In addition, a status register contains flags that are set when the FIFO is full, empty, almost full, or almost empty and when an end-of-packet marker is present. The NI chip can optionally generate an interrupt when each flag is set. Reads and writes to the NI

chip FIFOs are uncached and must be done one double word (64 bits) at a time. The first word of a message must contain the route (X-Y displacements in a 2D mesh), but the hardware does not impose any other restriction on the message format. In particular, it does not distinguish between system and user messages. In addition, the NI chip also performs parity and CRC checks to maintain end-to-end data integrity.

Two DMA engines, one for sending and the other for receiving, can transfer a contiguous block of data between main memory and the NI chip at 400MB/s. The memory region is specified as a physical address, aligned on a 32-byte boundary, with a length between 64 bytes and 16 KB (one DRAM page) in multiples of 32 bytes (a cache block). During DMA transfer, the DMA engine snoops on the processor caches to ensure consistency. Hardware flow control prevents the DMA from overflowing or underflowing the NI chip FIFOs. If the output buffer is full, the send-DMA will pause and free the bus. Similarly, the receive-DMA pauses when the input buffer is empty. The bus arbitrator gives priority to the DMA engine over the processors. A DMA transfer is started by storing an address-length pair to a memory-mapped DMA register using the `stio` instruction. Upon completion, the DMA engine sets a flag in a status register and optionally generates an interrupt.

With this hardware configuration, a small message of just less than two cache blocks (seven words) can be transferred from registers in one compute processor to registers waiting on the transfer in another compute processor in just over 10 μ s (500 cycles). This time breaks down almost equally between the three processor-to-processor transfers: compute processor to CP across the bus, CP to CP across the network, and CP to compute processor across the bus on the remote node. It may seem surprising that transfers between two processors on a cache-coherent memory bus would have the same latency as transfers between two CPs through the network, especially since the transfers between the processor and the network interface involve a transfer across the same bus.

Let's look at this situation in a little more detail. An i860 processor can write a cache block from registers using two quad-word store instructions. Suppose that part of the block is used as a full-empty flag. In the typical case, the last operation on the block was a store by the consumer to clear the flag; with the Paragon MESI protocol, this writes through to memory, invalidates the producer block, and leaves the consumer in the exclusive state. The producer's load on the flag that finds the flag clear misses in the producer cache, reads the block from memory, and downgrades the consumer block to shared state. The first store writes through and invalidates the consumer block, but it leaves the producer block in shared state since sharing was detected when the write was performed. The second store also writes through, but since there is no sharer it leaves the producer in the exclusive state. The consumer eventually reads the flag, misses, and brings in the entire line. Thus, four bus transactions are required for a single cache block transfer. (By having an additional flag that allows the producer to check that several blocks are empty, this can be reduced to three bus transactions. This is left as an exercise.) Data is written to the network interface as a sequence of uncached double-word stores. These are all pipelined through the write buffer; however, they do involve multiple bus transfers. Before writing the data, the CP needs to check that room is available in the output buffer to

hold it. Rather than pay the cost of an uncached read, it checks a bit in the processor status word corresponding to a masked “output buffer empty” interrupt. On the receiving end, the CP reads a similar “input nonempty” status bit and then reads the message data as a series of uncached loads. The actual NI-to-NI transfer takes only about 250 ns plus 40 ns per hop in an unloaded network.

For a bulk memory-to-memory transfer, there is additional work for the CP to start the send-DMA from the user source region. This requires about 2 μ s (100 cycles). The DMA transfer bursts at 400 MB/s into the network output buffer of 2,048 bytes. When the output buffer is full, the DMA engine backs off until a number of cache blocks are drained into the network. On the receiving end, the CP detects the presence of an incoming message, reads the first few words containing the destination memory address, and starts the receive-DMA to drain the remainder of the message into memory. For a large transfer, the send-DMA engine will still be moving data out of memory when the receive-DMA engine is moving data into memory, and a portion of the transfer occupies the buffers and network links in between. At this point, the message moves forward at the 175 MB/s of the network links. The send- and receive-DMA engines periodically kick in and move data to or from network buffers in 400-MB/s bursts.

A review of the requirements on the CP shows that it is responding to a large number of independent events on which it must take action. These include the current user program writing a message into the shared queue, the kernel on the compute processor writing a message into a similar “system queue,” the network delivering a message into the NI input buffer, the NI output buffer going empty as a result of the network accepting a message, the send-DMA engine completing, and the receive-DMA engine completing. The bandwidth of the CP is determined by the time it takes to detect and dispatch on these various events. While handling any one of the events, all the others are effectively locked out. Additional hardware, such as the DMA engines, is introduced to minimize the work in handling any particular event and to allow data to flow from the source storage area (registers or memory) to the destination storage area in a fully pipelined fashion. However, the communication rate (messages per second) is still limited by the sequential dispatch loop in the CP. In addition, the software on the CP, which keeps data flowing, avoids deadlock, and avoids starving the network, is rather tricky. Logically, it involves a number of independent cooperating threads, but these are folded into a single sequential dispatch loop that keeps track of the state of each of the partial operations. This concurrency problem is addressed by our next case study.

The basic architecture of the Paragon is employed in the ASCI Red machine, which is the first machine to sustain a TFLOPS (one trillion floating-point operations per second). This machine will contain 4,536 nodes with dual 200-MHz Pentium Pro processors and 64 MB of memory. It uses an upgraded version of the Paragon network with 400-MB/s links, still in a grid topology. The machine is spread over 85 cabinets, occupying about 1,600 square feet and drawing 800 kW of power. Forty of the nodes provide I/O access to large RAID storage systems, an additional 32 nodes provide operating system services to a lightweight kernel operating on the individual nodes, and 16 nodes provide “hot” spares.

Many cluster designs employ SMP nodes as the basic building block, with a scalable high-performance LAN or SAN. This approach admits the option of dedicating a processor to message processing or of having that responsibility taken on by processors as demanded by message traffic. One key difference is that networks such as that used in the Paragon will back up and stop all communication progress, including system messages, unless the inbound transactions are serviced by the nodes. Thus, dedicating a processor to message handling provides a more robust design. (In special cases, such as attempting to set the record on the LINPACK benchmark, even the Paragon and ASCI Red are run with both processors doing user computation.) Clusters usually rely on other mechanisms to keep communication flowing, such as dedicated processing within the network interface card, as we will discuss in Section 7.7.

7.5.2 Case Study: Meiko CS-2

The Meiko CS-2 provides a representative concrete design with an asymmetric CP that is closely integrated with the network interface and has a dedicated path to the network. The node architecture is essentially that of a Sun SparcStation 10, with two standard superscalar Sparc modules on the MBUS, each with an L₁ cache on chip and an L₂ cache on the module. Ethernet, SBUS, and SCSI connections are also accessible over the MBUS through a bus adapter to provide I/O. (A high-performance variant of the node architecture includes two Fujitsu μVP vector units sharing a three-ported memory system. The third port is the MBUS, which hosts the two compute processors and the communications module, as in the basic node.) The communications module functions as either another processor module or a memory module on the MBUS, depending on its operation. The network links provide 50-MB/s bandwidth in each direction. This machine takes a unique position on how the network transaction is interpreted and on how concurrency is supported in communication processing.

A network transaction on the Meiko CS-2 is a code sequence transferred across the network and executed directly by the remote CP. The network is *circuit switched*, which means that a channel is established and held open for the duration of the network transaction execution. The channel closes with an acknowledgment if the channel was established and the transaction executed to completion successfully. A NACK is returned if connection is not established, a CRC error occurs, the remote execution times out, or a conditional operation fails. The control flow for network transactions is straight-line code with conditional abort but no branching. A typical cause of time-out is a page fault at the remote end. The kinds of operations that can be included in a network transaction include read, write, or read-modify-write of remote memory; setting events; simple tests; DMA transfers; and simple reply transactions. Thus, the format of the information in a network transaction is fairly extensive. It consists of a context identifier, a start symbol, a sequence of operations in a concrete format, and an end symbol. A transaction is between 40 and 320 bytes long. We will return to the operations supported by network transactions in more detail after looking at the machine organization.

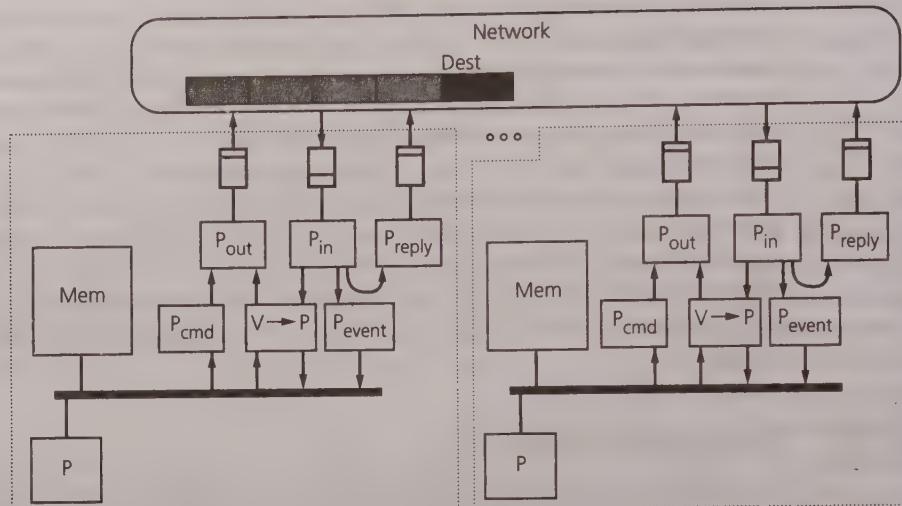


FIGURE 7.23 Meiko CS-2 conceptual structure with multiple specialized communication processors. Each of the individual aspects of generating and processing network transactions is associated in independently operating hardware function units.

Based on the preceding discussion, it makes sense to consider decomposing the CP into several independent processors, as indicated by Figure 7.23. A command processor (P_{cmd}) waits for communication commands to be issued on behalf of the user or system and carries them out. Since it resides as a device on the memory bus, it can respond directly to reads and writes of addresses for which it is responsible, rather than polling a shared memory location as would a conventional processor. It carries out its work by pushing route information and data into the output processor (P_{out}) or by moving data from memory to the output processor. It may require assistance of a device responsible for virtual-to-physical ($V \rightarrow P$) address translation. It also provides whatever protection checks are required for user-to-user communication. The output processor P_{out} monitors the status of the network output FIFO and delivers network transactions into the network. An input processor (P_{in}) waits for the arrival of a network transaction and executes it. This may involve delivering data into memory, posting a command to an event processor (P_{event}) to signal completion of the transaction, or posting a reply operation to a reply processor (P_{reply}), which operates very much like the output processor.

The Meiko CS-2 essentially provides these independent functions, although they operate as time-multiplexed threads on a single microprogrammed processor called the *elan* (Homewood and McLaren 1993). This makes the communication between the logical processors very simple and provides a clean conceptual structure, but it does not actually keep all the information flows progressing smoothly. The actual functional organization of the elan is depicted in Figure 7.24. A command is issued by the compute processor to the command processor via an exchange instruction, which swaps the value of a register and a memory location. The memory location is

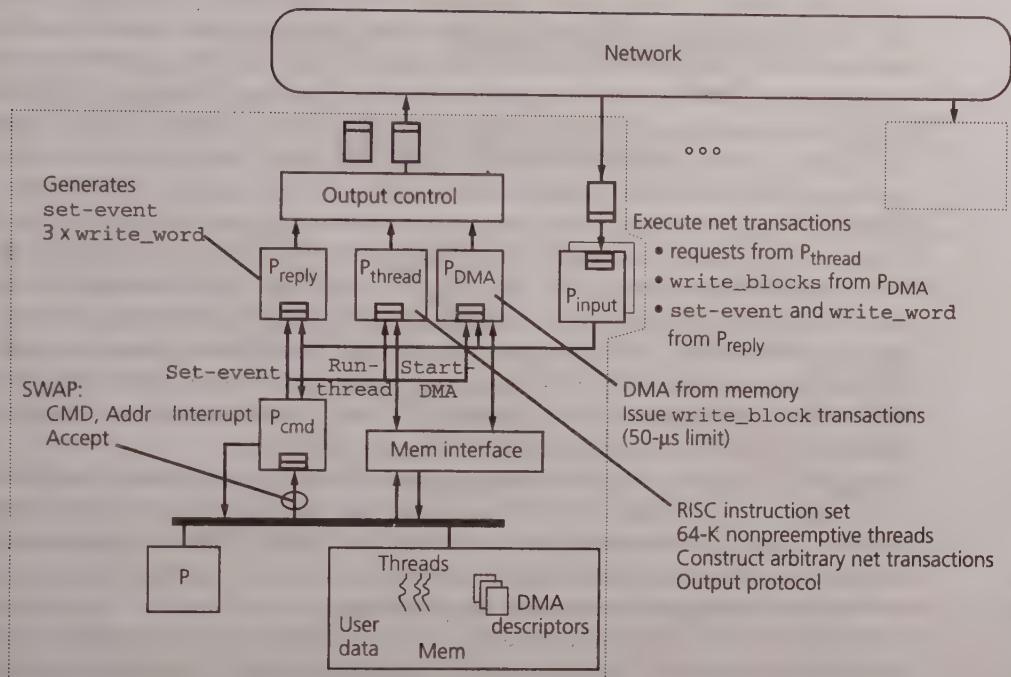


FIGURE 7.24 Meiko CS-2 machine organization. The communication assist provides five simple processors as time slices on a single microcoded processor. Four of them are dedicated to specific functions: receiving commands from the host, accepting transactions from the network, performing DMA transfers, and issuing replies. One, the thread processor, executes user-level code to generate network transactions and issue requests to the other processors.

mapped to the head of the command processor input queue. The value returned to the processor indicates whether the enqueue command was successful or whether the queue was already full. The value given to the command processor contains a command type and a virtual address. The command processor basically supports three commands: start-DMA, in which case the address points to a DMA descriptor; set-event, in which case the address refers to a simple event data structure; or start-thread, in which case the address specifies the first instruction in the thread. The DMA processor reads data from memory and generates a sequence of network transactions to cause data to be stored into the remote node. The command processor also performs event operations, which involve updating a small event data structure and possibly raising an interrupt for the main processor in order to wake a sleeping thread. The start-thread command is conveyed to a simple RISC thread processor, which executes an arbitrary code sequence to construct and issue network transactions. Network transactions are interpreted by the input processor, which may cause threads to execute, DMA to start, replies to be issued, or events to be set. The reply is simply a set-event operation with an optional write of three words of data.

To make the machine operation more concrete, let us consider a few simple operations. Suppose a user process wants to write data into the address space of another process of the same parallel program. Protection is provided by a capability for a communication context that both processes share. The source compute processor builds a DMA descriptor and issues a start-DMA command. Its DMA processor reads the descriptor and transfers the data as a sequence of blocks, each of which involves loading up to 32 bytes from memory and forming a `write_block` network transaction. The input processor on the remote node will receive and execute a series of `write_block` transactions, each containing a user virtual memory address and the data to be written at that address. Reading a block of data from a remote address space is somewhat more involved. A thread is started on the local CP, which issues a start-DMA transaction. The input processor on the remote node passes the start-DMA and its descriptor to the DMA processor on the remote node, which reads data from memory and returns it as a sequence of `write_block` transactions. To detect completion, these can be augmented with set-event operations.

In order to support direct user-to-user transfers, the communications processor on the Meiko CS-2 contains its own page table. The operating system on the main processor keeps this consistent with the normal page tables. If the CP experiences a page fault, an interrupt is generated at the processor so that the operating system there can fault in the page and update the page tables.

The major shortcoming with this design is that the thread processor is quite slow and is nonpreemptively scheduled. This makes it very difficult to off-load any but the most trivial processing to the thread processor. In addition, the set of operations provided by network transactions is not powerful enough to construct an effective remote enqueue operation with a single network transaction (Schauser and Scheiman 1995).

7.6

SHARED PHYSICAL ADDRESS SPACE

This section examines a fourth major design style for the communication architecture of scalable multiprocessors—a shared physical address space. This builds directly upon the modest-scale shared memory machines and provides the same communication primitives: loads, stores, and atomic operations on shared memory locations. Many machines have been developed to extend this approach to large-scale systems, including CM*, C.mmp, NYU Ultracomputer, BBN Butterfly, IBM RP3, Denelcor HEP-1, BBN TC2000, and the CRAY T3D (Scott 1996). Most of the early designs employed a dancehall organization, with the interconnect between the memory and the processors, whereas most of the later designs use distributed-memory organization. The communication assist translates bus transactions into network transactions. The network transactions are very specific, since they describe only a predefined set of memory operations, and are interpreted directly by the communication assist at the remote node.

A generic machine organization for a large-scale distributed shared physical address machine is shown in Figure 7.25. The communication assist is best viewed as forming a *pseudo-memory module* and a *pseudo-processor*, integrated into the

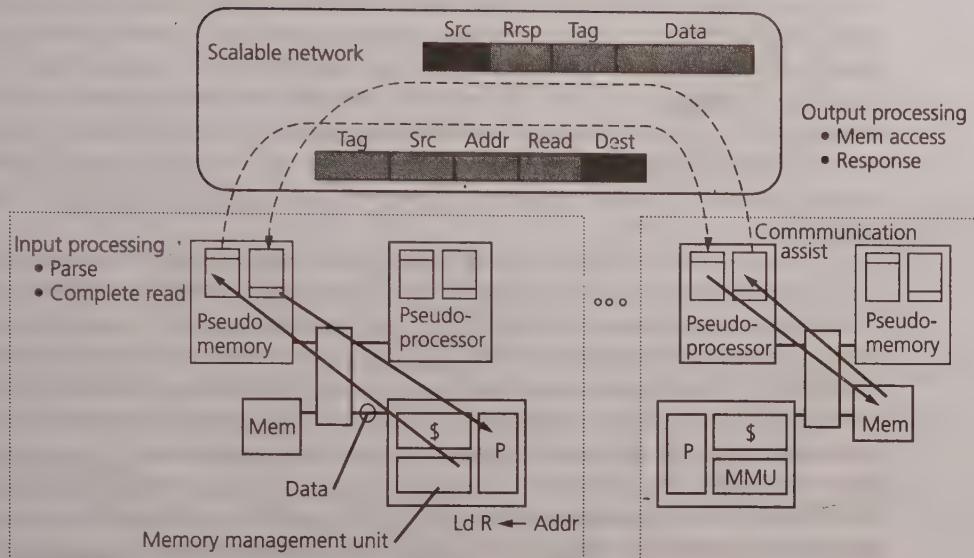


FIGURE 7.25 Shared physical address space machine organization. In scalable shared physical address machines, network transactions are initiated as a result of conventional memory instructions. They have a fixed set of formats, interpreted directly in the communication assist hardware. The operations are request-response, and most systems provide two distinct networks to avoid fetch deadlock. The communication architecture must assume the role of pseudo-memory unit on the issuing node and pseudo-processor on the servicing node. The remote memory operation is accepted by the pseudo-memory unit on the issuing node, which carries out request-response transactions with the remote node. The source bus transaction is held open for the duration of the request network transaction, the remote memory access, and the response network transaction. Thus, the communication assist must be able to access the local memory on the remote node even while the processor on that node is stalled in the middle of its own memory operation.

processor-memory connection. Consider, for example, a load instruction executed by the processor on one node. The on-chip memory management unit (MMU) translates the virtual address into a global physical address that is presented to the memory system. If this physical address is local to the issuing node, the memory simply responds with the contents of the desired location. If not, the communication assist must act like a memory module while it accesses the remote location. The pseudo-memory controller accepts the read transaction on the memory bus, extracts the node number from the global physical address, and issues a network transaction to the remote node to access the desired location. Note that at this point the load instruction is stalled between the address phase and the data phase of the memory operation. The remote communication assist receives the network transaction, reads the desired location, and issues a response transaction to the original node. The remote communication assist appears as a pseudo-processor to the memory system on its node when it issues the proxy read to the memory system. An important point

to note is that when the pseudo-processor attempts to access memory on behalf of a remote node, the main processor there may be stalled in the middle of its own remote load instruction. A simple memory bus supporting one outstanding operation is inadequate for the task. Either there must be two independent paths into memory or the bus must support a split-phase operation with unordered completion. Eventually, the response transaction will arrive at the originating pseudo-memory controller. It will complete the memory read operation just as if it were a (slow) memory module.

A key issue, which we will examine deeply in the next chapter, is the cachability of the shared memory locations. In most modern microprocessors, the cachability of an address is determined by a field in the page table entry for the containing page, which is extracted from the TLB when the location is accessed. In this discussion, it is important to distinguish two orthogonal concepts. An address may be either private to a process or shared among processes, and it may be either physically local to a processor or physically remote. Clearly, addresses that are private to a process and physically local to the processor on which that process executes should be cached. This requires no special hardware support. Private data that is physically remote can also be cached, although this requires that the communication assists support cache block transactions rather than just single words. No processor change is required to cache remote blocks since remote memory accesses appear identical to local memory accesses, only slower; however, coherence is not an issue as long as the process stays put since no other process accesses the private data. If physically local and logically shared data is cached locally, then accesses on behalf of remote nodes performed by the pseudo-processor must be cache coherent. If local shared data is cached only in write-through mode, this only requires that the pseudo-processor invalidate cached data when it performs writes to memory on behalf of remote nodes. To cache shared data as write back, the pseudo-processor needs to be able to cause data to be flushed out of the cache. The most natural solution is to integrate the pseudo-processor on a cache-coherent memory bus, but the bus must also be split phase, with some number of outstanding transactions reserved for the pseudo-processor. The final option is to cache shared, remote data. The hardware support for accessing, transferring, and placing a remote block in the local cache is completely covered by the preceding options. The new issue is keeping the possibly numerous copies of the block in various caches coherent. We must also deal with the consistency model for such a distributed shared memory with replication. These issues require substantially greater design consideration, and we devote the next two chapters to addressing them. It is clearly attractive from a performance viewpoint to cache shared, remote data that is for the most part accessed locally.

7.6.1 Case Study: CRAY T3D

The CRAY T3D provides a concrete example of a shared global physical address design. The design follows the basic outline of Figure 7.13, with a pseudo-memory controller and pseudo-processor providing remote memory access via a scalable network supporting independent delivery of request and response transactions. There

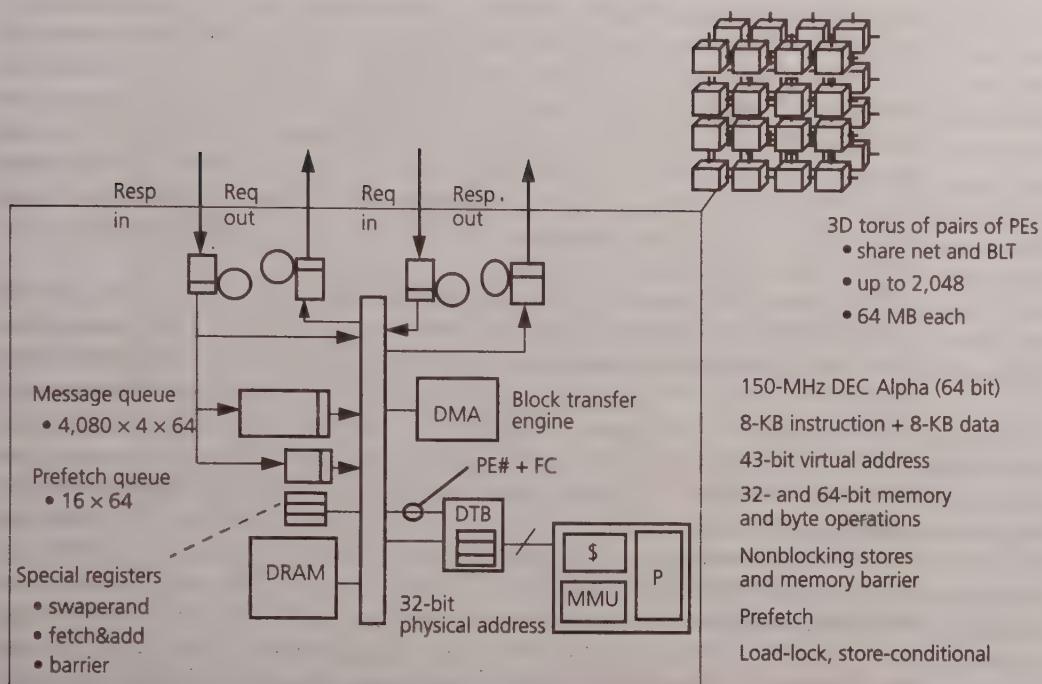


FIGURE 7.26 CRAY T3D machine organization. Each node contains an elaborate communication assist, which includes the pseudo-memory and pseudo-processor functions required for a shared physical address space. In addition, a set of external segment registers (the DTB) is provided to extend the machine's limited physical address space. A prefetch queue supports latency hiding through explicit read ahead. A message queue supports events associated with message-passing models. A DMA unit provides block transfer capability, and special pointwise and global synchronization operations are supported. The machine is organized as a 3D torus, as the name might suggest.

are seven specific network transaction formats, which are interpreted directly in hardware. However, the design extends the basic shared physical address approach in several significant ways. The T3D system is intended to scale to 2,048 nodes, each with a 150-MHz dual-issue DEC Alpha 21064 microprocessor and up to 64 MB of memory, as illustrated in Figure 7.26. The DEC Alpha architecture is intended to be used as a building block for parallel architectures (Digital Equipment Corporation 1992), and several aspects of the 21064 strongly influenced the T3D design. In this case study, we first look at salient aspects of the microprocessor itself and the local memory system. Then we will discuss the assists constructed around the basic processor to provide a shared physical address space, latency tolerance, block transfer, synchronization, and fast message passing. The CRAY designers sometimes refer to this as the “shell” of support circuitry around a conventional microprocessor that embodies the parallel processing capability.

The Alpha 21064 has 8-KB on-chip instruction and data caches as well as support for an external L₂ cache. In the T3D design, the L₂ cache is eliminated in order to

reduce the time to access main memory. The processor stalls for the duration of a cache miss, so reducing the miss latency directly increases the delivered bandwidth. The CRAY design is biased toward access patterns typical of vector codes, which scan through large regions of memory. The measured access time of a load that misses to memory is 155 ns (23 cycles) on the T3D, compared to 300 ns (45 cycles) on a DEC Alpha workstation at the same clock rate with a 512-KB L₂ cache (Arpac et al. 1995). (The CRAY T3D access time increases to 255 ns if the access is off page within the DRAM.) These measurements are very useful in calibrating the performance of the global memory accesses.

The Alpha 21064 provides a 43-bit virtual address space in accordance with the Alpha architecture; however, the physical address space is only 32 bits in size. Since the virtual-to-physical translation occurs on chip, only the physical address is presented to the memory system and communication assist. A fully populated system of 2,048 nodes with 64 MB each would require 37 bits of global physical address space. To enlarge the physical address space of each node, the T3D provides an external register set, called the DTB Annex, which uses 5 bits of the physical address to select a register containing a 21-bit node number; this is concatenated with a 27-bit local physical address to form a full global physical address.⁸ The annex registers also contain an additional field specifying the type of access, for example, cached or uncached. Annex register 0 always refers to the local node. The Alpha load-lock and store-conditional instructions are used in the T3D to read and write the annex registers. Updating an annex register takes 23 cycles, just like an off-chip memory access, and can be followed immediately by a load or store instruction that uses the annex register.

A read or write of a location in the global address space is accomplished by a short sequence of instructions. First, the processor number part of the global virtual address is extracted and stored into an annex register. Then, a temporary virtual address is constructed so that the upper bits specify this annex register and the lower bits specify the address on that node. Finally, a load or store instruction is issued on this temporary virtual address. The load operation takes 610 ns (91 cycles), not including the annex setup and address manipulation. (This number increases by 100 ns [15 cycles] if the remote DRAM access is off page. In addition, if a cache block is brought over, it increases to 785–885 ns.)

Remember, the virtual-to-physical translation occurs in the processor issuing the load. The page tables are set up so that the annex register number is simply carried through from the virtual address to the physical address. So that the resulting physical address will make sense on the remote node, the physical placement of all processes in a parallel program is identical (paging is not supported). In addition, care is exercised to ensure that all processes in a parallel program have extended their heap to the same length.

8. This situation is not altogether unusual. The C.mmp employed a similar trick to overcome the limited addressing capability of the LSI-11 building block. The problems that arose from this led to a famous quote, attributed variously to Gordon Bell or Bill Wulf, that the only flaw in an architecture that is hard to overcome is too small an address space.

The Alpha 21064 provides only nonblocking stores. Execution is allowed to proceed after a store instruction without waiting for the store to complete. Writes are buffered by a write buffer, which is four deep, and in each entry up to 32 bytes of write data can be merged. Several store instructions can be outstanding. A “memory barrier” instruction is provided to ensure that writes have completed before further execution commences. The Alpha nonblocking store allows remote stores to be overlapped so that high bandwidth can be achieved in writes to remote memory locations. Remote writes of up to a full cache block can issue from the write buffer every 250 ns, providing up to 120 MB/s of transfer bandwidth from the local cache to remote memory. A single blocking remote write involves a sequence to issue the store, push the store out of the write buffer with a memory barrier operation, and then wait on a completion flag provided by the network interface. This requires 900 ns, plus the annex setup and address arithmetic.

The Alpha provides a special prefetch instruction, intended to encourage the memory system to move important data closer to the processor. This is used in the T3D to hide the remote read latency. An off-chip prefetch queue of 16 words is provided. A prefetch causes a word to be read from memory and deposited into the queue. Reading from the queue pops the word at its head. The prefetch issue instruction is treated like a store and takes only a few cycles. The pop operation takes the 23 cycles typical of an off-chip access. If eight words are prefetched and then popped, the network latency is completely hidden, and the effective latency of each word is less than 300 ns.

The T3D also provides a bulk transfer engine, which can move blocks or regular strided data between the local node and a remote node in either direction. Reading from a remote node, the block transfer bandwidth peaks at 140 MB/s; writing to a remote node, it peaks at 90 MB/s. However, use of the block transfer engine requires a kernel trap to provide the virtual-to-physical translation. Thus, the prefetch queue provides better performance for transfers of up to 64 KB of data, and nonblocking stores are faster for any length. The primary advantage of the bulk transfer engine is the ability to overlap communication and computation. This capability is limited to some extent since the processor and the bulk transfer engine compete for the same memory bandwidth.

The T3D communication assist also provides special support for synchronization. First, there is a dedicated network to support global-or and global-and operations, used primarily for barriers. This allows processors to raise a flag indicating that they have reached the barrier, to continue executing, and then to wait for all to enter before leaving. Each node also has a set of external synchronization registers to support atomic swap and fetch&inc. There is also a user-level message queue, which will either cause a message to be enqueued or a thread to be invoked on a remote node. Unfortunately, either of these actions involves a remote kernel trap, so the two operations take 25 μ s and 70 μ s, respectively. In comparison, building a queue in memory using the fetch&inc operation allows a four-word message to be enqueued in 3 μ s and dequeued in 1.5 μ s.

7.6.2 Case Study: CRAY T3E

The CRAY T3E (Scott 1996) follow-on to the CRAY T3D provides an illuminating snapshot of the trade-offs in large-scale system design. The two driving forces in the design were the need to provide a more powerful, more contemporary processor in the node and to simplify the shell. The CRAY T3D has many complicated mechanisms for supporting similar functions, each with unique advantages and disadvantages. The T3E uses the 300-MHz, quad-issue Alpha 21164 processor with a sizable (96 KB) second-level on-chip cache. Since the L₂ cache is on chip, eliminating it is not an option as on the T3D. However, the T3E forgoes the board-level tertiary cache typically found in Alpha 21164-based workstations. The various remote access mechanisms are unified into a single external register concept. In addition, remote memory accesses are performed using virtual addresses that are translated to physical addresses by the remote communication assist.

A user process has access to a set of 512 E-registers of 64 bits each. The processor can read and write contents of the E-registers using conventional load and store instructions to a special region of the memory space. Operations are also provided to get data from global memory into an E-register, to put data from an E-register into global memory, and to perform atomic read-modify-writes between E-registers and global memory. Loading remote data into an E-register involves three steps. First, the processor portion of the global virtual address is constructed in an E-address register. Second, the get command is issued via a store to a special region of memory. A field of the address used in the command store specifies the get operation, and another field specifies the destination data E-register. The command-store data specifies an offset to be used relative to the address E-register. The command store has the side effect of causing the remote read to be performed and the destination E-register to be loaded with the data. Finally, the data is read into the processor via a load to the data E-register. The process for a remote put is similar, except that the store data is placed in the data E-register, which is specified in the put command store. This approach of causing loads and stores to data registers as a side effect of operations on address registers goes all the way back to the CDC-6600 (Thornton 1964), although it seems to have been largely forgotten in the meanwhile.

The utility of the prefetch queue is provided in E-registers by associating a full-empty bit with each E-register. A series of gets can be issued, and each one sets the associated destination E-register to empty. When a get completes, the register is set to full. If the processor attempts to load from an empty E-register, the memory operation is stalled until the get completes. The utility of the block transfer engine is provided by allowing vectors of four or eight words to be transferred through E-registers in a single operation. This has the added advantage of providing a means of efficient gather operations.

The improvements in the T3E greatly simplify code generation for the machine and offer several performance advantages; however, these are not at all uniform. The computational performance is significantly higher on the T3D due to the faster processor and larger on-chip cache. On the other hand, the remote read latency is more than twice that of the T3D, increasing from 600 ns to roughly 1500 ns. The increase

is due to the L₂ cache miss penalty and the remote address translation. The remote write latency is essentially the same in the two machines. The prefetch cost is improved by roughly a factor of two, obtaining a rate of one word read every 130 ns. Each of the memory modules can service a read every 67 ns. Nonblocking writes have essentially the same performance on the two machines. The block transfer capability of the T3E is far superior to the T3D. A bandwidth of greater than 300 MB/s is obtained without the large start-up cost of the block transfer engine. The bulk write bandwidth is greater than 300 MB/s, three times the T3D.

7.6.3

Summary

Wide variation exists in the degree of hardware interpretation of network transactions in modern large-scale parallel machines. These variations result in a wide range in the overhead experienced by the compute processor in performing communication operations as well as in the latency added to that of the actual network by the communication assist. By restricting the set of transactions, specializing the communication assist to the task of interpreting these transactions, and tightly integrating the communication assist with the memory system of the node, the overhead and latency can be reduced substantially. Hardware specialization can also provide the concurrency needed within the assist to handle several simultaneous streams of events with high bandwidth.

7.7

CLUSTERS AND NETWORKS OF WORKSTATIONS

Along with the use of commodity microprocessors, memory devices, and even workstation operating systems in modern large-scale parallel machines, scalable communication networks similar to those used in parallel machines have become available for use in a limited local area network setting. This naturally raises the question, To what extent are networks of workstations (NOWs) and parallel machines converging? Before entertaining this question, a little background is in order.

Traditionally, collections of complete computers with dedicated interconnects, often called *clusters*, have been used to serve multiprogramming workloads and to provide improved availability (Kronenberg, Levy, and Strecker 1986; Pfister et al. 1985). In multiprogramming clusters, a single front-end machine usually acts as an intermediary between a collection of compute servers and a large number of users at terminals or remote machines. The front end tracks the load on the cluster nodes and schedules tasks onto the most lightly loaded nodes. Typically, all the machines in the cluster are set up to function identically; they have the same instruction set, the same operating systems, and the same file system access. In older systems, such as the Vax VMS cluster (Kronenberg, Levy, and Strecker 1986), this was achieved by connecting each of the machines to a common set of disks. More recently, this *single-system image* is usually achieved by mounting common file systems over the network. By sharing a pool of functionally equivalent machines, better utilization can be achieved on a large number of independent jobs.

Availability clusters seek to minimize downtime of large critical systems, such as important on-line databases and transaction processing systems. Structurally, they have much in common with multiprogramming clusters. A very common scenario is to use a pair of SMPs running identical copies of a database system with a shared set of disks. Should the primary system fail due to a hardware or software problem, operation rapidly “fails over” to the secondary system. The actual interconnect that provides the shared disk capability can be dual access to the disks or some kind of dedicated network.

Increasingly, clusters are being used as parallel machines, often called *networks of workstations* (NOWs). A major influence on clusters has been the rise of popular public domain software, such as Condor (Litzkow, Livny, and Mutka 1988) and PVM (Geist et al. 1994), that allows users to farm jobs over a collection of machines or to run a parallel program on a number of machines connected by an arbitrary local area or even wide area network. Although the communication performance capability is quite small, typical latencies are a millisecond or more for even small transfers, and the aggregate bandwidth is often less than 1 MB/s, these tools provide an inexpensive vehicle for a class of problems with a very high ratio of computation to communication.

The technology breakthrough that presents the potential of clusters taking on an important role in large-scale parallel computing is a scalable, low-latency interconnect, similar in quality to that available in parallel machines but deployed like a local area network. Several potential candidate networks have evolved from three basic directions. Local area networks have traditionally been either a shared bus (e.g., Ethernet) or a ring (e.g., token ring and FDDI) with fixed aggregate bandwidth, or a dedicated point-to-point connection (e.g., HPPI). In order to provide scalable bandwidth to support a large number of fast machines, there has been a strong push toward switch-based local area networks (e.g., HPPI switches, FDDI switches [Lukowsky and Polit 1997], and FiberChannels). A significant development is the widespread adoption of the ATM (asynchronous transfer mode) standard, developed by the telecommunications industry, as a switched LAN. Several companies offer ATM switches with up to 16 ports at 155-Mb/s (19.4-MB/s) link bandwidth. These can be cascaded to form a larger network. Under ATM, a variable-length message is transferred on a preassigned route, called “virtual circuit,” as a sequence of 53-byte cells (48 bytes of data and 5 bytes of routing information). We will look at these networking technologies in more detail in Chapter 10. In terms of the model developed in Section 7.1, current ATM switches typically have a routing delay of about 10 μ s in an unloaded network, although some are much higher. A second major standardization effort is represented by SCI (scalable coherent interconnect), which includes a physical layer standard and a particular distributed cache coherency strategy. A third is the widespread use of switching for fast Ethernet and the standardization of gigabit Ethernet.

A strong trend has also emerged to evolve the proprietary networks used within MPP systems into a form that can be used to connect a large number of independent workstations or PCs over a sizable area. Examples of this include ServerNet from Tandem Corporation (Horst 1995) and Myrinet (Boden et al. 1995). The Myrinet

switch provides eight ports at 160 MB/s each, which can be cascaded in regular or irregular topologies to form a large network. It transfers variable-length packets with a routing delay of about 350 ns per hop. Link-level flow control is used to avoid dropping packets in the presence of contention.

As with more tightly integrated parallel machines, the hardware primitives in emerging NOWs and clusters remain an open issue and subject to much debate. Conventional TCP/IP communication abstractions over these advanced networks exhibit large overhead (a millisecond or more) (Keeton, Anderson, and Patterson 1995), in many cases larger than that of common Ethernet. A very fast processor is required to move even 20 MB/s using TCP/IP. However, the bandwidth does scale with the number of processors, at least if there is little contention. Several more efficient communication abstractions have been proposed, including Active Messages (Anderson, Culler, and Patterson 1995; von Eicken et al. 1992; von Eicken, Basu, and Buch 1995) and reflective memory (Gillett 1996; Gillett and Kaufmann 1997). Active Messages provide user-level network transactions, as discussed previously. Reflective memory allows writes to special regions of memory to appear as writes into regions on remote processors; there is no ability to read remote data, however. Supporting a true shared physical address space in the presence of potential unreliability (i.e., node failures and network failures) remains an open question. An intermediate strategy is to view the logical network connecting a collection of communicating processes as a fully connected group of queues. Each process has a communication endpoint consisting of a send queue, a receive queue, and a certain amount of state information, such as whether notifications should be delivered on message arrival. Each process can deliver a message to any of the receive queues by depositing it in its send queue with an appropriate destination identifier. This approach is being standardized by an industry consortium—led by Intel, Microsoft, and Compaq—as the Virtual Interface Architecture (Dunning et al. 1998), based on several research efforts including Berkeley NOW, Cornell UNET, Illinois FM, and Princeton SHRIMP.

The hardware support for the communication assists and the interpretation of the network transactions within clusters and NOWs span most of the range of design points discussed in the preceding sections. However, since the network plugs into existing machines rather than being integrated into the system at the board or chip level, typically it must interface at an I/O bus rather than at the memory bus or closer to the processor. In this area too, there is considerable innovation. Several relatively fast I/O buses have been developed that maintain cache coherency, the most notable being PCI. Experimental efforts have integrated the network through the graphics bus (Martin 1994; Banks and Prudence 1993) or the SIMM attachment (Minnich, Burns, and Hady 1995).

An important technological force further driving the advancement of clusters is the availability of relatively inexpensive SMP building blocks. For example, clustering a few tens of Pentium Pro “quad pack” commodity servers yields a fairly large parallel machine with very little effort. At the high end, most of the very large machines are being constructed as highly optimized clusters of the vendor’s largest commercially available SMP node. For example, in the 1997–1998 window of

machines purchased by the Department of Energy as part of the Accelerated Strategic Computing Initiative, the Intel machine is built as 4,536 dual-processor Pentium Pros. The IBM machine is to be 512 four-way PowerPC 604s, upgraded to eight-way PowerPC 630s. The SGI/CRAY machine is initially sixteen 32-way Origins interconnected with many HPPI 6400 links, eventually to be integrated into larger cache-coherent units, as described in Chapter 8.

7.7.1 Case Study: Myrinet SBUS Lanai

A representative example of an emerging NOW is illustrated in Figure 7.27. A collection of UltraSparc workstations is integrated using Myricom's Myrinet scalable network via an intelligent network interface card (NIC). Let us start with the basic hardware operation and work upward. The network illustrates what is becoming known as a *system area network* (SAN) as opposed to a tightly packaged parallel machine network or a widely dispersed local area network (LAN). The links are parallel copper twisted pairs (18 bits wide) and can be a few tens of feet long, depending on link speed and cable type. The communication assist follows the dedicated communication processor approach similar to the Meiko CS-2 and the IBM SP-2. The NIC contains a small embedded "Lanai" processor to control message flow between the host and the network. A key difference in the cluster design is that the NIC contains a sizable amount of SRAM storage. All message data is staged through NIC memory between the host and the network. This memory is also used for Lanai instruction and data storage. There are three DMA engines on the NIC, one for network input, one for network output, and one for transfers between the host and the NIC memory. The host processor can read and write NIC memory using conventional loads and stores to proper regions of the address space, that is, through programmed I/O. The NIC processor uses DMA operations to access host memory. The kernel establishes regions of host memory that are accessible to the NIC. For short transfers, it is most efficient for the host to move the data directly into and out of the NIC, whereas for long transfers it is better for the host to write addresses into the NIC memory and have the NIC pick up these addresses and use them to set up DMA transfers. The Lanai processor can read and write the network FIFOs, or it can drive them by DMA operations from or to NIC memory.

The firmware program executing on the Lanai primarily manages the flow of data by orchestrating DMA transfers in response to commands written to it by the host and packet arrivals from the network. Typically, a command is written into NIC memory, where it is picked up by the NIC processor. The NIC transfers data, as required, from the host and pushes it into the network. The Myricom network uses source-based routing, so the header of the packet includes a simple routing directive for each network switch along the path to the destination. The destination NIC receives the packet into NIC memory. It can then inspect the information in the transaction and process it as desired to support the communication abstraction.

The NIC is implemented as four basic components: a bus interface; a link interface; SRAM; and the Lanai chip, which contains the processor, DMA engines, and link FIFOs. The link interface converts from on-board CMOS signals to long-line

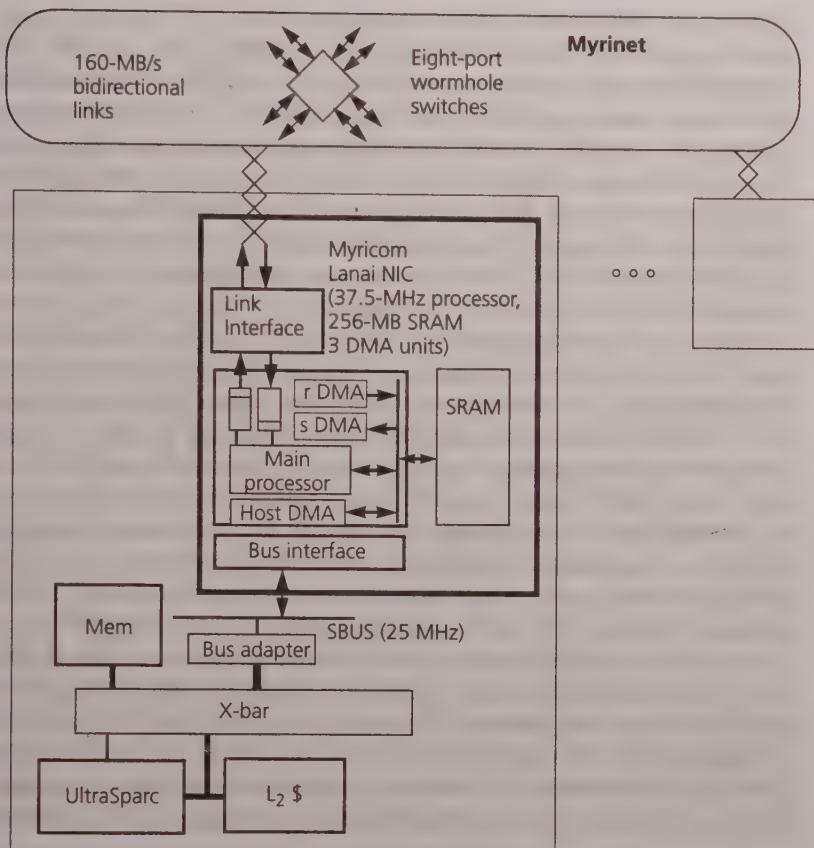


FIGURE 7.27 NOW organization using Myrinet and dedicated message processing with an embedded processor. Although the nodes of a cluster are complete conventional computers, a sophisticated communication assist can be provided within the interface to a scalable, low-latency network. Typically, the network interface is attached to a conventional I/O bus, but increasingly vendors are providing means of tighter integration with the node architecture. In many cases, the communication assist provides dedicated processing of network transactions.

differential signaling over twisted pairs. A critical aspect of the design is the bandwidth to the NIC memory. The three DMA engines and the processor share an internal bus, implemented within the Lanai chip. The network DMA engines can demand 320 MB/s, whereas the host DMA can demand short bursts of 100 MB/s on an SBUS or long bursts of 133 MB/s on a PCI bus. The design goal for the firmware is to keep all three DMA engines active simultaneously; however, this is a little tricky because once it starts the DMA engines, its available bandwidth and hence its execution rate are reduced considerably by competition of the SRAM.

Typically, the NIC memory is logically divided into a collection of functionally distinct regions, including instruction storage, internal data structures, message

queues, and data transfer buffers. Each page of the NIC memory space is independently mapped by the host virtual memory system. Thus, the NIC processor code and data space can be made accessible only to the kernel. The remaining communication space can be partitioned into several disjoint communication regions. By controlling the mapping of these communication regions, several user processes can have *communication endpoints* that are resident on the NIC, each containing message queues and associated data transfer areas (Chun, Mainwaring, and Culler 1998). In addition, a collection of host memory frames can be mapped into the I/O space accessible from the NIC. Thus, several user processes can have the ability to write messages into the NIC and read messages directly from the NIC, or to write message descriptors containing pointers to data that is to be DMA transferred through the card. These communication endpoints can be managed much like conventional virtual memory so that writing into an endpoint causes it to be made resident on the NIC. The NIC firmware is responsible for multiplexing messages from the collection of resident endpoints onto the actual network link. It detects when a message has been written into a send queue and forms a packet by translating the user's destination address into a route through the network to the destination node and an identifier of the destination endpoint on that node. In addition, it can place a source identifier in the header, which can be checked at the destination. The NIC firmware inspects the header for each incoming packet. If it is destined for a resident endpoint, then it can be deposited directly in the associated receive buffer; optionally, for a bulk data transfer, if the destination region is mapped, the data can be DMA transferred into host memory. If these conditions are not met, or if the message is corrupted or the protection check violated, the packet can be NACKed to the source. The driver that manages the mapping of endpoints and data buffer spaces is notified to cause the situation to be remedied before the message is successfully retried.

7.7.2 Case Study: PCI Memory Channel

A second important representative cluster communication assist design is the Memory Channel (Gillett 1996) developed by Digital Equipment Corporation, based on the Encore reflective memory and on research efforts in virtual memory-mapped communication (Blumrich et al. 1994; Dubnicki et al. 1996). This approach seeks to provide a limited form of a shared physical address space without fully integrating the pseudo-memory device and pseudo-processor into the memory system of the node. It also preserves some of the autonomy and independent failure characteristics of clusters. As with other clusters, the communication assist is contained in a network interface card that is inserted into a conventional node on an extension bus, in this case the PCI bus.

The basic idea behind reflective memory is to establish a connection between a region of the address space of one process—a transmit region—and a receive region in another, as indicated by Figure 7.28. Data written to a transmit region by the source is “reflected” into the receive region of the destination. Usually, a collection of processes will have a fully connected set of transmit-receive region pairs. The transmit regions on a node are allocated from a portion of the physical address space that

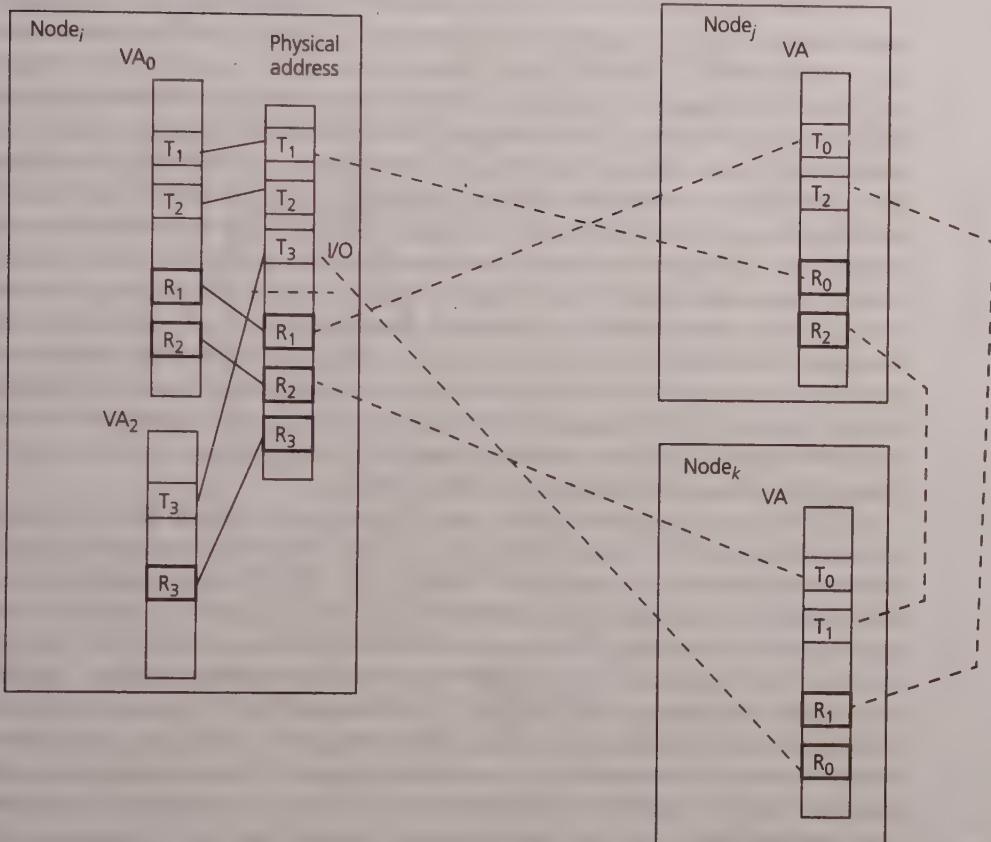


FIGURE 7.28 Typical reflective memory address space organization. Transmit regions of the virtual address space of the processes on a node (VA) are mapped to regions in the physical address space associated with the NIC. Receive regions are pinned in memory, and mappings within the NIC are established so that it can DMA-transfer the associated data to host memory. Here, Node_i has two communicating processes, one of which has established reflective memory connections with processes on two other nodes. Writes to a transmit region generate memory transactions against the NIC. It accepts the write data, builds a packet with a header identifying the receive page and offset, and routes the packet to the destination node. The NIC, upon accepting a packet from the network, inspects the header and DMA-transfers the data into the corresponding receive page. Optionally, packet arrival may generate an interrupt. In general, the user process scans receive regions for relevant updates. To support message passing, the receive pages contain message queues.

is mapped to the NIC. The receive regions are locked down in memory via special kernel calls, and the NIC is configured so that it can DMA-transfer data into them. In addition, the source and destination processes must establish a connection between the source transmit region and the destination receive region. Typically, this is done by associating a key with the connection and binding each region to the key. The regions are an integral number of pages.

The DEC Memory Channel is a PCI-based NIC, typically placed in an Alpha-based SMP, described by Figure 7.29 (Gillett 1996). In addition to the usual transmit and receive FIFOs, it contains a page control table (PCT), a receive-DMA engine, and transmit and receive controllers. A block of data is written to the transmit region with a sequence of stores. The Alpha write buffer will attempt to merge updates to a cache block, so the transmit controller will typically see a cache block write operation. The upper portion of the address given to the controller is the frame number, which is used to index into the PCT to obtain a descriptor for the associated receive region (i.e., the destination node or route to the node, the receive frame number, and the associated control bits). This information, along with the source information, is placed into the header of the packet that is delivered to the destination node through the network. The receive controller extracts the receive frame number and uses it to index into the PCT. After checking the packet integrity and verifying the source, the data is DMA-transferred into memory. The receive regions can be cachable host memory since the transfers across the memory bus are cache coherent. If required, an interrupt is raised upon write completion.

As with a shared physical address space, this approach allows data to be transferred between nodes by simply storing the data from the source and loading it at the destination. However, the use of the address space is much more restrictive since data that is to be transferred to a particular node must be placed in a specific transmit page and receive page. This is quite different from the scenario where any process can write to any shared address and any process can read that address. Typically, shared data structures are not placed in the communication regions. Instead, the regions are used as dedicated message buffers. Data is read from a logically shared data structure and transmitted to a process that requires it through a logical memory channel. Thus, the communication abstraction is really one of memory-based message passing. There is no mechanism to read a remote location; a process can only read the data that has been written to it. To better support the “write by one, read by any” aspect of shared memory, the DEC Memory Channel allows transmit regions to multicast to a group of receive regions, including a loopback region on the source node. To ease construction of distributed data structures—in particular, a distributed lock manager—the FIFO order is preserved across operations from a transmit region. Since reflective memory builds upon, rather than extends, the node memory system, the write to a transmit region finishes as soon as the NIC accepts the data. To determine that the write has actually occurred, the host checks a status register in the NIC.

The raw Memory Channel interface obtains a one-way communication latency of 2.9 μ s and a transfer bandwidth of 64 MB/s between two 300-MHz DEC Alpha-Servers (Lawton et al. 1996). The one-way latency for a small MPI message over Memory Channel is about 7 μ s, and a maximum bandwidth of 61 MB/s is achieved on long messages. Using the TruCluster Memory Channel software (Cardoza, Glover, and Snaman 1996), acquiring and releasing an uncontended spin-lock takes approximately 130 μ s and 120 μ s, respectively.

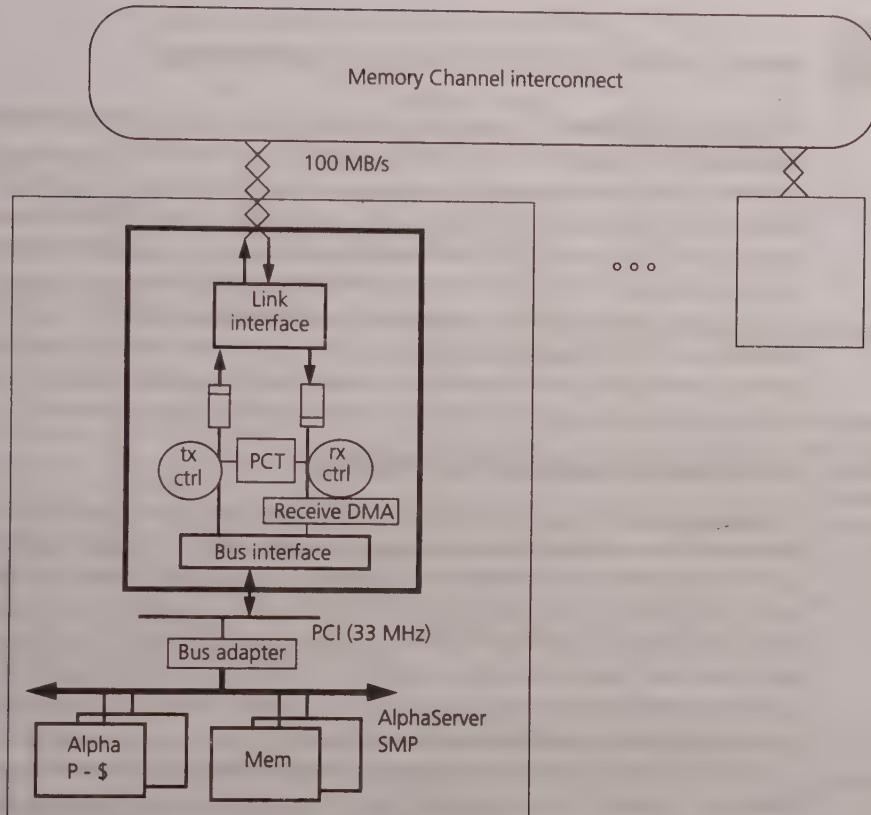


FIGURE 7.29 DEC Memory Channel hardware organization. A Memory Channel cluster consists of a collection of AlphaServer SMPs with PCI Memory Channel adapters. The adapter contains a page control table, which maps local transmit regions to remote receive regions and local receive regions to locked-down physical page frames. The transmit controller (tx ctrl) accepts PCI write transactions, constructs a packet header using the store address and PCT entry contents, and deposits a packet containing the write data into the transmit FIFO. The receive controller (rx ctrl) checks the CRC and parses the header of an incoming packet before initiating a DMA transfer of the packet data into host memory. Optionally, an interrupt may be generated. In the initial offering, the Memory Channel interconnect is a shared 100-MB/s bus, but this is intended to be replaced by a switch. Each of the nodes is typically a sizable SMP AlphaServer.

The Princeton SHRIMP designs (Blumrich et al. 1994; Dubnicki et al. 1996) extend the reflective memory model to better support message passing. They allow the receive offset to be determined by a register at the destination so that successive packets will be queued into the receive region. In addition, a collection of writes can be performed to a transmit region, and then a segment of the region can be transmitted to the receive region.

7.8**IMPLICATIONS FOR PARALLEL SOFTWARE**

Now that we have seen the design spectrum of modern scalable distributed-memory machines, we can solidify our understanding of the impact of these design trade-offs in terms of the communication performance that is delivered to applications. In this section, we will examine communication performance through microbenchmarks at three levels. The first set of microbenchmarks uses a communication abstraction that closely approximates the basic network transaction on a user-to-user basis. The second uses a shared address space and the third the standard MPI message-passing abstraction. In making this comparison, we can see the effects of the different organizations and of the protocols used to realize the communication abstractions.

7.8.1 Network Transaction Performance

Many factors interact to determine the end-to-end latency of the individual network transaction as well as the abstractions built on top of it. When we measure the time per communication operation, we observe the cumulative effect of these interactions. In general, the measured time will be larger than what we would obtain by adding up the time through each of the individual hardware components. As architects, we may want to know how each of the components impacts performance; however, what matters to programs is the cumulative effect, including the subtle interactions that inevitably slow things down.

We will take an empirical approach to determining the communication performance of several of our case study machines. A simple user-level communication abstraction, Active Messages, is used as a basis for this study. We want to measure not only the total message time but the portion of this time that is overhead in our data transfer equation (Equation 1.5) and the portion that is due to occupancy and delay.

The microbenchmark that uses Active Messages is a simple echo test, where the remote processor is continually servicing the network and issuing replies. This eliminates the timing variations that would be observed if the processor was busy doing other work when the request arrived. In addition, since our focus is on the node-to-network interface, we pick processors that are adjacent in the physical network. All measurements are performed by the source processor since many of these machines do not have a global synchronous clock and the “time skew” between processors can easily exceed the scale of an individual message time. To obtain the end-to-end message time, the round-trip time for a request-response transaction is divided by two. However, this one-way message time has three distinct portions, as illustrated by Figure 7.30. When a processor injects a message, it is occupied for a number of cycles as it interfaces with the communication assist. We call this the *send overhead*, as it is time spent that cannot be used for useful computation. Similarly, the destination processor spends a number of cycles extracting or otherwise dealing with the message, called the *receive overhead*. The portion of the total message cost that is not covered by overhead is the *communication latency*. In terms of the communication cost expression developed in Chapter 3, this includes the portions of the transit

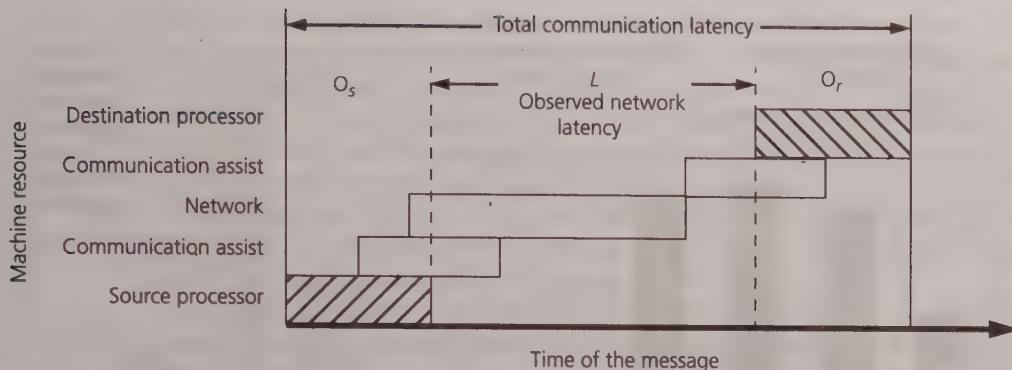


FIGURE 7.30 Breakdown of message time into send overhead, network latency, and receive overhead. This depicts the machine operation associated with our basic data transfer model for communication operations. The source processor spends O_s cycles injecting the message into the communication assist, during which time it can perform no other useful work, and similarly the destination experiences O_r overhead in extracting the message. To actually transfer the information involves the communication assists and network interfaces as well as the links and switches of the network. As seen from the processor, these subcomponents are indistinguishable. It experiences the portion of the transfer time that is not covered by its own overhead as latency that can be overlapped with other useful work. The processor also experiences the maximum message rate, which is specified in terms of the minimum average time between messages, or gap.

delay and occupancy components that are not overlapped with send or receive overhead. It can potentially be masked by other useful work or by processing additional messages, as we will discuss in detail in Chapter 11.

In our microbenchmark, we seek to determine the length of these three portions. The processor cannot distinguish time spent in the communication assists from that spent in the actual links and switches of the interconnect. In fact, the communication assist may begin pushing the message into the network while the source processor is still checking status, but this work will be masked by the overhead.

The left portion of the graph in Figure 7.31 shows a comparison of one-way Active Message time for the Thinking Machines CM-5 (Section 7.4.2), the Intel Paragon (Section 7.5.1), the Meiko CS-2 (Section 7.5.2), a cluster of workstations called the NOW Ultra (Section 7.7.1), and the CRAY T3D (Section 7.6.1). The bars show the total one-way latency of a small (five-word) message divided into three segments, indicating the processing overhead on the sending side (O_s), the processing overhead on the receiving side (O_r), and the remaining communication latency (L). The bars on the right (g) show the time per message for a pipelined sequence of request-response operations. For example, on the Paragon, an individual message has an end-to-end latency of about 10 μ s, but a burst of messages can go at about 7.5 μ s per message, or a rate of $1/7.5 \mu\text{s} = 133,000$ messages per second. Let us examine each of these components in more detail.

The send overhead is nearly uniform over the five designs; however, the factors determining this component are different on each system. On the CM-5, this time is

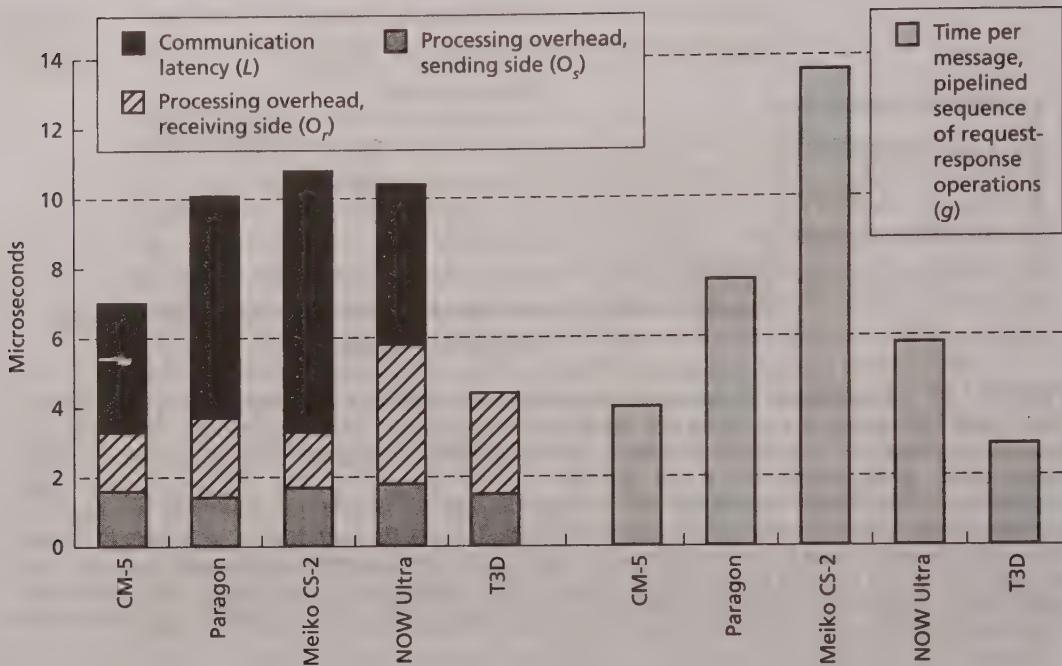


FIGURE 7.31 Performance comparison at the network transaction level via Active Messages. The bars on the left show the overall latency for five machines described in this chapter divided into components of send overhead, receive overhead, and network latency. The latter component is dominated by the communication assist occupancy but includes the network delay. The bars on the right show the time per message, called the gap, for these machines when transferring a series of small messages. This is determined primarily by the communication assist occupancy.

dominated by uncached writes of the data plus the uncached read of the NI status to determine if the message was accepted. The status also indicates whether an incoming message has arrived, which is convenient since the node must receive messages even while it is unable to send. Unfortunately, two status reads are required for the two networks. (A later model of the machine, the CM-500, provided more effective status information and substantially reduced the overhead.) On the Paragon, the send overhead is determined by the time to write the message into the memory buffer shared by the compute processor and communication (or message) processor (CP). The compute processor is able to write the message and set the “message present” flag in the buffer entry with two quad-word stores, unique to the i860, so it is surprising that the overhead is so high. The reason has to do with the inefficiency of the bus-based cache coherence protocol within the node for such a producer-consumer situation discussed earlier. The compute processor has to pull the cache blocks away from the CP before refilling them. In the Meiko CS-2, the message is

built in cached memory, and then a pointer to the message (and command) is enqueued in the NI with a single swap instruction. This instruction is provided in the Sparc instruction set to support synchronization operations. In this context, it provides a way of passing information to the NI and getting status back indicating whether the operation was successful. Unfortunately, the exchange operation is considerably slower than the basic uncached memory operations. In the NOW system, the send overhead is due to a sequence of uncached double-word stores and an uncached load across the I/O bus to NI memory. Surprisingly, these have the same effective cost as the more tightly integrated designs.

An important point revealed by this comparison is that the cost of uncached operations, misses, and synchronization instructions, generally considered to be infrequent events and therefore a low priority for architectural optimization, is critical to communication performance. The time spent in the cache controllers before they allow the transaction to be delivered to the next level of the storage hierarchy dominates even the bus protocol. The comparison of the receive overhead for these designs shows that cache-to-cache transfer from the network processor to the compute processor is more costly than uncached reads from the NI on the memory bus of the CM-5 and CS-2. However, the NOW system is subject to the greater cost of uncached reads over the I/O bus.

Several facets of the machine contribute to the latency component, including the processing time on the communication assist or in the network interface, the time to channel the message onto a link, and the delay through the network. Different facets are dominant in our case study machines. The CM-5 links operate at 20 MB/s (4 bits wide at 40 MHz). Thus, the occupancy of a single wire to transfer a message with 40 bytes of data (the payload), plus an envelope containing route information, CRC, and message type, is nearly 2.5 μ s. Each router adds a delay of roughly 200 ns, and there are at most $2 \log_4 N$ hops. The network interface occupancy is essentially the same as the wire occupancy, as it is a very simple device that spools the packet on or off the wire.

In the Paragon, the latency is dominated by processing in the CP at the source and destination. With 175-MB/s links, the occupancy of a link is only about 300 ns. The routing delay per hop is also quite small; however, in a large machine the total delay may be substantially larger than the link occupancy, as the number of hops can be $2\sqrt{N}$. The dominant factor is the assist (CP) occupancy in writing the message across the memory bus into the NI at the source and reading the message from the destination. These steps account for 4 μ s of the latency. Eliminating the CPs in the communication path on the Paragon decreases the latency substantially (Krishnamurthy et al. 1996). Surprisingly, it does not increase the overhead much; writing and reading the message to and from the NI have essentially the same cost as the cache-to-cache transfers. However, since the NI does not provide sufficient interpretation to enforce protection and to ensure that messages move forward through the network adequately, avoiding the CPs is not a viable option in practice.

The Meiko has a very large latency component. The network accounts for a small fraction of this, as it provides 40-MB/s links and topology similar to the CM-5. The CP is closely coupled with the network on a single chip. The latency is almost

entirely due to accessing system memory from the CP. Recall that the compute processor provides the CP with a pointer to the message. The CP performs DMA operations to pull the message out of memory. At the destination, it writes the message into a shared queue. An unusual property of this machine is that a circuit through the network is held open through the network transaction and an acknowledgment is provided to the source. This mechanism is used to convey to the source CP whether it successfully obtained a lock on the destination incoming message queue. Thus, even though the latency component is large, it is difficult to hide through pipelining communication operations since source and destination CPs are occupied for the duration.

The latency in the NOW system is distributed fairly evenly over the facets of the communication system. The link occupancy is small with 160-MB/s links, and the routing delay is modest with 350 ns per hop. The data is deposited into the NI by the host and accessed directly from the NI. Time is spent on both ends of the transaction to manipulate message queues and to perform DMA operations between NI memory and the network. Thus, the two NIs and the network each contribute about one-third of the 4- μ s latency.

The CRAY T3D provides hardware support for user-level messaging in the form of a per-processor message queue. The capability in the DEC Alpha to extend the instruction set with privileged subroutines, called PAL code, is used. A four-word message is composed in registers, and a PAL call is issued to send the message. It is placed in a user-level message queue at the destination processor; the destination processor is interrupted, and control is returned either to the user application thread, which can poll the queue, or to a specific message handler thread. The send overhead to inject the message is only 0.8 μ s; however, the interrupt has an overhead of 25 μ s and the switch to the message handler has a cost of 33 μ s (Arpacı et al. 1995). A packet can be inserted in the message queue, without interrupts or thread switch, using the fetch&increment registers provided for atomic operations. The fetch&increment to advance the queue pointer and the writing of the message takes 1.5 μ s, whereas dispatching on the message and reading the data via uncached reads takes 2.9 μ s.

The Paragon, Meiko, and NOW machines employ a complete operating system on each node. The systems cooperate using messages to provide a single-system image and to run parallel programs on collections of nodes. The CP is instrumental in multiplexing communication from many user processes onto a shared network and demultiplexing incoming network transactions to the correct destination processes. It also provides flow control and error detection. The MPP systems rely on the physical integrity of a single box to provide highly reliable operation. When a user process fails, the other processes in its parallel program are aborted. When a node crashes, its partition of the machine is rebooted. The more loosely integrated NOW must contend with individual node failures that are a result of hardware errors, software errors, or physical disconnection. As in the MPPs, the operating systems cooperate to control the processes that form a parallel program. When a node fails, the system reconfigures to continue without it.

7.8.2 Shared Address Space Operations

It is useful to compare the communication architectures of the case study machines for shared address space operations: read and write. These operations can easily be built on top of the user-level message abstraction, but this does not exploit the opportunity to optimize for these simple, frequently occurring operations. In addition, on machines where the assist does not provide enough interpretation, the remote processor is involved whenever its memory is accessed. In machines with a dedicated CP, it can potentially service memory requests on behalf of remote nodes without involving the compute processor. On machines supporting a shared physical address, this memory request service is provided directly in hardware.

Figure 7.32 shows the performance of a read of a remote location for the case study machines (Krishnamurthy et al. 1996). The bars on the left show the total read time, broken down into the overhead associated with issuing the read and the latency of the remaining communication and remote processing. The bars on the right show the minimum time between reads in the steady state. For the CM-5, there is no opportunity for optimization since the remote processor must handle the network transaction. In the Paragon, the remote CP performs the read request and replies. The remote processing time is significant because the CP must read the message from the NI, service it, and write a response to the NI. Moreover, the CP must

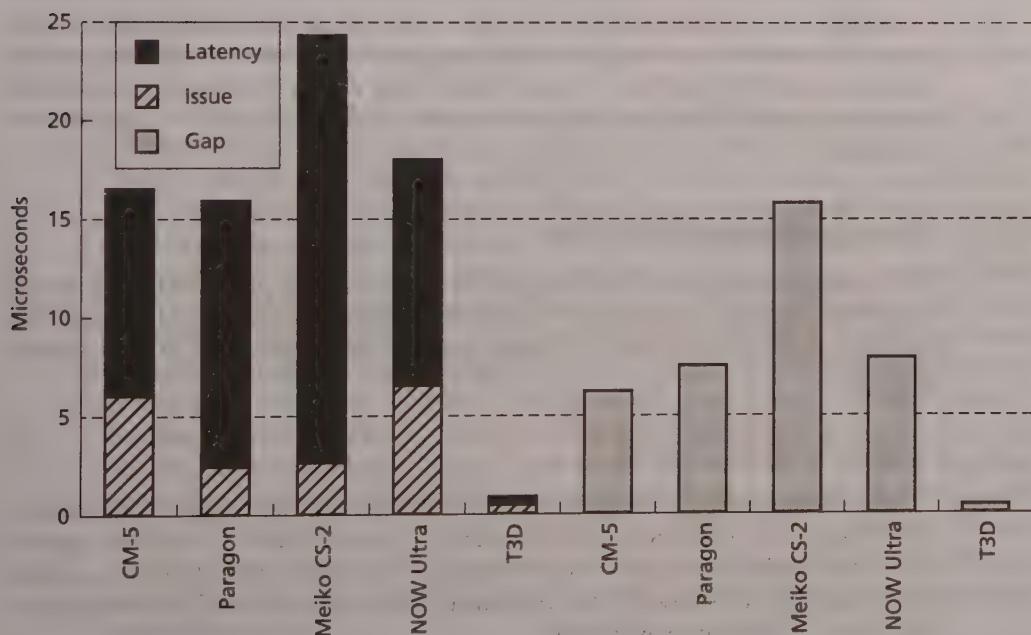


FIGURE 7.32 Performance comparison on shared address read. For the five case study platforms, the bars on the left show the total time to perform a remote read operation and isolate the portion of that time that involves the processor issuing and completing the read. The remainder can be overlapped with useful work, as is discussed in depth in Chapter 11. The bars on the right show the minimum time between successive reads, which is the reciprocal of the maximum rate.

perform a protection check and the virtual-to-physical translation. On the Paragon with the OSF1/AD operating system, the CP runs in kernel mode and operates on physical addresses; thus, it performs the page table lookup for the requested address (which may not be for the currently running process) in software. The Meiko CS-2 provides read and write network transactions, where the source-to-destination circuit in the network is held open until the read response or write acknowledgment is returned. The remote processing is dedicated and uses a hardware page table to perform the virtual-to-physical transaction on the remote node. Thus, the read latency is considerably less than a pair of messages but still substantial. If the remote CP performs the read operation, the latency increases by an additional 8 μ s. The NOW system achieves a small performance advantage by avoiding the use of the remote processor. As in the Paragon, the message processor must perform the protection check and address translation, which are quite slow in the embedded CP. In addition, accessing remote memory from the network interface involves a DMA operation and is costly. The major advantage of dedicated processing of remote memory operations in all of these systems is that the performance of the operation does not depend on the remote compute processor servicing incoming requests from the network.

The T3D shows an order-of-magnitude improvement available through dedicated hardware support for a shared address space. Given that the remote reads and writes are implemented through add-on hardware, there is a cost of 400 ns to issue the operation. The transmission of the request, the remote service, and the transmission of the reply take an additional 450 ns. For a sequence of reads, performance can be improved further by utilizing the hardware prefetch queue. Issuing the prefetch instruction and the pop from the queue takes about 200 ns. The latency is amortized over a sequence of such prefetches and is almost completely hidden if eight or more are issued.

7.8.3 Message-Passing Operations

Let us also take a look at the measured message-passing performance of several large-scale machines. As discussed earlier, the most common performance model for message-passing operations is a linear model for the overall time to send n bytes, given by

$$T(n) = T_0 + \frac{n}{B} \quad (7.2)$$

The start-up cost, T_0 , is logically the time to send zero bytes, and B is the asymptotic bandwidth. The delivered data bandwidth is simply $BW(n) = n/T(n)$. Equivalently, the transfer time can be characterized by two parameters, r_∞ and $n_{1/2}$, which are the asymptotic bandwidth and the transfer size at which half of this bandwidth is obtained (i.e., the half-power point).

The start-up cost reflects the time to carry out the protocol as well as whatever buffer management and matching is required to set up the data transfer. The asymptotic bandwidth reflects the rate at which data can be pumped through the system from end to end.

Table 7.1 Message-Passing Start-Up Costs and Asymptotic Bandwidths

Machine	Year	T_0 (μs)	Maximum Bandwidth (MB/s)	Cycles per T_0	MFLOPS per Processor	Floating- Point Operations per T_0	$n_{1/2}$
iPSC/2	1988	700	.2.7	5,600	1	700	1,400
nCUBE/2	1990	160	2	3,000	2	300	300
iPSC/860	1991	160	2	6,400	40	6,400	320
CM-5	1992	95	8	3,135	20	1,900	760
SP-1	1993	50	25	2,500	100	5,000	1,250
Meiko CS-2*	1993	83	43	7,470	24	1,992	3,560
Paragon	1994	30	175	1,500	50	1,500	7,240
SP-2	1994	35	40	3,960	200	7,000	2,400
CRAY T3D (PVM)	1994	21	27	3,150	94	1,974	1,502
NOW	1996	16	38	2,672	180	2,880	4,200
SGI Power-Challenge	1995	10	64	900	308	3,080	800
Sun E6000	1996	11	160	1,760	180	1,980	2,100

Although this model is easy to understand and useful as a programming guide, it presents a couple of methodological difficulties for architectural evaluation. As with network transactions, the total message time is difficult to measure unless a global clock is available since the send is performed on one processor and the receive on another. This problem is commonly avoided by measuring the time for an echo test—one processor sends the data and then waits until it receives a message. However, this approach only yields a reliable measurement if the receive is posted before the message arrives; otherwise, it is measuring the time for the remote node to get around to issuing the receive. If the receive is preposted, then the test does not measure the full time of the receive and does not reflect the costs of buffering data since the match succeeds. Finally, the measured times are not linear in the message size, so fitting a line to the data yields a T_0 parameter that has little to do with the actual start-up cost. Usually, there is a flat region for small values of n , so the start-up cost obtained through the fit will be smaller than the time for a 0-byte message, perhaps even negative. These methodological concerns are not a problem for older machines, which had very large start-up costs and simple, software-based message-passing implementations.

Table 7.1 shows the start-up cost and asymptotic bandwidth reported for commercial message-passing libraries on several important large parallel machines over a period of time. (Also shown in the table are two small-scale SMPs, discussed in Chapter 6.) The start-up cost T_0 has dropped by an order of magnitude in less than a

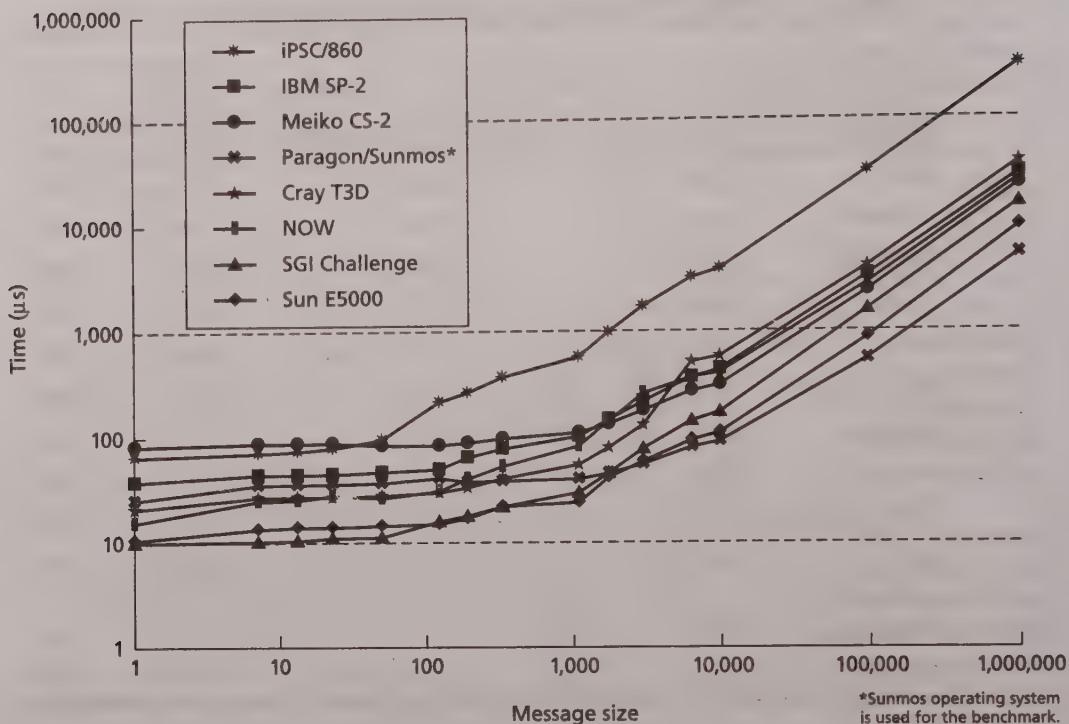


FIGURE 7.33 Time for message-passing operation versus message size. Message-passing implementations exhibit nearly an order-of-magnitude spread in start-up cost, and the cost is constant for a range of small message sizes, introducing a substantial nonlinearity in the time per message. The time is nearly linear for a range of large messages, where the slope of the curve gives the bandwidth.

decade; this improvement essentially tracks improvements in cycle time, as indicated by the middle column of the table. As illustrated by the columns on the right, improvements in start-up cost have failed to keep pace with improvements in either floating-point performance or in bandwidth. Notice that the start-up costs are on an entirely different scale from the hardware latency of the communication network, which is typically a few microseconds to a fraction of a microsecond. It is dominated by processing overhead on each message transaction, the protocol, and the processing required to provide the synchronization semantics of the message-passing abstraction.

Figure 7.33 shows the measured one-way communication time on the echo test for several machines as a function of message size. In this log-log plot, the difference in start-up costs is apparent, as is the nonlinearity for small messages. The bandwidth is given by the slope of the lines. This is more clearly seen from plotting the equivalent $BW(n)$ in Figure 7.34. A caveat must be made in interpreting the bandwidth data for message passing, since the pairwise bandwidth only reflects the data rate on a point-to-point basis. We know, for example, that for the bus-based

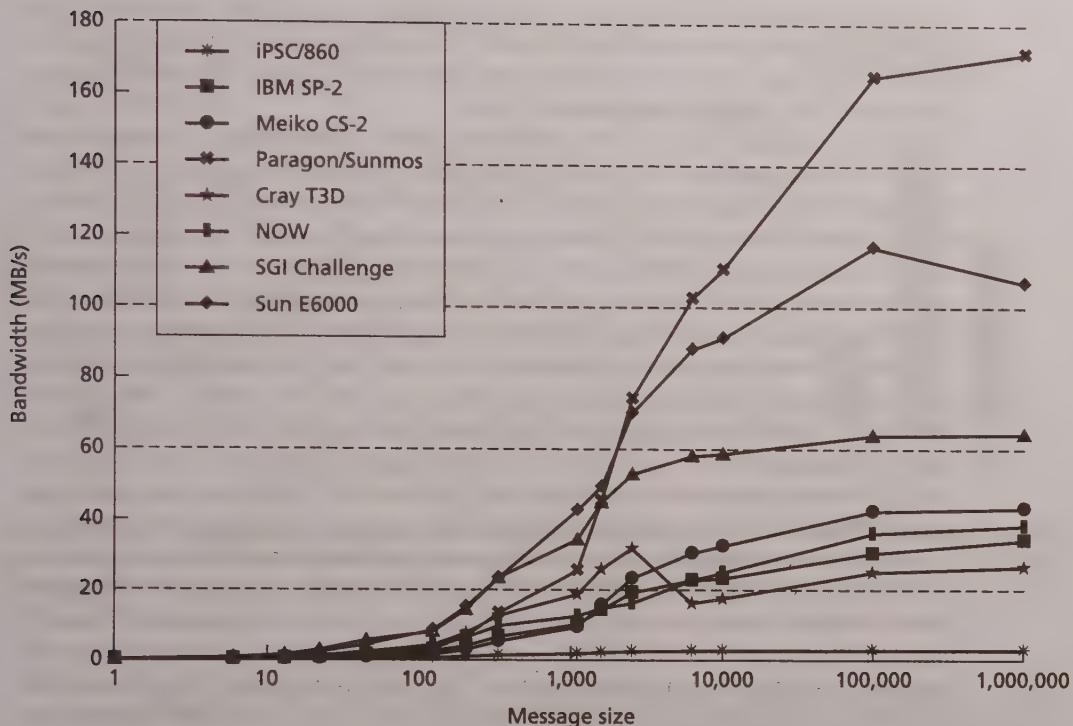


FIGURE 7.34 Bandwidth versus message size. Scalable machines generally provide a few tens of MB/s per node on large message transfers, although the Paragon design shows that this can be pushed into the hundreds of MB/s. The bandwidth is limited primarily by the ability of the DMA support to move data through the memory system, including the internal buses. Small-scale shared memory designs exhibit a point-to-point bandwidth dictated by out-of-cache memory copies when few simultaneous transfers are ongoing.

machines this is not likely to be sustained if many pairs of nodes are communicating. We will see in Chapter 10 that some networks can sustain much higher aggregate bandwidths than others. In particular, the Paragon data is optimistic for many communication patterns involving multiple pairs of nodes, whereas the other large-scale systems can sustain the pairwise bandwidth for most patterns.

7.8.4 Application-Level Performance

The end-to-end effects of all aspects of the computer system come together to determine the performance obtained at the application level. This level of analysis is most useful to the end user and is usually the basis for procurement decisions. It is also the ultimate basis for evaluating architectural trade-offs, but this requires mapping cumulative performance effects down to root causes in the machine and in the program. Typically, this mapping involves profiling the application to isolate the por-

tions where most time is spent and extracting the usage characteristics of the application to determine its requirements. This section briefly compares performance on our case study machines for two of the NAS parallel benchmarks in the NPB2 suite (NAS Parallel Benchmarks 1998).

The *LU benchmark* solves a finite difference discretization of the 3D compressible Navier-Stokes equations used for modeling fluid dynamics through a block-lower-triangular, block-upper-triangular approximate factorization of the difference scheme. The LU factored form is cast as a relaxation and is solved by symmetric successive overrelaxation (SSOR). The LU benchmark is based on the NAS reference implementation from 1991 using the Intel message-passing library, NX (Barszcz et al. 1993). It requires a power-of-two number of processors. A 2D partitioning of the 3D data grid onto processors is obtained by halving the grid repeatedly in the first two dimensions, alternating between x and y , until all processors are assigned, resulting in vertical pencil-like grid partitions. Each pencil can be thought of as a stack of horizontal tiles. The ordering of point-based operations constituting the SSOR procedure proceeds on diagonals that progressively sweep from one corner on a given z plane to the opposite corner of the same z plane, thereupon proceeding to the next z plane. This constitutes a diagonal pipelining method and is called a “wavefront” method by its authors (Barszcz et al. 1993). The software pipeline spends relatively little time filling and emptying and is well load balanced. Communication of partition boundary data occurs after completion of computation on all diagonals that contact an adjacent partition. The result is a relatively large number of small communications. Still, the total communication volume is small compared to computation expense, making this parallel LU scheme relatively efficient. Cache block reuse in the relaxation sections is high.

Figure 7.35 shows the speedups obtained on sparse LU with the Class A input (200 iterations on a $64 \times 64 \times 64$ grid) for the IBM SP-2 (wide nodes), the CRAY T3D, and the UltraSparc cluster (NOW). The speedup is normalized to the performance on four processors, shown in the right portion of the figure, because this is the smallest number of nodes for which the problem can be run on the T3D. We see that scalability is generally good to beyond 100 processors, but both the T3D and NOW scale considerably better than the SP-2. This is consistent with the ratio of the processor performance to the performance of small message transfers.

The *BT algorithm* solves three sets of uncoupled systems of equations—first in the x , then in the y , and finally in the z direction. These systems are block tridiagonal with 5×5 blocks and are solved using a multipartition scheme (Bruno and Cappello 1988). The multipartition approach provides good load balance and uses coarse-grained communication. Each processor is responsible for several disjoint subblocks of points (cells) of the grid. The cells are arranged such that for each direction of the line-solve phase, the cells belonging to a certain processor will be evenly distributed along the direction of the solution. This allows each processor to perform useful work throughout a line solve, instead of being forced to wait for the partial solution to a line from another processor before beginning work. Additionally, the information from a cell is not sent to the next processor until all sections of linear equation systems handled in this cell have been solved. Therefore, the granularity of commu-

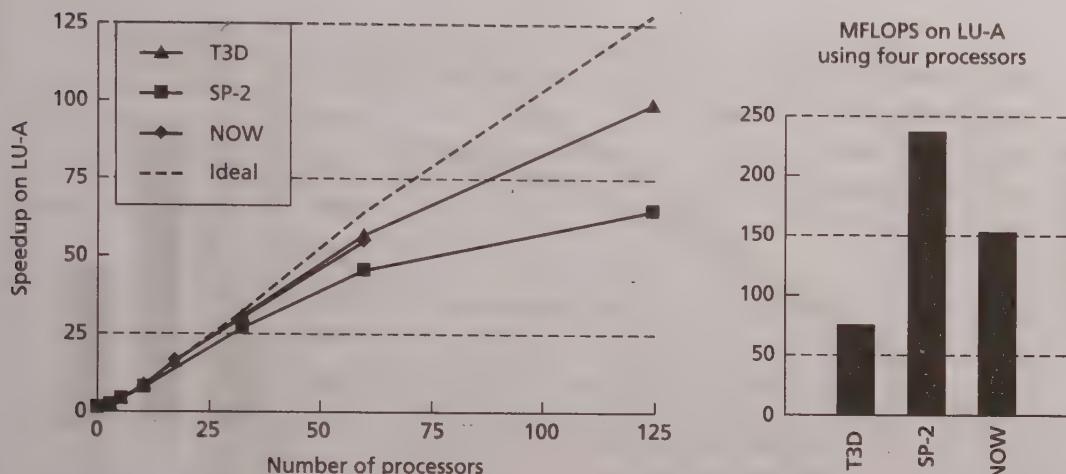


FIGURE 7.35 Application performance on sparse LU NAS parallel benchmark. Scalability of the IBM SP-2, the CRAY T3D, and the NOW UltraSparc cluster on the sparse LU benchmark is shown (left) normalized to the performance obtained on four processors. The base performance of the three systems is shown on the right.

nication is kept large and fewer messages are sent. The BT codes require a square number of processors. These codes have been written so that if a given parallel platform permits only a power-of-two number of processors to be assigned to a job, then unneeded processors are deemed inactive and are ignored during computation but are counted when determining MFLOPS rates.

Figure 7.36 shows the scalability of the IBM SP-2, the CRAY T3D, and the UltraSparc NOW on the Class A problem of the BT benchmark. Here the speedup is normalized to the performance on 25 processors, shown in the right portion of the figure, because that is the smallest T3D configuration for which performance data is available. The scalability of all three platforms is good, with the SP-2 still lagging somewhat but having much higher per-node performance.

To understand why the scalability is less than perfect, we need to look more closely at the characteristics of the application, in particular at the communication characteristics. To focus our attention on the dominant portion of the application, we will study one iteration of the main outermost loop. Typically, the application performs a few hundred of these iterations after an initialization phase. Rather than employ the simulation methodology of the previous chapters to examine communication characteristics, we collect this information by running the program using an instrumented version of the MPI message-passing library. One useful characterization is the histogram of messages by size. For each message that is sent, a counter associated with its size bin is incremented. The result, summed over all processors, is shown in the top portion of Table 7.2 for a fixed problem size and processors scaled from 4 to 64. For each configuration, the nonzero bins are indicated along with the number of messages of that size and an estimate of the total amount of data

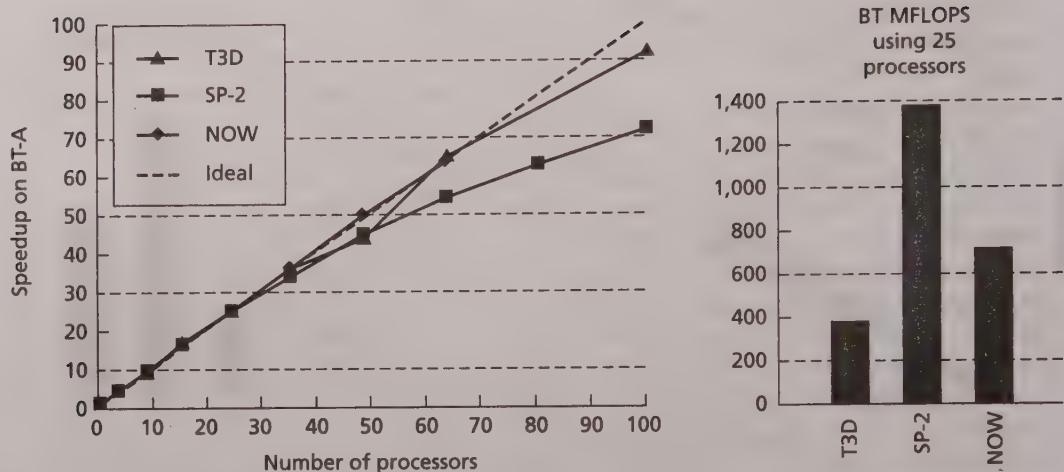


FIGURE 7.36 Application performance on BT NAS parallel benchmark. Scalability of the IBM SP-2, the CRAY T3D, and an UltraSparc cluster on the BT benchmark is shown (left) normalized to the performance obtained on 25 processors. The base performance of the three systems is shown on the right.

transferred in those messages. We see that this application essentially sends three sizes of messages, but these sizes and frequencies vary strongly with the number of processors. Both of these properties are typical of message-passing programs. Well-tuned programs tend to be highly structured and use communication sparingly. In addition, since the problem size is held fixed, as the number of processors increases, the portion of the problem that each processor is responsible for decreases. Since the data transfer size is determined by how the program is coded, rather than by machine parameters, it can change dramatically with configuration. Whereas on small configurations the program sends a few very large messages, on a large configuration it sends many relatively small messages. The total volume of communication increases by almost a factor of eight when the number of processors increases by 16. This increase is certainly one factor affecting the speedup curve. In addition, the machine with higher start-up cost is affected more strongly by the decrease in message size.

It is important to observe that with a fixed problem size scaling rule, the workload on each processor changes with scale. Here we see it for communication, but it also changes cache behavior and other factors. The bottom portion of Table 7.2 gives an indication of the average communication requirement for this application. Taking the total communication volume and dividing by the time per iteration⁹ on the UltraSparc cluster, we get the average delivered message data bandwidth. Indeed, the

9. The execution time is the only data in the table that is a property of the specific platform. All of the other communication characteristics are a property of the program and are the same when measured on any platform.

Table 7.2 Communication Characteristics for One Iteration of BT on the Class A Problem over a Range of Processor Counts

4 Processors			16 Processors			36 Processors			64 Processors		
Message Size (KB)	Total Data Transfer (KB)	Message Size (KB)	Total Data Transfer (KB)	Message Size (KB)	Total Data Transfer (KB)	Total Data Transfer (KB)	Message Size (KB)	Message Size (KB)	Total Data Transfer (KB)	Message Size (KB)	Total Data Transfer (KB)
43.5+	12	51.3	11.5+	144	1,652	4.5+	540	2,505	3+	1,344	4,266
81.5+	24	1,916	61+	96	5,742	29+	540	15,425	19+	1,344	25,266
261+	12	3,062	69+	144	9,738	45+	216	9,545	35.5+	384	13,406
Total Communication Volume (KB)	5,491			17,132			27,475			42,938	
Time per Iteration (s)	5.43			1.46			0.67			0.38	
Average Bandwidth (MB/s)	1.0			11.5			40.0			110.3	
Average Bandwidth per Processor (MB/s)	0.25			0.72			1.11			1.72	

communication rate scales substantially, increasing by a factor of more than 100 over an increase in machine size of 16. Dividing further by the number of processors, we see that the average per-processor bandwidth is significant but not extremely large. It is in the same general ballpark as the rates we observed for shared address space applications on simulations of SMPs in Section 5.4. However, we must be extremely careful about making design decisions based on average communication requirements because communication tends to be very bursty. Often, substantial periods of computation are punctuated by intense communication. For the BT application, we can get an idea of the temporal communication behavior by taking snapshots of the message histogram at regular intervals. The result is shown in Figure 7.37 for several iterations on one of the 64 processors executing the program. For each sample interval, the bar shows the size of the largest message sent in that interval. For this application, the communication profile is similar on all processors because it follows a bulk synchronous style with all processors alternating between local computation and phases of communication. The three sizes of messages are

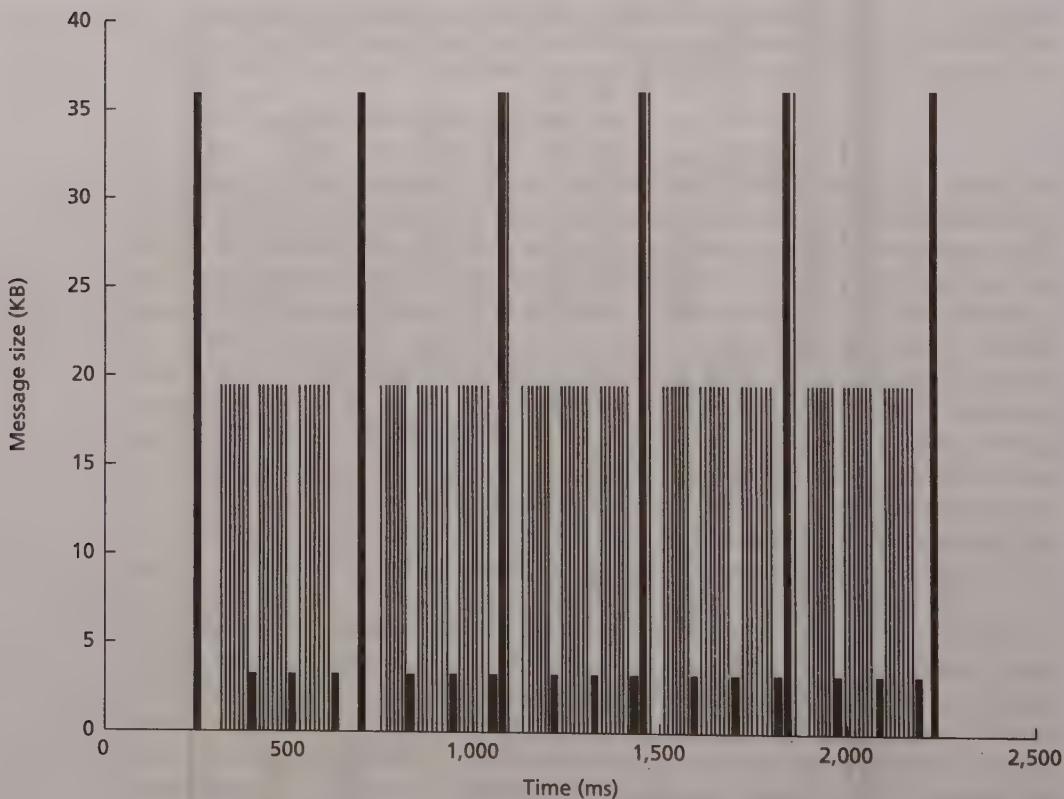


FIGURE 7.37 Message profile over time for BT-A on 64 processors. The program is executed with an instrumented MPI library that samples the communication histogram at regular intervals. The graph shows the largest message sent from one particular processor during each sample interval. It is clear that communication is regular and bursty.

clearly visible, repeating in a regular pattern with the two smaller sizes more common than the larger one. Overall, there is substantially more white space than dark, so the average communication bandwidth is more of an indication of the ratio of communication to computation than it is of the rate of communication during a communication phase.

If we apply the framework provided by Equation 3.1 to the speedup measurements on the NAS parallel benchmarks, we find that all of the parallelism costs increase with the number of processors. The extra work increases, the amount of communication increases, the cost of that communication increases, and the wait time increases. Nonetheless, several of these benchmarks obtain perfect speedup on a sizable number of processors. The reason is that the computational work becomes more efficient as the same size problem is spread over a larger number of processors. In particular, the working set behavior illustrated in Figure 3.6 has a very significant impact, even though the programs are in a message-passing programming model. The impact of this effect can be seen in Figure 7.38 for LU. Each curve in the figure shows the cache miss rate on a typical node executing the parallel program as a function of the cache size. This is obtained by running the program in parallel, collecting a cache trace on each node, and feeding the trace through a cache simulator that models fully associative caches with 64-byte blocks of various sizes. The key point to observe is that each machine size has a different working set profile. On four processors, the knee occurs at 512 KB, whereas on 32 processors it occurs at

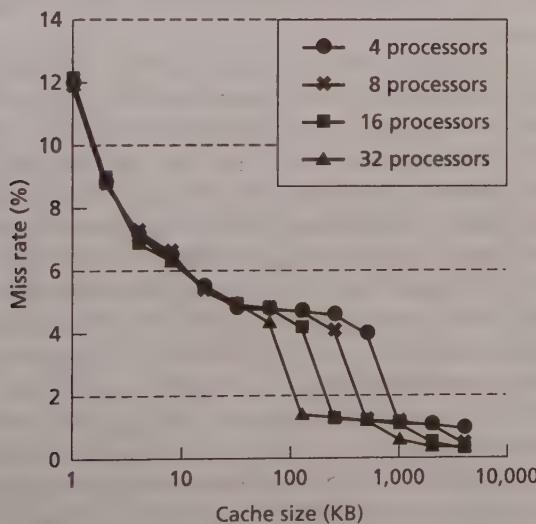


FIGURE 7.38 Working set curves for the NAS Parallel Benchmark LU on a range of machine sizes. With the CPS scaling, the curve changes for each machine size since the data set is spread over a different number of nodes. The separation of the curves for large cache sizes improves the observed speedup by reducing the time spent in computation. This effect does not occur, indeed the opposite often appears, for small cache configurations common in the mid-1980s.

64 KB. Thus, on a machine with 256-KB caches, the miss rate drops from 5% to 1% as the configuration scales from four to sixteen or more processors. Indeed, the sum of the computational time over all the processors drops as the configuration scales, so perfect speedup is obtained even though the time spent communicating increases. This effect is less pronounced on the SP-2 because the basic node is optimized for operation on data that does not fit in the cache.

7.9 SYNCHRONIZATION

Scalability is a primary concern in the combination of software and hardware that implements synchronization operations in large-scale distributed-memory machines. With a message-passing programming model, mutual exclusion is a given since each process has exclusive access to its local address space. Point-to-point events are implicit in every message operation. The more interesting case is orchestrating global or group synchronization from point-to-point messages. An important issue here is balance: it is important that the communication pattern used to achieve the synchronization be balanced among nodes, in which case high message rates and efficient synchronization can be realized. In the extreme, we should avoid having all processes communicate with or wait for one process at a given time. Machine designers and implementers of message-passing layers attempt to maximize the message rate in such circumstances, but only the program can relieve the load imbalance. Other issues for global synchronization are similar to those for a shared address space.

In a shared address space, the issues for mutual exclusion and point-to-point events are essentially the same as those discussed in Chapter 5. As in small-scale shared memory machines, the trend in scalable machines is to build user-level synchronization operations (like locks and barriers) in software on top of basic atomic exchange primitives. Two major differences, however, may affect the choice of algorithms. First, the interconnection network is not centralized but has many parallel paths. On one hand, this means that disjoint sets of processors can coordinate with one another in parallel on entirely disjoint paths; on the other hand, it can complicate the implementation of synchronization primitives. Second, physically distributed memory may make it important to allocate synchronization variables appropriately among memories. The importance of this depends on whether the machine caches nonlocal shared data or not and is clearly greater for machines that do not, such as the ones described in this chapter. This section covers new algorithms for locks and barriers appropriate for machines with physically distributed memory and interconnect, starting from the algorithms discussed for shared memory machines. We will return to this comparison once we have studied scalable cache-coherent systems in the next chapter. Let us begin with algorithms for locks.

7.9.1 Algorithms for Locks

Section 5.5 presents the basic test&set lock, the test&set lock with backoff, the test-and-test&set lock, the ticket lock, and the array-based lock. Each successive step

went further in reducing bus traffic and fairness but often at a cost in overhead. For example, the ticket lock allowed only one process to issue a test&set when a lock was released, but all processors were notified of the release through an invalidation and a subsequent read miss to determine who should issue the test&set. The array-based lock fixed this problem by having each process wait on a different location and the releasing process notify only one process of the release by writing the corresponding location.

However, the array-based lock has two potential problems for scalable machines with physically distributed memory. First, each lock requires space proportional to the number of processors. Second, and more important for machines that do not cache remote data, there is no way to know ahead of time which location a process will spin on since this is determined at run time through a fetch&increment operation. This makes it impossible to allocate the synchronization variables in such a way that the variable a process spins on is always in its local memory (in fact, all of the locks in Chapter 5 have this problem). On a distributed-memory machine without coherent caches, such as the CRAY T3D and T3E, this is a big problem since processes will spin on remote locations, causing inordinate amounts of traffic and contention. Fortunately, a software lock algorithm is available that both reduces the space requirements and ensures all spinning will be on locally allocated variables. This lock, known as a *software queuing lock*, is a software implementation of a lock originally proposed for an all-hardware implementation by the Wisconsin Multicube project (Goodman, Vernon, and Woest 1989). The idea is to have a distributed linked list or a queue of waiters on the lock. The head node in the list represents the process that holds the lock. Every other node is a process that is waiting on the lock and is allocated in that process's local memory. A node points to the process (node) that tried to acquire the lock just after it. There is also a tail pointer that points to the last node in the queue, that is, the last node to have tried to acquire the lock. Let us look pictorially at how the queue changes as processes acquire and release the lock; then we will examine the code for the acquire and release methods.

Assume that the lock in Figure 7.39 is initially free. When process A tries to acquire the lock, it gets it, and the queue looks as shown in Figure 7.39(a). In step (b), process B tries to acquire the lock, so it is put on the queue and the tail pointer now points to it. Process C is treated similarly when it tries to acquire the lock in step (c). B and C are now spinning on local flags associated with their queue nodes while A holds the lock. In step (d), process A releases the lock. It then "wakes up" the next process, B, in the queue by writing the flag associated with B's node, and leaves the queue. B now holds the lock and is at the head of the queue. The tail pointer does not change. In step (e), B releases the lock similarly, passing it on to C. There are no other waiting processes, so C is at both the head and tail of the queue. If C releases the lock before another process tries to acquire it, then the lock pointer will be NULL and the lock will be free again. In this way, processes are granted the lock in FIFO order with regard to the order in which they tried to acquire it. The latter order will be defined next.

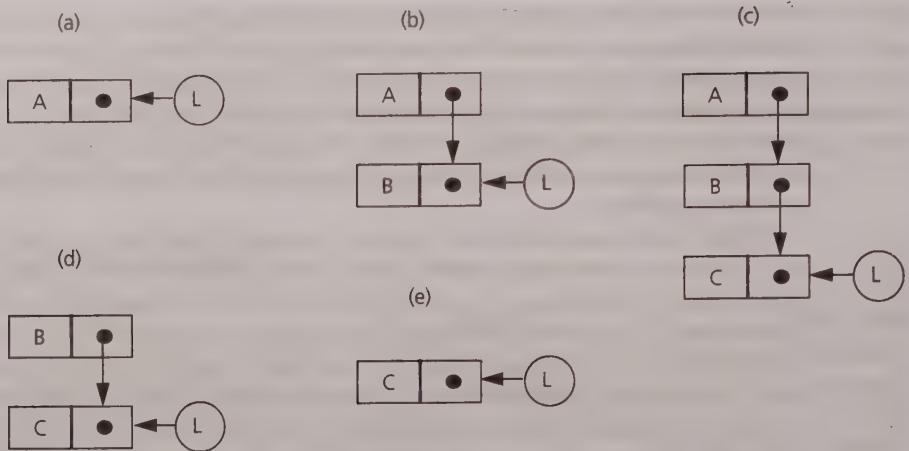


FIGURE 7.39 States of the queue for a lock as processes try to acquire and as processes release. The queue grows as new waiters are added to the tail. When the lock is released, the next waiter at the head is notified. Waiters always spin on local locations.

The code for the acquire and release methods is shown in Figure 7.40. In terms of primitives needed, the key is to ensure that changes to the tail pointer are atomic. In the acquire method, the acquiring process wants to change the lock pointer to point to its node. It does this using an atomic *fetch&store* operation, which takes two operands: it returns the current value of the first operand (here the current tail pointer) and then sets it to the value of the second operand, returning only when it succeeds. The order in which the atomic *fetch&store* operations of different processes succeed defines the order in which they acquire the lock.

In the release method, we want to atomically check if the process doing the release is the last one in the queue, and if so, set the lock pointer to NULL. We can do this using an atomic *compare&swap* operation, which takes three operands: it compares the first two (here the tail pointer and the node pointer of the releasing process), and if they are equal, it sets the first (the tail pointer) to the third operand (here NULL) and returns TRUE; if they are not equal, it does nothing and returns FALSE. The setting of the lock pointer to NULL must be atomic with the comparison since otherwise another process could slip in between and add itself to the queue, in which case setting the lock pointer to NULL would be the wrong thing to do. Recall from Chapter 5 that a *compare&swap* is difficult to implement as a single machine instruction since it requires three operands in a memory instruction (the functionality can, however, be implemented using load-locked and store-conditional instructions). It is possible to implement this queuing lock without a *compare&swap*—using only a *fetch&store*—but the implementation is more complicated (it allows the queue to be broken and then repairs it), and it loses the FIFO property of lock granting (Michael and Scott 1996).

```

struct node {
    struct node *next;
    int locked;
} *mynode, *prev_node;
shared struct node *Lock;

lock (Lock, mynode) {
    mynode->next = NULL;           /*make me last on queue*/
    prev_node = fetch&store(Lock, mynode); /*Lock currently points to the previous tail of
                                             the queue; atomically set prev_node to the
                                             Lock pointer and set Lock to point to my node
                                             so I am last in the queue*/
    if (prev_node != NULL) {        /*if by the time I get on the queue I am not the
                                   only one, i.e., some other process on queue
                                   still holds the lock*/
        mynode->locked = TRUE;     /*Lock is locked by other process*/
        prev_node->next = mynode; /*connect me to queue*/
        while (mynode->locked) {}; /*busy-wait till I am granted the lock*/
    }
}

unlock (Lock, mynode) {
    if (mynode->next == NULL) {    /*no one to release, it seems*/
        if compare&swap(Lock, mynode, NULL) /*really no one to release*/
            return;                  /*i.e., Lock points to me, then set Lock to
                                         NULL and return*/
        while (mynode->next == NULL); /*if I get here, someone just got on the
                                         queue and made my c&s fail, so I should wait
                                         till they set my next pointer to point to
                                         them before I grant them the lock*/
    }
    mynode->next->locked = FALSE; /*someone to release; release them*/
}

```

FIGURE 7.40 Algorithm for the software queuing lock. The data for the lock is a list of length equal to the number of waiters. A node requests the lock by atomically adding an item to the tail of the list and spinning on the local item until an unlock by a previous requestor provides notification.

It should be clear that the software queuing lock needs only as much space per lock as the number of processes waiting on or participating in the lock, not space proportional to the number of processes in the program. It is the lock of choice for machines that support a shared address space with distributed memory but without coherent caching (Kägi, Burger, and Goodman 1997).

7.9.2 Algorithms for Barriers

In both message-passing and shared address space models, global events like barriers are a key concern. A question of considerable debate is whether special hardware support is needed for global operations or whether sophisticated software algorithms upon point-to-point operations are sufficient. The CM-5 represented one end of the spectrum, with a special “control” network providing barriers, reductions, broadcasts, and other global operations over a subtree of the machine. The CRAY T3D provided hardware support for barriers also. Since it is easy to construct barriers that spin only on local variables or use only point-to-point messages, many scalable machines provide no special support for barriers at all but build them in software libraries.

In the centralized barrier used on bus-based machines, all processors used the same lock to increment the same counter when they signaled their arrival, and all waited on the same flag variable until they were released. On a large machine, the allowing for all processors to access the same lock and to read and write the same variables can lead to a lot of traffic and contention. Again, this is particularly true of machines that are not cache coherent, where the variable quickly becomes a hot spot as several processors spin on it without caching it.

It is possible to implement the arrival and departure in a more distributed way, in which not all processes have to access the same variable or lock. The coordination of arrival or release can be performed in phases or rounds with subsets of processes coordinating with one another in each round, such that after a few rounds all processes are synchronized. The coordination of different subsets can proceed in parallel with no serialization needed across them. In a bus-based machine, distributing the necessary coordination actions wouldn't matter much since the bus serializes all actions that require communication anyway; however, it can be very important in machines with distributed memory and interconnect where different subsets can coordinate in different parts of the network. The techniques used in a shared address space closely reflect natural message-passing approaches. Let us examine a few such distributed-barrier algorithms.

Software Combining Trees

A simple distributed way to coordinate the arrival or release of processes is through a tree structure (see Figure 7.41), just as was suggested for avoiding hot spots in Chapter 3. An arrival tree is a tree that processors use to signal their arrival at a barrier. It replaces the single lock and counter of the centralized barrier by a tree of counters. The tree may be of any chosen degree or branching factor, say, k . In the simplest case, each leaf of the tree is a process that participates in the barrier. When a process arrives at the barrier, it signals its arrival by performing a `fetch&increment` on the counter associated with its parent (or by sending a message to the parent). It then checks the value returned by the `fetch&increment` to see if it was the last of its siblings to arrive. If not, its work for the arrival is done and it simply waits for the release. If so, it considers itself chosen to represent its siblings at the next level of the

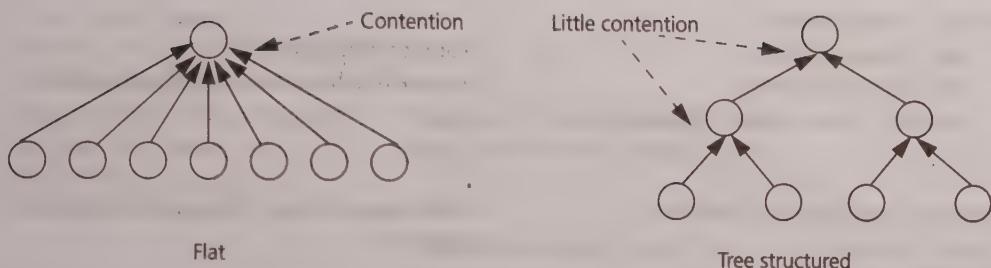


FIGURE 7.41 Replacing a flat arrival structure for a barrier by an arrival tree (here of degree 2). A deeper tree with smaller branching utilizes the many paths through the network of a large-scale machine to avoid serialization.

tree and so does a fetch&increment on the counter at that level. In this way, each tree node sends only a single representative process up to the next higher level in the tree when all the processes represented by that node's children have arrived. For a tree of degree k , it takes $\log_k p$ levels and hence that many steps to complete the arrival notification of p processes. If subtrees of processes are placed in different parts of the network and if the counter variables at the tree nodes are distributed appropriately across memories, fetch&increment operations on nodes that do not have an ancestor-descendent relationship need not be serialized at all.

A similar tree structure can be used for the release as well, so all processors don't busy-wait on the same flag. That is, the last process to arrive at the barrier sets the release flag associated with the root of the tree, on which only $k - 1$ processes are busy-waiting. Each of the k processes then sets a release flag at the next level of the tree, on which $k - 1$ other processes are waiting, and so on down the tree until all processes are released. (Similarly, messages can be passed down the tree.) The critical path length of the barrier in terms of the number of dependent or serialized operations (e.g., network transactions) is thus $O(\log_k p)$ as opposed to $O(p)$ for the centralized barrier or $O(p)$ for any barrier on a centralized bus. The code for a simple combining tree barrier with sense reversal is shown in Figure 7.42.

Although this tree barrier distributes traffic in the interconnect, it has the same problem as the simple lock for machines that do not cache remote shared data: the variables that processors spin on are not necessarily allocated in their local memory. Multiple processors spin on the same variable, and which processors reach the higher levels of the tree and spin on the variables there depends on the order in which processors reach the barrier and perform their fetch&increment instructions, which is impossible to predict. This leads to a lot of network traffic while spinning.

Tree Barriers with Local Spinning

There are two ways to ensure that a processor spins on a local variable. One is to pre-determine which processor moves up from a node to its parent in the tree, based on the process identifier and the number of processes participating in the barrier. In this

```

struct tree_node {
    int count = 0;                                /*counter initialized to 0*/
    int local_sense;                            /*release flag implementing sense reversal*/
    struct tree_node *parent;
}

struct tree_node tree[P];                      /*each element (node) allocated in a different
                                                memory*/

private int sense = 1;
private struct tree_node *myleaf; /*pointer to this process's leaf in the tree*/

barrier () {
    barrier_helper(myleaf);
    sense = !(sense);                         /*reverse sense for next barrier call*/
}

barrier_helper(struct tree_node *mynode) {
    if (fetch&increment (mynode->count) == k-1){ /*last to reach node*/
        if (mynode->parent != NULL)
            barrier_helper(mynode->parent);          /*go up to parent node*/
        mynode->count = 0;                          /*set up for next time*/
        mynode->local_sense = !(mynode->local_sense); /*release*/
    }
    while (sense != mynode->local_sense) [];      /*busy-wait*/
}

```

FIGURE 7.42 A software combining barrier algorithm with sense reversal. Each time the barrier is used, the sense of the flag is reversed, so the flag does not need to be reset.

case, a binary tree makes local spinning easy since the flag to spin on can be allocated in the local memory of the spinning processor rather than the one that goes up to the parent level. In fact, in this case, it is possible to perform the barrier without any atomic operations like `fetch&increment` but with only simple reads and writes as follows. For arrival, one process arriving at each node simply spins on an arrival flag associated with that node. The other process associated with that node simply writes the flag when it arrives. The process whose role was to spin now simply spins on the release flag associated with that node while the other process now proceeds up to the parent node. Such a static binary tree barrier has been called a “tournament barrier” in the literature, since one process can be thought of as dropping out of the tournament at each step in the arrival tree. (As an exercise, think about how you might modify this scheme to handle the case where the number of participating processes is not a power of two and to use a nonbinary tree.)

The other way to ensure local spinning is to use p -node trees to implement a barrier among p processes, where each tree node (leaf or internal) is assigned to a unique process. The arrival and wake-up trees can be the same, or they can be main-

```

struct tree_node {
    struct tree_node *parent;
    int parent_sense = 0;
    int wkup_child_flags[2]; /*flags for children in wake-up tree*/
    int child_ready[4];      /*flags for children in arrival tree*/
    int child_exists[4];
}
} /*nodes are numbered from 0 to P - 1 level-by-level starting
   from the root*/
struct tree_node tree[P]; /*each element (node) allocated in a different memory*/
private int sense = 1, myid;
private me = tree[myid];

barrier() {
    while (me.child_ready is not all TRUE) {}; /*busy-wait*/
    set me.child_ready to me.child_exists; /*reinitialize for next barrier call*/
    if (myid !=0) {                      /*set parent's child_ready flag, and wait for release*/
        tree[ $\left\lfloor \frac{myid-1}{4} \right\rfloor$ ].child_ready[(myid-1) mod 4] = true;
        while (me.parent_sense != sense) {};
    }
    me.child_pointers[0] = me.child_pointers[1] = sense;
    sense = !sense;
}

```

FIGURE 7.43 A combining tree barrier that spins on local variables only. Each tree node is assigned to a unique process and allocated in the memory that is local to the process.

tained as different trees with different branching factors. Each internal node (process) in the tree maintains an array of arrival flags, with one entry per child, allocated in that node's local memory. When a process arrives at the barrier, if its tree node is not a leaf, then it first checks its arrival flag array and waits until all its children have signaled their arrival by setting the corresponding array entries. Then it sets its entry in its parent's (remote) arrival flag array and busy-waits on the release flag associated with its tree node in the wake-up tree. When the root process arrives and when all its arrival flag array entries are set, this means that all processes have arrived. The root then sets the (remote) release flags of all its children in the wake-up tree; these processes break out of their busy-wait loop and set the release flags of their children, and so on until all processes are released. The code for this barrier is shown in Figure 7.43, assuming an arrival tree of branching factor 4 and a wake-up tree of branching factor 2. In general, choosing branching factors in tree-based barriers is largely a trade-off between contention and critical path length counted in network transactions. Either of these types of barriers may work well for scalable machines without coherent caching.

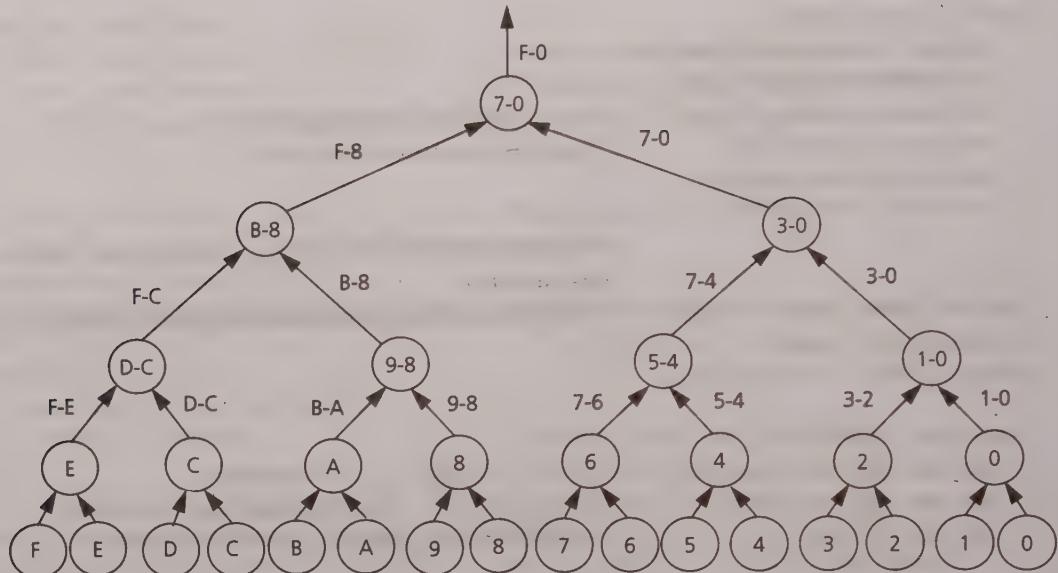


FIGURE 7.44 Upward sweep of the parallel prefix operation. Each node receives two elements from its children, combines them and passes the result to its parent, and holds the element from the least significant (right) child.

Parallel Prefix

In many parallel applications, a point of global synchronization is associated with combining information that has been computed by many processors and distributing a result based on the combination. Parallel prefix operations are an important, widely applicable generalization of reductions and broadcasts (Blelloch 1993). Given some associative binary operator \oplus , we want to compute $S_i = x_i \oplus x_{i-1} \dots \oplus x_0$ for $i = 0, \dots, P$. A canonical example is a running sum, but several other operators are useful. The carry-lookahead operator from adder design is actually a special case of a parallel prefix circuit. The surprising fact about parallel prefix operations is that they can be performed as quickly as a reduction followed by a broadcast, with a simple pass up a binary tree and back down. Figure 7.44 shows the upward sweep, in which each node applies the operator to the pair of values it receives from its children and passes the result to its parent, just as with a binary reduction. (The value that is transmitted is indicated by the range of indices next to each arc; this is the subsequence over which the operator is applied to get that value.) In addition, each node holds onto the value it received from its least significant child (rightmost in the figure). Figure 7.45 shows the downward sweep. Each node waits until it receives a value from its parent. It passes this value along unchanged to its rightmost child. It combines this value with the value that was held over from the upward pass and passes the result to its left child. The nodes along the right edge of the tree are

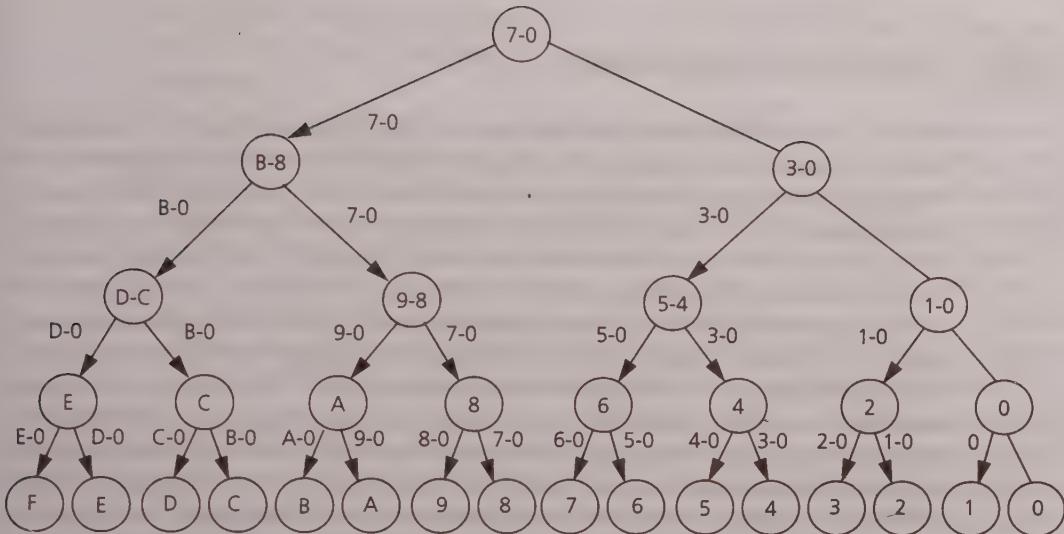


FIGURE 7.45 Downward sweep of the parallel prefix operation. When a node receives an element from above, it passes the data down to its right child, combines it with its stored element, and passes the result to its left child. Nodes along the rightmost branch need nothing from above.

special because they do not need to receive anything from their parent. This parallel prefix tree can be implemented either in hardware or in software.

All-to-All Personalized Communication

All-to-all personalized communication occurs when each process has a distinct set of data to transmit to every other process. The canonical example of this is a transpose operation, say, where each process owns a set of rows of a matrix and needs to access data in a set of columns. Another important example is remapping a data structure between blocked and cyclic layouts. Many other permutations of this form are widely used in practice. Quite a bit of work has been done in implementing all-to-all personalized communication operations efficiently on specific network topologies (i.e., with no contention internal to the network). If the network is highly scalable, the internal communication flows within the network become secondary, but contention at the endpoints of the network is critical, regardless of the quality of the network. A simple, widely used scheme is to schedule the sequence of communication events so that P rounds of disjoint pairwise exchanges are performed. In round i , process p transmits the data it has for process $q = p \oplus i$ obtained as the exclusive-or of the binary number for p and the binary representation of i . Since exclusive-or is commutative, $p = q \oplus i$, and the round is indeed an exchange.

7.10 CONCLUDING REMARKS

We have seen in this chapter that most modern large-scale machines are constructed from general-purpose nodes with a complete local memory hierarchy augmented by a communication assist interfacing to a scalable network. However, a wide range of design options is available for the communication assist. The design is influenced very strongly by where the communication assist interfaces to the node architecture: at the processor, at the cache controller, at the memory bus, or at the I/O bus. It is also strongly influenced by the target communication architecture and programming model. Programming models are implemented on large-scale machines using protocols constructed out of primitive network transactions. The challenges in implementing such a protocol are that a large number of transactions can be outstanding simultaneously and that global arbitration is unavailable. Essentially, any programming model can be implemented on any of the available primitives at the hardware/software boundary, and many of the correctness and scalability issues are the same; however, the performance characteristics are quite different. The performance ultimately influences how the machine is viewed by programmers.

Modern large-scale parallel machine designs are rooted heavily in the technological revolution of the mid-1980s—the single-chip microprocessor—which was coined the “killer micro” as the MPP machines began to take over the high-performance market from traditional vector supercomputers. However, these machines pioneered a technological revolution of their own—the single-chip scalable network switch. Like the microprocessor, this technology has grown beyond its original intended use, and a wide class of scalable system area networks are emerging, including switched gigabit (perhaps multigigabit) Ethernets. As a result, large-scale parallel machine design has split somewhat into two branches. Machines oriented largely around message-passing concepts and explicit get/put access to remote memory are being overtaken by clusters because of their extreme low cost, ease of engineering, and ability to track technology. The other branch is made up of machines that deeply integrate the network into the memory system to provide cache-coherent access to a global physical address space with automatic replication—in other words, machines that look to the programmer, like those of the previous chapter, but that are built like the machines in this chapter. Of course, the challenge is advancing the cache coherence mechanisms in a manner that provides scalable bandwidth and low latency. This is the subject of Chapter 8.

7.11 EXERCISES

- 7.1 A radix-2 FFT over n complex numbers is implemented as a sequence of $\log n$ completely parallel steps, requiring $5n \log n$ floating-point operations while reading and writing each element of data $\log n$ times. Calculate the communication-to-computation ratio on a dancehall design where all processors access memory

through the network, as in Figure 7.3. What communication bandwidth would the network need to sustain for the machine to deliver $250p$ MFLOPS on p processors?

- 7.2 If the data in Exercise 7.1 is spread over the memories in a NUMA design using either a cycle or block distribution, the $\log n/p$ of the steps will access data in the local memory, assuming both n and p are powers of two, and in the remaining $\log p$ steps half of the reads and half of the writes will be local. (The choice of layout determines which steps are local and which are remote, but the ratio stays the same.) Calculate the communication-to-computation ratio on a distributed-memory design where each processor has a local memory, as in Figure 7.2. What communication bandwidth would the network need to sustain for the machine to deliver $250p$ MFLOPS on p processors? How does this compare with Exercise 7.1?
- 7.3 If the programmer pays attention to the layout, the FFT can be implemented so that a single, global transpose operation is required where $n - n/p$ data elements are transmitted across the network. All of the $\log n$ steps are performed on local memory. Calculate the communication-to-computation ratio on a distributed-memory design. What communication bandwidth would the network need to sustain for the machine to deliver $250p$ MFLOPS on p processors? How does this compare with Exercise 7.2?
- 7.4 Reconsider Example 7.1 where the number of hops for an n -node configuration is \sqrt{n} . How does the average transfer time increase with the number of nodes? What about $\sqrt[3]{n}$?
- 7.5 Formalize the cost scaling for the designs in Exercise 7.4.
- 7.6 Consider a machine as described in Example 7.1, where the number of links occupied by each transfer is $\log n$. In the absence of contention for individual links, how many transfers can occur simultaneously?
- 7.7 Reconsider Example 7.1 where the network is a simple ring. The average distance between two nodes on a ring of n nodes is $n/2$. How does the average transfer time increase with the number of nodes? Assuming each link can be occupied by at most one transfer at a time, how many such transfers can take place simultaneously?
- 7.8 For a machine as described in Example 7.1, suppose that a broadcast from a node to all the other nodes uses n links. How would you expect the number of simultaneous broadcasts to scale with the number of nodes?
- 7.9 Suppose a 16-way SMP lists at \$10,000 plus \$2,000 per node, where each node contains a fast processor and 128 MB of memory. How much does the cost increase when doubling the capacity of the system from 4 to 8 processors? From 8 to 16 processors?
- 7.10 Prove the statement from Section 7.1.3 that parallel computing is more cost-effective whenever $\text{Speedup}(p) > \text{Costup}(p)$.

- 7.11 Assume a bus transaction with n bytes of payload occupies the bus for

$$4 + \left\lceil \frac{n}{8} \right\rceil$$

cycles, up to a limit of 64 bytes. Draw a graph comparing the bus utilization for programmed I/O and DMA for sending various-sized messages where the message data is in memory, not in registers. For DMA, include the extra work to inform the communication assist of the DMA address and length. Consider both the case where the data is in cache and where it is not. What assumptions do you need to make about reading the status registers?

- 7.12 The Intel Paragon has an output buffer of size 2 KB, which can be filled from memory at a rate of 400 MB/s and drained into the network at a rate of 175 MB/s. The buffer is drained into the network while it is being filled, but if the buffer gets full, the DMA device will stall. In designing your message layer you decide to fragment long messages into DMA bursts that are as long as possible without stalling behind the output buffer. Clearly, these can be at least 2 KB in size, but in fact they can be longer. Calculate the apparent size of the output buffer driving a burst into an empty network, given these rates of flow.
- 7.13 Based on a rough estimate from Figure 7.33, which of the machines will have a negative T_0 if a linear model is fit to the communication time data?
- 7.14 Use the message frequency data presented in Table 7.2 to estimate the time each processor would spend in communication on an iteration of BT for the machines described in Table 7.1.
- 7.15 Table 7.3 describes the communication characteristics of sparse LU.
- How do the message size characteristics differ from that of BT? What does this say about the application?
 - How does the message frequency differ?
 - Estimate the time each processor would spend in communication on an iteration of LU for the machines described in Table 7.1.

Table 7.3 Communication Characteristics for One Iteration of LU on the Class A Problem over a Range of Processor Counts

4 Processors			16 Processors			32 Processors			64 Processors		
Message Size (KB)	Messages	Total Data Transfer (KB)	Message Size (KB)	Messages	Total Data Transfer (KB)	Bins (KB)	Messages	Total Data Transfer (KB)	Bins (KB)	Messages	Total Data Transfer (KB)
1+	496	605	0.5–1	2,976	2,180	0–0.5	2,976	727	0–0.5	13,888	3,391
163.5+	8	1,279	81.5+	48	3,382	0.5–1	3,472	2,543	81.5	224	8,914
Total Communication Volume (KB)			5,562			9,651			12,305		
Time per Iteration (s)			2.4			0.58			0.34		
Average Bandwidth (MB/s)			0.77			10.1			27.7		
Average Bandwidth per Processor (MB/s)			0.19			0.63			0.87		
									0.99		

Directory-Based Cache Coherence

This chapter examines an important part of the development of parallel architectures: putting together cache coherence and a scalable, distributed-memory machine organization. We have studied cache coherence for bus-based machines with centralized memory. We have also seen that in order to scale up machines, memory is distributed, a scalable point-to-point interconnection network is introduced, and a communication assist provides varying degrees of interpretation of network transactions to support programming models. Regardless of the sophistication of that assist, all of the scalable machines we have studied have the generic structure depicted in Figure 8.1.

At the final point in our design spectrum so far, the communication assist provides a shared address space in hardware. However, while the natural inclination of caches is to replicate referenced data in a shared address space, we have not yet examined how cache coherence may be provided. In fact, to avoid the coherence problem and simplify memory consistency, the machines in that final design point disable the hardware caching of logically shared but physically remote data, restricting the programming model.

This chapter takes on the important issue of how implicit caching and coherence may be provided in hardware on a machine with physically distributed memory, without the benefits of a globally snoopable interconnect such as a bus. Not only must the hardware latency and bandwidth scale well, as we have seen, but so must the protocols used for coherence, at least up to the scales of practical interest. We focus on full hardware support for cache coherence and particularly on the most common approach called directory-based cache coherence. In terms of the layers of abstraction, the shared address space programming model with coherent replication is supported directly at the hardware/software interface, as shown in Figure 8.2. Other programming models, such as message passing, can be implemented in software. The next chapter describes some alternative approaches that take different positions on hardware/software trade-offs, such as coherent replication in main memory rather than in the caches, coherence under software control, and alternative memory consistency models.

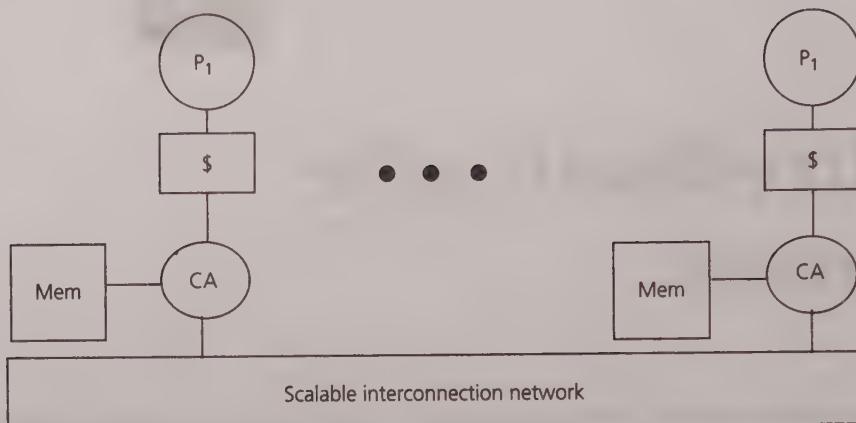


FIGURE 8.1 A generic scalable multiprocessor. This diagram represents the generic structure of the machines discussed in Chapter 7: processing nodes with physically distributed memory and a scalable interconnect. The processing nodes may be uniprocessors (as shown) or multiprocessors.

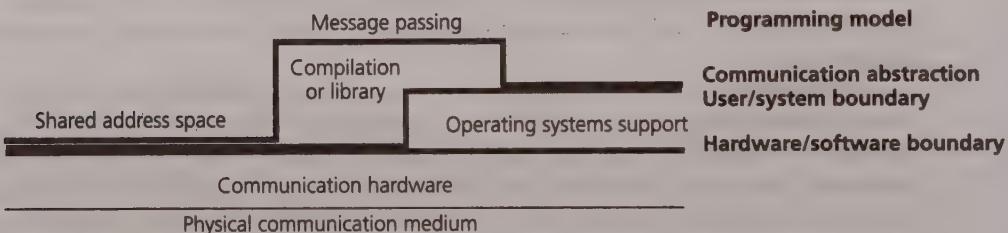


FIGURE 8.2 Layers of abstraction for systems discussed in this chapter. A coherent, shared physical address space is supported directly in hardware and message passing through software layers.

Scalable cache coherence is typically based on the concept of a directory. Since the state of a block in the caches can no longer be determined implicitly by placing a request on a shared bus and having it snooped by the cache controllers, the idea is to maintain this state explicitly in a place—called a *directory*—where requests can go and look it up. Consider a simple example. Imagine that each cache-line-sized block of main memory has associated with it a record of the caches that currently contain a copy of the block and the state of the block in those caches. This record is called the *directory entry* for that block (see Figure 8.3). As in bus-based systems, there may be many caches with a clean, readable block, but if the block is writable (possibly modified) in one cache, then only that cache may have a valid copy. When a node incurs a cache miss, it first communicates with the directory entry for the block using point-to-point network transactions. Since the directory entry is colocated with the

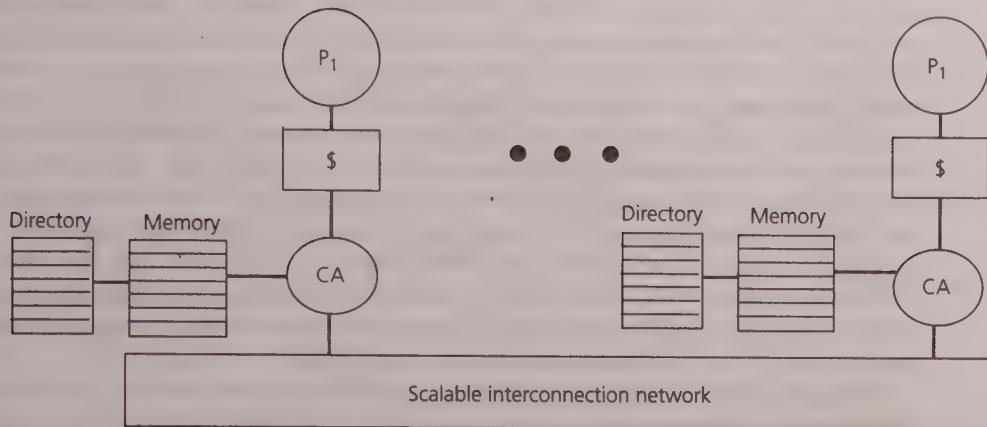


FIGURE 8.3 A scalable multiprocessor with directories. Every block of main memory, the size of a cache block, has a directory entry that keeps track of its cached copies and their state.

main memory for the block, its location can be determined from the address of the block. From the directory, the node determines where the valid cached copies (if any) are and what further actions to take. It then communicates with the cached copies as necessary using additional network transactions. For example, it may obtain a modified block from another node or, on a write operation, send invalidations to other nodes and receive acknowledgments from them. The resulting changes to the states of cached blocks are also communicated to the directory entry through network transactions, so the directory stays up-to-date.

In a directory protocol, requests, replies, invalidations, updates, and acknowledgments across nodes are all network transactions like those of the previous chapter, only here the endpoint processing at the destination of the transaction (invalidating blocks, retrieving and replying with data) is typically done by the communication assist rather than the main processor. (As in previous chapters, we will call response transactions that carry data “replies” and all others simply “responses.”) Since directory schemes rely on point-to-point network transactions, they can be used with any interconnection network. Important questions for directories include the form in which the directory information is stored and how correct, efficient protocols may be designed using these representations.

While directories constitute the dominant approach to scalable cache coherence, other approaches can be contemplated. One approach that has been tried is to extend the broadcast and snooping mechanism, using a hierarchy of broadcast media like buses or rings. This is conceptually attractive because it builds larger systems hierarchically out of existing small-scale mechanisms. However, it does not apply to general network topologies such as meshes and cubes, and we will see that it has problems with latency and bandwidth, so it has not become very popular. An

approach that is popular is a limited, two-level protocol hierarchy. Each node of the machine is itself a multiprocessor. The caches within a node are kept coherent by one coherence protocol called the *inner protocol*. Coherence across nodes is maintained by another, possibly different protocol called the *outer protocol*. To the outer protocol, each multiprocessor node looks like a single cache, and coherence within the node is the responsibility of the inner protocol. Usually, an adapter or a shared tertiary cache is used to represent a node to the outer protocol. A common organization is for the outer protocol to be a directory protocol and the inner one to be a snooping protocol (Lovett and Clapp 1996; Lenoski et al. 1993; Clark and Alnes 1996; Weber et al. 1997). However, other combinations such as snooping-snooping (Frank, Burkhardt, and Rothnie 1993), directory-directory (Convex Computer Corporation 1993), and even snooping-directory may be used (see Figure 8.4).

Putting together smaller-scale machines to build larger machines in a two-level organization is an attractive engineering option: it amortizes fixed per-node costs over the processors in a node, may take advantage of packaging hierarchies, and may satisfy much of the interprocessor communication less expensively within a node. The main focus of this chapter will be on directory protocols across nodes, regardless of whether the node is a uni- or multiprocessor or what coherence method it uses. The interactions among two-level protocols are also discussed. While we focus on directory protocols because they have been most successful and are likely to remain the most popular, we will briefly examine the less popular hierarchical approaches as well. As we examine the organizational structure of the directory, the protocols used to support coherence and consistency, and the requirements placed on the communication assist, we will find another rich and interesting design space.

The first section of this chapter presents a framework for understanding the different approaches to providing coherent replication in a shared address space, including snooping, directories, and hierarchical snooping. Section 8.2 introduces the basic operation of a directory protocol using a simple directory representation and then provides an overview of alternative directory organizations and protocols. This is followed by a quantitative assessment of some high-level issues and architectural trade-offs for directory protocols in Section 8.3.

The next few sections cover the issues and techniques involved in actually designing correct, efficient protocols. Section 8.4 discusses the major new challenges introduced by the presence of multiple copies of data without a serializing interconnect. The next two sections delve deeply into the two most popular types of directory-based protocols, discussing various design alternatives and using two commercial architectures as case studies: the Origin2000 from Silicon Graphics, Inc. and the NUMA-Q from Sequent Computer Systems, Inc. Section 8.7 examines the impact of key performance parameters of the communication architecture on the end performance of parallel programs under directory protocols.

Synchronization for directory-based multiprocessors is discussed in Section 8.8 and the implications for parallel software in Section 8.9. Section 8.10 covers some advanced topics, including the approaches of hierarchically extending snooping and directory protocols for scalable coherence.

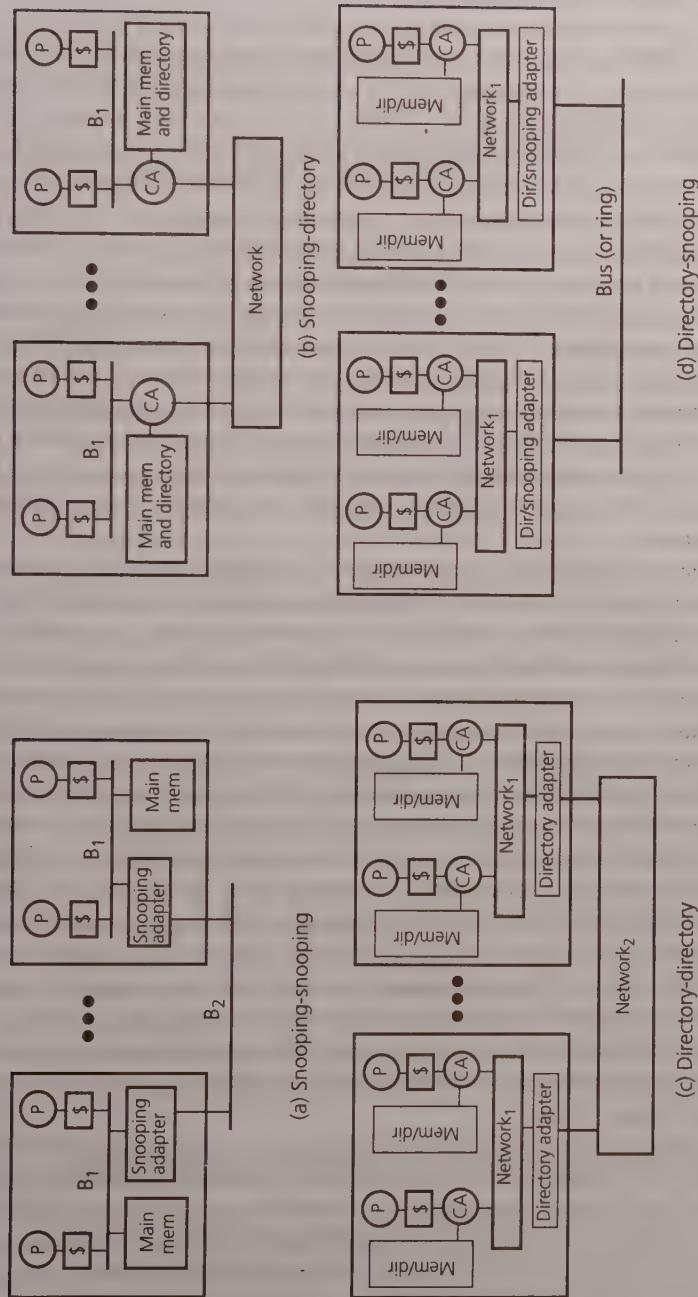


FIGURE 8.4 Some possible organizations for two-level cache-coherent systems. Each node visible at the outer level is itself a multiprocessor. B₁ is a first-level bus, and B₂ is a second-level bus. CA is the communication assist. The label snooping-directory, for example, means that a snooping protocol is used to maintain coherence within a multiprocessor node, and a directory protocol is used to maintain coherence across nodes.

8.1 SCALABLE CACHE COHERENCE

This section briefly lays out the major organizational alternatives for providing coherent replication in a multiprocessor's extended memory hierarchy and introduces the basic mechanisms that any approach to coherence must provide.

On a machine with physically distributed memory, nonlocal data may be replicated either only in the processors' caches or in the local main memory. If coherent replication is provided in main memory, additional support for keeping caches coherent may not be necessary since only data that is already coherent in the local main memory may enter the cache. This chapter assumes that data is automatically replicated only in the caches, not in main memory, and that it is kept coherent in hardware at the granularity of cache blocks, just as in bus-based machines. Since main memory is physically distributed and has nonuniform access costs to a processor, architectures of this type are often called *cache-coherent, nonuniform memory access* or CC-NUMA architectures. More generally, systems that provide a shared address space programming model with physically distributed memory and coherent replication (either in caches or main memory) are called *distributed shared memory* (DSM) systems.

Any approach to coherence, including the snooping coherence discussed in Chapters 5 and 6, must provide certain critical mechanisms. First, a block can be in each cache (or local replication store) in one of a number of states, potentially in different states in different caches. The protocol must provide these cache states as well as the state transition diagram, according to which blocks in different caches independently change states, and the set of actions associated with the state transition diagram. Directory-based protocols also have a *directory state* for each block, which is the state of the block as known to the directory. The protocol may be invalidation based, update based, or hybrid, and the stable cache states themselves are very often the same (e.g., MESI), regardless of whether the system is based on snooping or directories. The trade-offs in the choices of stable cache states are very similar to those discussed in Chapter 5 and are not revisited in this chapter. Conceptually, for any protocol, the cache state of a memory block is a vector containing its state in every cache in the system. The same state transition diagram governs the copies in different caches, though the current state of the block at any given time may be different in different caches. The state changes for a block in different caches are coordinated through transactions on the interconnect, whether bus transactions or more general network transactions.

Given a protocol at the cache state transition level, a coherent system must provide mechanisms for managing the protocol. First, a mechanism is needed to determine when (i.e., on which operations) to invoke the protocol. This is done in the same way on most systems: through an *access fault* (cache miss) detection mechanism. The protocol is invoked if the processor makes an access that its cache cannot satisfy by itself, for example, an access to a block that is not in the cache or a write access to a block that is present but in shared state. However, even when they use the same set of cache states, transitions, and access fault mechanisms, approaches to

cache coherence differ substantially in the mechanisms they provide for three important functions that may need to be performed when an access fault occurs:

1. Finding out enough information about the state of the location (cache block) in other caches to determine what action to take
2. Locating those other copies, if needed (e.g., to invalidate them)
3. Communicating with the other copies (e.g., obtaining data from them or invalidating or updating them)

In snooping protocols, all three functions are performed by the broadcast and snooping mechanism. The processor puts a “search” request on the bus, containing the address of the block, and other cache controllers snoop and respond. It is possible to use a broadcast and “snooping” method in distributed machines as well; the assist at the node incurring the miss can broadcast messages to all nodes, and their assists can examine the incoming request and respond as appropriate. However, broadcast does not scale since it generates a large amount of traffic (at least p network transactions on every miss on a p -node machine). Scalable approaches include hierarchical snooping and directory-based approaches.

In a hierarchical snooping approach, the interconnection network is not a single broadcast bus (or ring) but a tree of buses. The processors are in the bus-based snooping multiprocessors at the leaves of the tree. Parent buses are connected to children by interfaces that snoop the buses on both sides and propagate relevant transactions upward or downward in the hierarchy. Main memory may be centralized at the root or distributed among the leaves. In this case, all of the preceding functions are performed by the hierarchical extension of the broadcast and snooping mechanism: a processor puts a search request on its bus as before, and it is propagated up and down the hierarchy as necessary based on snoop results. The hope is that most of the time a request will not have to be propagated very far. Hierarchical snooping systems are discussed further in Section 8.10.2.

In the simple directory approach introduced earlier in the chapter, information about the state of blocks in other caches is found by looking up the directory through network transactions. The location of the copies is also found from the directory, and the copies are communicated with using point-to-point network transactions in an arbitrary interconnection network, without resorting to broadcast. How the directory information is actually organized influences how protocols might be structured around this organization using network transactions and, hence, how the protocol addresses the three key functions required for coherence.

8.2 OVERVIEW OF DIRECTORY-BASED APPROACHES

This section begins by more fully describing a simple directory scheme and how it might operate using cache states, directory states, and network transactions. It then discusses the organizational issues in scaling directories to large numbers of nodes, provides a classification of scalable directory organizations, and discusses the basics of protocols associated with these organizations.

The following definitions are useful for our discussion of directory protocols. For a given cache or memory block:

- The *home node* is the node in whose main memory the block is allocated.
- The *dirty node* is the node that has a copy of the block in its cache in modified (dirty) state. Note that the home node and the dirty node for a block may be the same.
- The *owner node* is the node that currently holds the valid copy of a block and must supply the data when needed; in directory protocols, this is either the home node (when the block is not in dirty state in a cache) or the dirty node.
- The *exclusive node* is the node that has a copy of the block in its cache in an exclusive state, either dirty or (clean) exclusive as the case may be. (Recall from Chapter 5 that the cache state called exclusive means this is the only valid cached copy and that the block in main memory is up-to-date.) Thus, the dirty node is also the exclusive node.
- The *local node*, or *requesting node*, is the node containing the processor that issues a request for the block.
- Blocks whose home is local to the issuing processor are called *locally allocated* or simply *local blocks*, whereas all others are called *remotely allocated* or *remote blocks*.

Let us begin with the basic operation of directory-based protocols, using a very simple directory organization.

8.2.1 Operation of a Simple Directory Scheme

When a cache miss (access control fault) is incurred, the local node sends a request network transaction to the home node where the directory information for the block is located. On a read miss, the directory indicates from which node the data may be obtained, as shown in Figure 8.5(a). On a write miss, the directory identifies the copies of the block, and invalidation or update network transactions may be sent to these copies (Figure 8.5[b]). (Recall that a write to a block in shared state is also considered a write miss.) Since invalidations or updates are sent to multiple copies through potentially disjoint paths in the network, determining the completion or commitment of a write now requires that all copies reply to invalidations with explicit acknowledgment transactions; we cannot assume completion or commitment when the read-exclusive or update request obtains access to the interconnect as we did on a shared bus since we cannot guarantee ordering with respect to other transactions within the interconnect.

A natural way to organize a directory is to maintain the directory information for a block together with the block in main memory, that is, at the home node for the block. A simple organization for the directory information for a block is as a bit vector of p *presence bits*—which indicate for each of the p nodes (uniprocessor or multi-processor) whether that node has a cached copy of the block—together with one or more state bits (see Figure 8.6). Let us assume for simplicity that there is only one state bit, called the *dirty bit*, which indicates if the block is dirty in one of the node

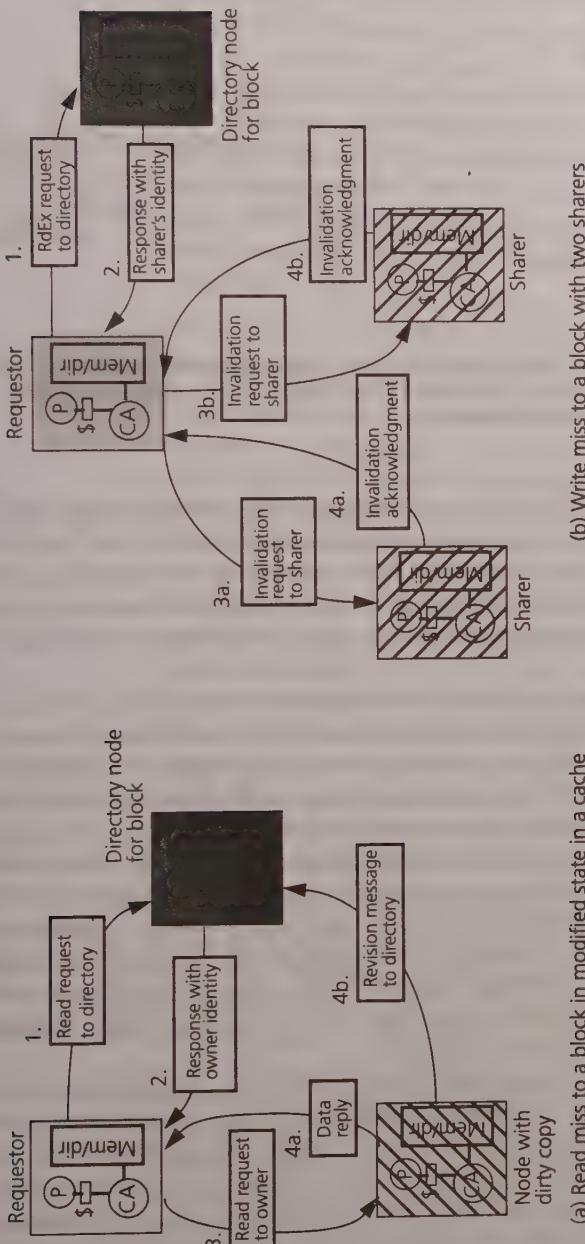


FIGURE 8.5 Basic operation of a simple directory. Two example operations are shown. On the left is a read miss to a block that is currently held in modified (dirty) state by a node that is not the requestor or the node that holds the directory information. A read miss to a block that is clean in main memory (i.e., at the directory node) is simpler: the main memory simply replies to the requestor with the data, and the miss is satisfied in a single request-response pair of transactions. On the right is a write miss to a block that is currently in shared state in two other nodes' caches (the two sharers). The big rectangles are the nodes, and the arcs (with boxed labels) are network transactions. The numbers 1, 2, and so on next to a transaction show the serialization of transactions. Different letters next to the same number indicate that the transactions can be performed in parallel and hence overlapped.

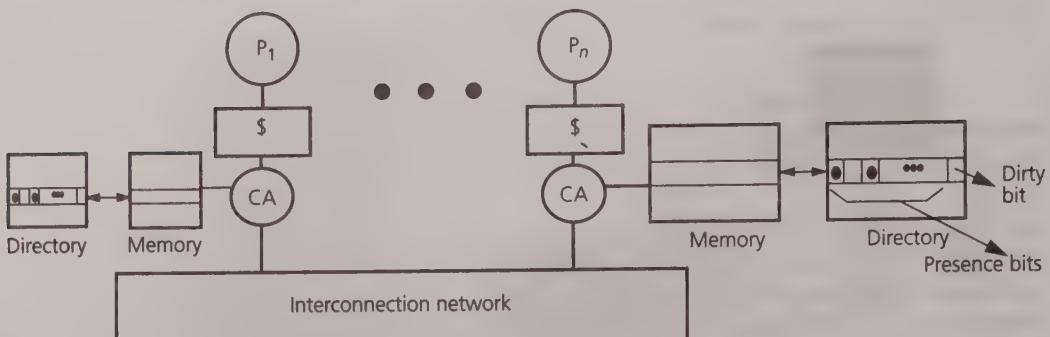


FIGURE 8.6 Directory information for a distributed-memory multiprocessor. In simple organization, the directory entry for a block is a vector of p presence bits, one for each node, and a dirty bit indicating whether any node has the block in modified state.

caches. Of course, if the dirty bit is ON, then only one node (the dirty node) should be caching that block and only that node's presence bit should be ON. With this structure, a read miss can easily determine from looking up the directory which node, if any, has a dirty copy of the block or if the block is valid in main memory at the home, and a write miss can determine which nodes are the sharers that must be invalidated.

The directory information for a block is simply main memory's view of the cache state of that block in different caches. The directory does not necessarily need to know the exact state (e.g., MESI) in each cache but only enough information to determine what actions to take. The number of states at the directory is therefore typically smaller than the number of cache states. In fact, since the directory and the caches communicate through a distributed interconnect, there will be periods when a directory's knowledge of a cache state is incorrect since the cache state has been modified but notice of the modification has not reached the directory. During this time, the directory may send a message to the cache based on its old (no longer valid) knowledge. The race conditions caused by this distribution of state make directory protocols interesting, and we see how they are handled using transient states or other means in Sections 8.4 through 8.6.

To see in greater detail how a read miss and write miss might interact with this bit vector directory organization, consider a protocol with three stable cache states (MSI), a single level of cache per processor, and a single processor per node. The protocol is orchestrated by the assists, which are also referred to as *coherence controllers* or *directory controllers*. On a read miss or a write miss at node i (including an upgrade from shared state), the local communication assist or controller looks up the address of the memory block to determine if the home is local or remote. If it is remote, a network transaction is sent to the home node for the block. There, the directory entry for the block is looked up, and the assist at the home may treat the miss as follows, using network transactions similar to those that were shown in Figure 8.5 (other, more optimized treatments are discussed later in the chapter):

- If the dirty bit is OFF, then the assist obtains the block from main memory, supplies it to the requestor in a reply network transaction, and turns the i th presence bit, $\text{presence}[i]$, ON.
- If the dirty bit is ON, then the assist responds to the requestor with the identity of the node whose presence bit is ON (i.e., the owner or dirty node). The requestor then sends a request network transaction to that owner node. At the owner, the cache changes its state to shared and supplies the block to both the requesting node, which stores the block in its cache in shared state, as well as to main memory at the home node. At memory, the dirty bit is turned OFF, and $\text{presence}[i]$ is turned ON.

A write miss by processor i goes to memory and is handled as follows:

- If the dirty bit is OFF, then main memory has a clean copy of the data. Invalidations must be sent to all nodes j for which $\text{presence}[j]$ is ON. Assuming a strict request-response scenario, as in Figure 8.5, the home node supplies the block to the requesting node i together with the presence bit vector. The directory entry is cleared, leaving only $\text{presence}[i]$ and the dirty bit ON. (If the request is an upgrade instead of a read exclusive, an acknowledgment containing the bit vector is returned to the requestor instead of the data itself.) The assist at the requestor sends invalidation requests to the required nodes and waits for invalidation acknowledgment transactions from the nodes, indicating that the write has completed with respect to them. Finally, the requestor places the block in its cache in dirty state.
- If the dirty bit is ON, then the block is first recalled from the dirty node (whose presence bit is ON), using network transactions with the home and the dirty node. That cache changes its state to invalid, and then the block is supplied to the requesting processor, which places the block in its cache in dirty state. The directory entry is cleared, leaving only $\text{presence}[i]$ and the dirty bit ON.

On a replacement of a dirty block by node i , the dirty data being replaced is written back to main memory, and the directory is updated to turn off the dirty bit and $\text{presence}[i]$. (As in bus-based machines, write backs cause interesting race conditions that are discussed later in the context of real protocols.) Finally, if a block in shared state is replaced from a cache, a message may or may not be sent to the directory to turn off the corresponding presence bit so an invalidation is not sent to this node the next time the block is written. This message is called a *replacement hint*; whether it is sent or not does not affect the correctness of the protocol or the execution.

A directory scheme similar to this one was introduced as early as 1978 (Censier and Feautrier 1978). It was designed for use in systems with a few processors and a centralized main memory and was used in the S-1 multiprocessor project at Lawrence Livermore National Laboratories (Widdoes and Correll 1980). However, directory schemes in one form or another were in use even before this. The earliest scheme was used in IBM mainframes, which had a few processors connected to a centralized memory through a high-bandwidth switch rather than a bus. With no broadcast medium to snoop on, a duplicate copy of the cache tags for each processor was maintained at the main memory, and it served as the directory. Requests coming

to the memory looked up all the tags to determine the states of the block in the different caches (Tang 1976; Tucker 1986). Of course, the tag copies at main memory had to be kept up-to-date. Since the directory was centralized in these early schemes, they are called *centralized directory schemes*.

The value of directories is that they keep track of which nodes have copies of a block, eliminating the need for broadcast. This is clearly very valuable on read misses since a request for a block will either be satisfied at the main memory or the directory will tell it exactly where to go to retrieve the exclusive copy. On write misses, the value of directories over the simpler broadcast approach is greatest if the number of sharers of the block (to which invalidations or updates must be sent) is usually small and does not scale up quickly with the number of processing nodes.

We might already expect the typical number of sharers to be small from our understanding of parallel applications. For example, in a near-neighbor grid computation, usually two, and at most four, processes should share a block at a partition boundary, regardless of the grid size or the number of processors. Even when a location is actively read and written by all processes in an application, the number of sharers to be invalidated at a write depends upon the temporal interleaving of reads and writes by processors. A common example is *migratory* data, which is read and written by one processor, then read and written by another processor, and so on (for example, a global sum into which processes accumulate their values). Although all processors read and write the location, only one other processor on a write—the previous writer—has a valid copy and must be invalidated since all others were invalidated before the previous write.

Empirical measurements of program behavior show that the number of valid copies on most writes to shared data is indeed very small the vast majority of the time, that this number does not grow quickly with the number of processors used, and that the frequency of writes that generate many invalidations is very low. Such data for our parallel applications will be presented and analyzed in light of application characteristics in Section 8.3.1. (Note that even if all processors running the application have to be invalidated on most writes, directories are still valuable for writes if the application does not run on all nodes of the multiprocessor.) These facts are also promising for the scalability of directory-based approaches and help us understand how to organize directories cost-effectively.

8.2.2 Scaling

The main goal of using directory protocols is to allow cache coherence to scale beyond the number of processors that may be sustained by a bus. It is important to understand the scalability of directory protocols in terms of both performance and the storage overhead for directory information. A system with distributed memory and interconnect already provides good scalability of raw latency and bandwidth under well-behaved loads. The major performance scaling issues for a protocol are how the latency and bandwidth demands it presents to the system scale with the number of processors used. The bandwidth demands are governed by the number of network transactions generated per miss (multiplied by the frequency of misses) and

latency by the number of these transactions that are in the critical path of the miss. In turn, these quantities are affected both by the directory organization and by how well the flow of network transactions is optimized in the protocol (given a directory organization). Storage, however, is affected only by how the directory information is organized. For the simple bit vector organization, the number of presence bits needed scales linearly with both the number of processing nodes (p bits per memory block) and the amount of main memory (1 bit vector per memory block), leading to a potentially large storage overhead for the directory. With a 64-byte block size and 64 processors, the directory storage overhead as a fraction of nondirectory (i.e., data) memory is 64 bits (plus state bits) divided by 64 bytes, or 12.5%, which is not so bad. With 256 processors and the same block size, the overhead is 50%, and with 1,024 processors it is 200%! The directory overhead does not scale well, though it may be acceptable if the number of nodes visible to the directory at the target machine scale is not very large.

Fortunately, there are many other ways to organize directory information that improve the scalability of directory storage. The different organizations naturally lead to different high-level protocols with different ways of addressing the three protocol functions presented in Section 8.1 and different performance characteristics. The rest of this section lays out the space of directory organizations and briefly describes how individual read and write misses might be handled in straightforward protocols that use these organizations. The discussion assumes that no other cache misses are in progress at the time, hence no race conditions, so the directory and the caches are always encountered as being in stable states. Deeper protocol issues are discussed in Sections 8.4 through 8.6.

8.2.3 Alternatives for Organizing Directories

Since communication with cached copies is always done through network transactions, the real differentiation among approaches is in the first two functions of coherence protocols: finding the source of the directory information upon a miss and determining the locations of the relevant copies.

The two major classes of alternatives for finding the source of the directory information for a block are known as *flat directory schemes* and *hierarchical directory schemes*.

The simple directory scheme described earlier is a flat scheme. Flat schemes are so named because the source of the directory information for a block is in a fixed place, usually at the home that is determined from the address of the block; on a miss, a single request network transaction is sent directly to the home node to look up the directory (if the home is remote) regardless of how far away the home is.

In hierarchical schemes, the source of directory information is not known a priori. Memory is again distributed with the processors, but the directory information for each block is logically organized as a hierarchical data structure (a tree). The processing nodes, each with its portion of memory, are at the leaves of the tree. The internal nodes of the tree are simply hierarchically maintained directory information for the block: a node keeps track of whether each of its children has a copy of a

block. Upon a miss, the directory information for the block is found by traversing up the hierarchy level by level through network transactions until a directory node is reached that indicates its subtree has the block in the appropriate state. Thus, a processor that misses simply sends a search message up to its parent, and so on, rather than directly to the home node for the block with a single network transaction. The directory tree for a block is logical, not necessarily physical, and can be embedded in any general interconnection network. Every block has its own logical directory tree. In fact, in practice, every processing node in the system not only serves as a leaf node for the blocks it contains but also stores directory information as an internal tree node for other blocks.

In the hierarchical case, the information about locations of copies is also maintained through the hierarchy itself; copies are found and communicated with by traversing up and down the hierarchy guided by directory information. For example, a directory entry at a node may indicate not only whether its subtree has valid copies of the block but also if copies of blocks allocated within its subtree may exist beyond its subtree. In flat schemes, how this information about copies is stored varies considerably. At the highest level, flat schemes can be divided into two classes: memory-based schemes and cache-based schemes. *Memory-based schemes* store the directory information about all cached copies at the home node of the block. The basic bit vector scheme described earlier is memory based: the locations of all copies are discovered at the home, and they can be communicated with directly through point-to-point messages. In *cache-based schemes*, the information about cached copies is not all contained at the home but is distributed among the copies themselves. The home simply contains a pointer to one cached copy of the block. Each cached copy then contains a pointer to (or the identity of) the node that has the next cached copy of the block, in a distributed linked-list organization. The locations of copies are therefore determined by traversing this list via network transactions.

Figure 8.7 summarizes the taxonomy. Hierarchical directories have some potential advantages. For example, a read miss to a block whose home is far away in the interconnection network topology might be satisfied closer to the issuing processor if another copy is found nearby as the request traverses up and down the hierarchy, instead of going all the way to the home. In addition, requests from different nodes can potentially be combined at a common ancestor in the hierarchy, with only one request sent on from there. These advantages depend on how well the logical hierarchy matches the underlying physical network topology. However, instead of only a few point-to-point network transactions needed to satisfy a miss in many flat schemes, the number of network transactions needed to traverse up and down the hierarchy can be much larger, which tends to have much greater impact on performance than distance traversed in the network (since the endpoint cost of initiating and handling network transactions dominates the per-hop cost). Each transaction along the way needs to look up (and perhaps modify) the directory information at its destination node, making transactions more expensive. As a result, the latency and bandwidth characteristics of hierarchical directory schemes tend to be much worse than for flat schemes, and these organizations are not popular on modern systems. Hierarchical directories are not, therefore, discussed much in this chapter but

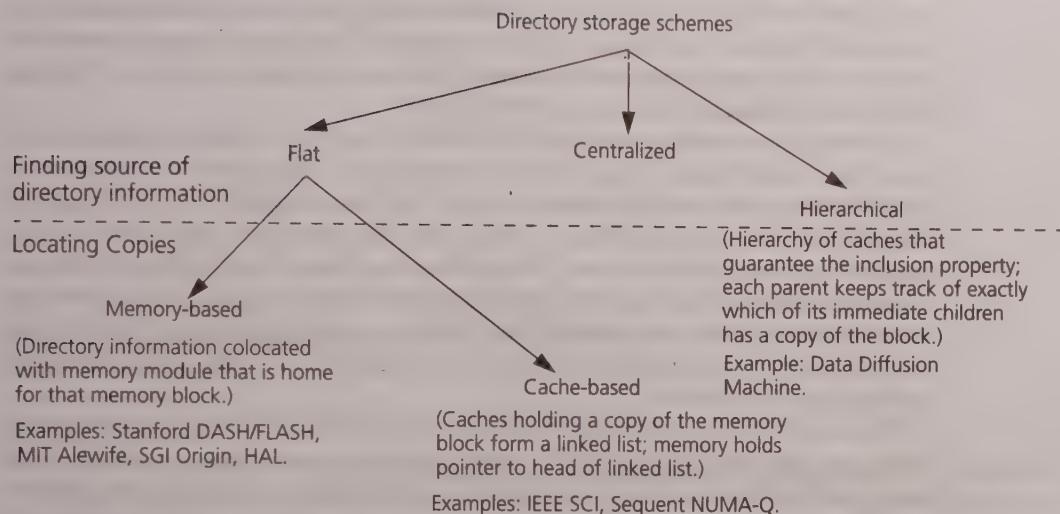


FIGURE 8.7 Alternatives for storing directory information. The two-level taxonomy is based on how the source of directory information and the copies themselves are located. In the hierarchical case, the same mechanism performs both functions.

are described briefly together with hierarchical snooping approaches in Section 8.10.2. The rest of this section examines flat directory schemes, both memory based and cache based, looking at directory organizations, storage overhead, the structure of protocols, and the impact on performance characteristics.

Flat, Memory-Based Directory Schemes

The bit vector organization described earlier, called a *full bit vector* organization, is the most straightforward way to store directory information in a flat, memory-based scheme. The style of protocol that results has already been discussed. Consider its basic performance characteristics on writes. Since it preserves information about sharers precisely and at the home, the number of network transactions per invalidating write grows only with the number of actual sharers. Because the identity of all sharers is available at the home, invalidations sent to them can be overlapped or even sent in parallel; the number of fully serialized network transactions in the critical path is thus not proportional to the number of sharers, reducing latency.

The main disadvantage of full bit vector schemes, as discussed earlier, is storage overhead. There are two ways to reduce this overhead for a given number of processors while still using full bit vectors. The first is to increase the cache block size. The second is to put multiple processors, rather than just one, in a node that is visible to the directory protocol; that is, to use a two-level protocol. For example, the Stanford DASH machine uses a full bit vector scheme, and its nodes are four-processor bus-based multiprocessors. These two methods actually make full bit vector directories

quite attractive for even fairly large machines: using four-processor nodes and 128-byte cache blocks, the directory memory overhead for a 256-processor machine is only 6.25%. As small-scale multiprocessors become increasingly attractive building blocks, this storage problem may not be severe.

However, these methods reduce the overhead by only a small constant factor each. The total directory storage is still proportional to $P \cdot M$, where P is the number of processing nodes and M is the number of total memory blocks in the machine ($M = P \cdot m$, where m is the number of blocks per local memory), and would become intolerable in very large machines. The overhead can be reduced further by addressing each of the factors in the $P \cdot M$ expression. We can reduce the number of bits per directory entry, or *directory width*, by not letting it grow proportionally to P . Or we can reduce the total number of directory entries, or *directory height*, by not having an entry per memory block.

Directory width is reduced by using what are called *limited pointer directories*, which are motivated by the earlier observation that most of the time only a few caches have a copy of a block when the block is written. Limited pointer schemes therefore do not store yes or no information for all nodes but simply maintain a fixed number of pointers (say, i), each pointing to a node that currently caches a copy of the block (Agarwal et al. 1988). Each pointer takes $\log P$ bits of storage for P nodes, but the number of pointers used is small. For example, for a machine with 1,024 nodes, each pointer needs 10 bits, so even having 100 pointers uses less storage than a full bit vector scheme. In practice, five or less pointers seem to suffice. Of course, these schemes need some kind of backup or overflow strategy for the situation when more than i readable copies are cached since they can keep track of only i copies precisely. One strategy is to resort to broadcasting invalidations to all nodes when there are more than i copies. Many other strategies have been developed to avoid broadcast even in these cases. Different limited pointer schemes differ primarily in their overflow strategies and in the number of pointers they use.

Directory height can be reduced by organizing the directory itself as a cache, taking advantage of the fact that since the total amount of cache in the machine is much smaller than the total amount of memory, only a very small fraction of the memory blocks will actually be present in caches at a given time, so most of the directory entries will be unused anyway (Gupta, Weber, and Mowry 1990; O’Kafka and Newton 1990). Section 8.10 discusses techniques reducing directory width and height in more detail.

Regardless of these storage-reducing optimizations, the basic approach to finding copies and communicating with them (protocol functions [2] and [3]) remains the same for the different flat, memory-based schemes. The identities of the sharers are maintained at the home and (at least when there is no overflow) the copies are communicated with by sending point-to-point transactions to each.

Flat, Cache-Based Directory Schemes

In flat, cache-based schemes, there is still a home main memory for the block; however, the directory entry at the home node does not contain the identities of all

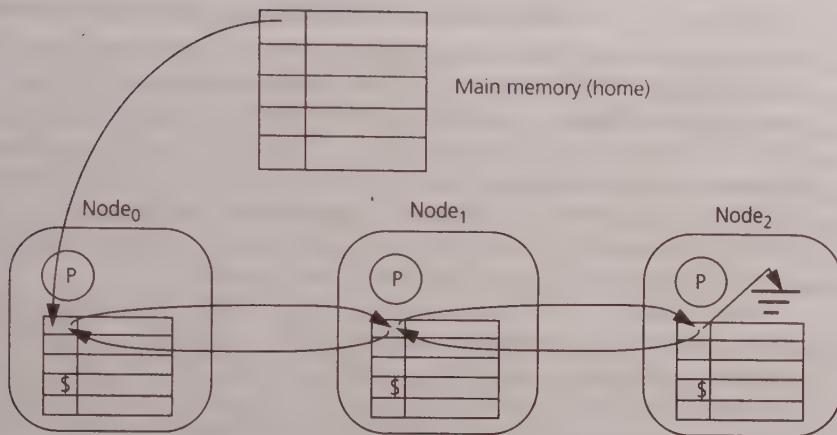


FIGURE 8.8 A doubly linked-list distributed directory organization. A cache line contains not only data and state for the block but also forward and backward pointers for the distributed linked list.

sharers but only a pointer to the first sharer in the list plus a few state bits. This pointer is called the *head pointer* for the block. The remaining nodes caching that block are joined together (using additional pointers that are associated with each cache line in a node) in a *distributed, doubly linked list*—that is, a cache that contains a copy of the block also contains pointers to the next and previous caches that have a copy, called the *forward* and *backward* pointers, respectively (see Figure 8.8).

On a read miss, the requesting node sends a network transaction to the home memory to find out the identity of the head node of the linked list, if any, for that block. If the head pointer is null (no current sharers), the home replies with the data. If the head pointer is not null, then the requestor must be added to the list of sharers. The home responds to the requestor with the head pointer. The requestor then sends a message to the head node, asking to be inserted at the head of the list and hence to become the new head node. The net effect is that the head pointer at the home now points to the requestor, the forward pointer of the requestor's own cache entry points to the old head node (which is now the second node in the linked list), and the backward pointer of the old head node points to the requestor. The data for the block is provided by the home if it has the latest copy or by the head node, which always has the latest copy (is the owner) otherwise.

On a write miss, the writer again obtains the identity of the head node, if any, from the home. It then inserts itself into the list as the head node as before (if the writer was already in the list as a sharer and is now performing an upgrade, it is deleted from its current position in the list and inserted as the new head). Following this, the rest of the distributed linked list is traversed node by node via network transactions to find and invalidate successive copies of the block. If a block that is written is shared by three nodes A, B, and C, the home only knows about A so the writer sends an invalidation message to it; the identity of the next sharer B can only

be known once A is reached, and so on. Acknowledgments for these invalidations are sent to the writer. Once again, if the data for the block is needed by the writer, it is provided by either the home or the head node as appropriate. The number of messages per invalidating write—the bandwidth demand—is proportional to the number of sharers as in the memory-based schemes, but now so is the number of messages in the critical path, that is, the latency. Each of these serialized messages invokes the communication assist at its destination, increasing latency and overall assist occupancy further. In fact, even a read miss to a clean block involves the assists of three nodes to insert the node in the linked list.

Write backs or other replacements from the cache also require that the node delete itself from the sharing list, which means communicating with the nodes that are before and after it in the list. This is necessary because the new block that replaces the old one in the cache will need the forward and backward pointer slots of the cache entry for its own sharing list. Synchronization is required to avoid simultaneous replacement of adjacent nodes in the list, and the involvement of multiple nodes increases overall assist occupancy. An example cache-based protocol is described in more depth in Section 8.6.

To counter the latency and occupancy disadvantages, cache-based schemes have some important advantages over memory-based schemes. First, the directory overhead is small. Every block in main memory has only a single head pointer. The number of forward and backward pointers is proportional to the number of cache blocks in the machine, which is much smaller than the number of memory blocks. The second advantage is that a linked list records the order in which accesses were made to memory for the block, thus making it easier to provide fairness and to avoid livelock in a protocol (most memory-based schemes do not keep track of request order, as we will see). Third, the work to be done by assists in sending invalidations is not centralized at the home but rather distributed among sharers, thus perhaps spreading out assist occupancy and reducing the corresponding bandwidth demands placed on a particularly busy home assist.

Manipulating insertion in and deletion from distributed linked lists can lead to complex protocol implementations. For example, deleting a node from a sharing list requires careful coordination and mutual exclusion with processors ahead of and behind it in the linked list since those processors may also be trying to replace the same block concurrently. These complexity issues have been greatly alleviated by the formalization and publication of a standard for a cache-based directory organization and protocol: the IEEE 1596-1992 Scalable Coherent Interface (SCI) standard (Gustavson 1992). The standard includes a full specification and C code for the protocol. Several commercial machines use this protocol (e.g., Sequent NUMA-Q [Lovett and Clapp 1996], Convex Exemplar [Convex Computer Corporation 1993; Thekkath et al. 1997], and Data General [Clark and Alnes 1996]), and variants that use alternative list representations (singly linked lists instead of the doubly linked lists in SCI) have also been explored (Thapar and Delagi 1990). We shall examine the SCI protocol itself in more detail in Section 8.6 and so defer detailed discussion of advantages and disadvantages.

Summary of Directory Organization Alternatives

To summarize, there are many different ways to organize how directories store the cache state of memory blocks. Simple bit vector representations work well for machines that have a moderate number of nodes visible to the directory protocol. For larger machines, many alternatives are available to reduce the memory overhead. The organization chosen does, however, affect the complexity of the coherence protocol and the performance of the directory scheme against various sharing patterns. Hierarchical directories have not been popular on real machines, whereas machines with flat memory-based and cache-based (linked-list) directories have been built and used for some years now.

The next section quantitatively assesses the behavior of parallel programs and the implications for directory-based approaches as well as some important protocol and architectural trade-offs at this basic level.

8.3 ASSESSING DIRECTORY PROTOCOLS AND TRADE-OFFS

As in Chapter 5, this section uses a simulator to examine some relevant characteristics of applications that can inform architectural trade-offs but that cannot be measured on real machines. Issues such as three-state versus four-state or invalidation-based versus update protocols that were discussed in Chapter 5 are not revisited here. The focus is on invalidation-based protocols, since update protocols have an additional disadvantage in scalable machines: useless updates incur a separate network transaction for each destination rather than a single bus transaction that is snooped by all caches. In addition, update-based protocols make it much more difficult to preserve the desired memory consistency model in directory-based systems. This section quantifies the distribution of invalidation patterns in directory protocols, examines how the distribution of traffic between local and remote changes as the number of processors is increased for a fixed problem size, and revisits the impact of cache block size on traffic. In all cases, the experiments assume a memory-based flat directory protocol.

8.3.1 Data Sharing Patterns for Directory Schemes

It was claimed earlier that the number of invalidations that need to be sent out on a write is usually small, which makes directories especially valuable and can reduce directory storage overhead without hurting performance. This subsection quantifies that claim for our parallel application case studies. It also develops a framework for categorizing data structures in terms of sharing patterns and understanding how the invalidation patterns scale, and explains the behavior of the application case studies in light of this framework. The simulated protocol assumes only the three basic cache states (MSI) for simplicity.

Sharing Patterns for Application Case Studies

For invalidation-based directory protocols, it is important to understand two aspects of an application's data sharing patterns: (1) the frequency with which processors issue writes that may require invalidating other copies (i.e., writes to data that is not in modified state in the writer's cache in an MSI protocol, or *invalidating writes*), called the *invalidation frequency*; and (2) the distribution of the number of invalidations (sharers) needed upon these writes, called the *invalidation size distribution*. Directory schemes are particularly advantageous if the average invalidation size is small and the frequency is significant enough that using broadcast all the time would indeed be a performance problem. Figure 8.9 shows the invalidation size distributions for our parallel application case studies running on 64-node systems (one processor per node) for the default problem sizes presented in Chapter 4. Infinite per-processor caches are used in these simulations to capture inherent sharing patterns. With finite caches, replacement hints sent to the directory may turn off presence bits and reduce the number of invalidations sent on writes in some cases (though traffic will not be reduced since the replacement hints have to be sent). A write may send zero invalidations in an MSI protocol if the block was loaded in shared state but there are currently no other sharers. This would not happen with infinite caches in a MESI protocol. With infinite caches, invalidation frequency is proportional to the communication-to-computation ratio.

It is clear that the invalidation sizes are usually small, indicating both that directories are indeed likely to be very useful in containing traffic and that it is not necessary for the directory to maintain a presence bit per processor in a flat memory-based scheme. The nonzero frequencies of very large invalidation sizes are usually due to synchronization variables, where many processors spin on a variable and one processor writes it, invalidating them all. We are interested not just in the results for a given problem size and number of processors but also in how they scale. The communication-to-computation ratios discussed in Chapter 4 give us a good idea about how the frequency of invalidating writes should scale. For the size distributions, we can appeal to our understanding of applications and their usage of data structures (and validate with experiments), which can also help explain the basic results observed in Figure 8.9.

A Framework for Sharing Patterns

Data access patterns in applications can be categorized in many ways: predictable versus unpredictable, regular versus irregular, coarse-grained versus fine-grained (or contiguous versus noncontiguous in the address space), near-neighbor versus long-range in an interconnection topology, and so on. For understanding invalidation patterns, the relevant categories are read-only, producer-consumer, migratory, and irregular read-write. (A similar categorization can be found in [Gupta and Weber 1992].)

- **Read-only.** Read-only data structures are never written once they have been initialized. There are no invalidating writes, so data in this category is not an

issue for directories. Examples include program code and the scene data in the Raytrace application.

- *Producer-consumer*. A processor produces (writes) a data item, then one or more processors consume (read) it, then a processor produces it again, and so on. Flag-based synchronization is an example, as is the near-neighbor sharing in an iterative grid computation. The producer may be the same process every time or it may change; for example, in a branch-and-bound algorithm, the bound variable may be written by different processes as they find improved bounds. The invalidation size for this category is determined by how many consumers there have been each time the producer writes the value. We can have situations with one consumer, all processes being consumers, or a few processes being consumers. These situations may have different frequencies and scaling properties, although for most applications either the size does not scale quickly with the number of processors or the frequency has been found to be low.¹
- *Migratory*. Migratory data bounces around, or migrates, from one processor to another, being written (and usually read) by each processor to which it bounces. An example is a global sum, into which different processes add their partial sums. Each time a processor writes the variable, only the previous writer has a copy (since it invalidated the previous “owner” when it did its write); so only a single invalidation is generated upon a write, regardless of the number of processors used.
- *Irregular read-write*. This corresponds to irregular or unpredictable read and write access patterns to data by different processes. A simple example is a distributed task-queue system. Processes will probe (read) the head pointer of a task queue when they are looking for work to steal, and this head pointer will be written when a task is added at the head. These and other irregular patterns usually lead to wide-ranging invalidation size distributions, but in most observed applications the frequency concentration tends to be very much toward the small end of the spectrum (see the Radiosity example in Figure 8.9).

1. Examples of the producer-consumer size distribution not scaling up are the noncorner border elements in a near-neighbor regular grid partition and the key permutations in Radix. They lead to an invalidation size of one, which does not increase with the number of processors or the problem size. Examples of all processes being consumers (invalidation size $p - 1$) are a global energy variable that is read by all processes during a time-step of a physical simulation and then written by one at the end or a synchronization variable on which all processes spin. While the invalidation size here is large, such writes fortunately tend to happen very infrequently in real applications. Finally, examples of a few processes being consumers are the corner elements of a grid partition or the flags used for tree-based synchronization. This leads to an invalidation size of a few, which may or may not scale with the number of processors (it doesn't in these two examples).

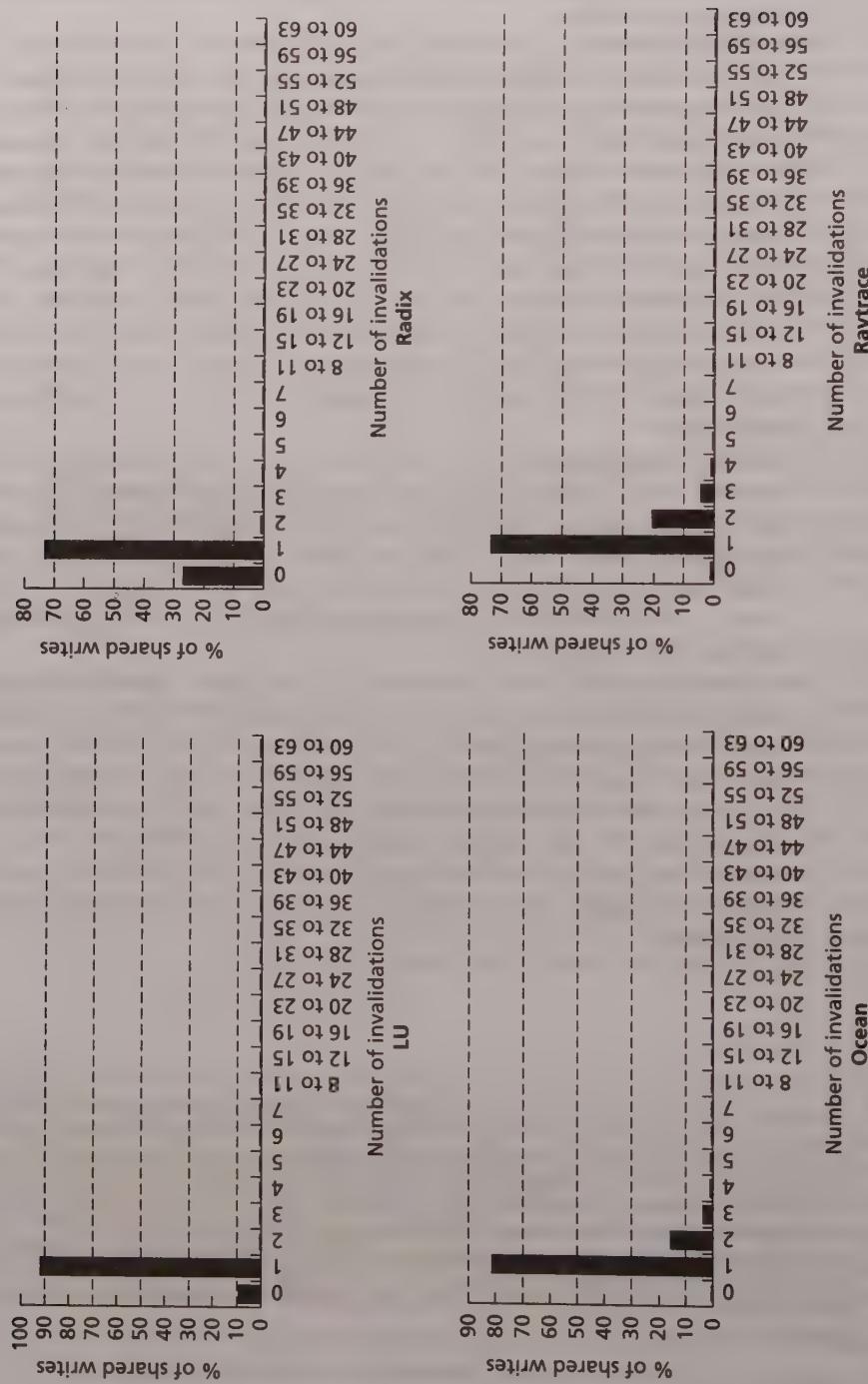


FIGURE 8.9 Invalidation patterns with default data sets and 64 processors

(continued)

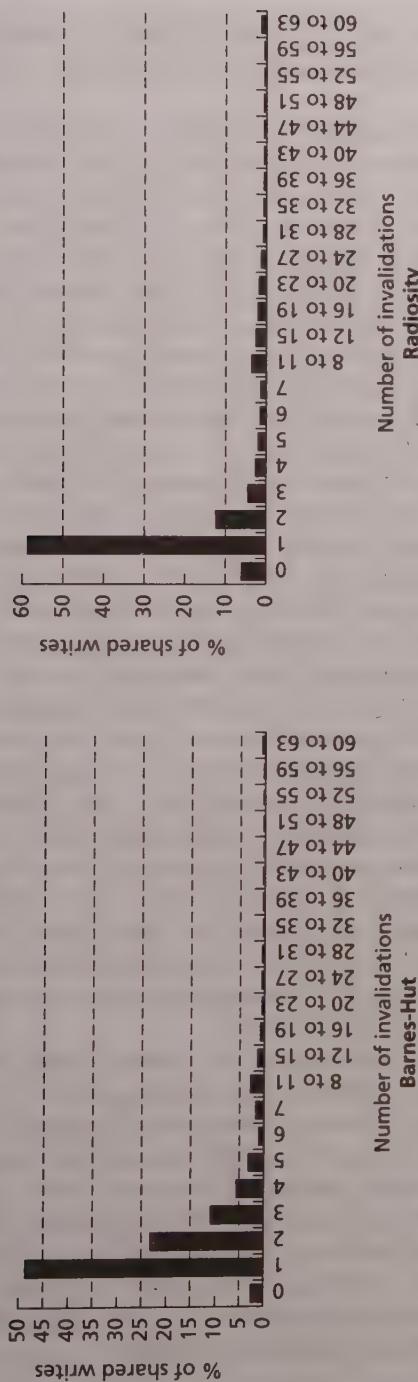


FIGURE 8.9 Invalidation patterns with default data sets and 64 processors. The x-axis shows the invalidation size (number of active sharers) upon an invalidating write, and the y-axis shows the percentage of invalidating writes whose size takes that x-axis value. The invalidation size distribution was measured by simulating a full-bit vector directory representation and recording for every invalidating write how many presence bits (other than the writer's) are set when the write occurs. The data is averaged over all the 64 processors used.

Applying the Framework to the Application Case Studies

Let us now look briefly at each of the applications in Figure 8.9 to interpret the results in light of these four sharing patterns and to understand how the size distributions might scale.

In the LU factorization program, when a block is written it has previously been read only by the same processor that is doing the writing (the process to which it is assigned). This means that no other processor should have a cached copy, and zero invalidations should be sent. Once it is written, it is read by several other processes and no longer written further. The reason that we see one invalidation being sent in the figure is that the matrix is initialized by a single process; particularly with the infinite caches we use, that process has a copy of the entire matrix in its cache and will be invalidated the first time another processor does a write to a block. An insignificant number of invalidating writes invalidates all processes, which is due to some global variables and not the main matrix data structure. Scaling the problem size or the number of processors does not change the invalidation size distribution for the matrix but only for the global variables. Of course, the invalidation frequencies do change with scaling, just like the communication-to-computation ratio.

In the Radix sorting kernel, invalidations are sent in two producer-consumer situations. In the permutation phase, the word or block written has been read since the last write only by the process to which that key is assigned, so at most a single invalidation is sent out. The same key position in the destination array may be written by different processes in different outer loop iterations of the sort; however, in each iteration there is only one reader of a key, so even this infrequent case generates only two invalidations (one to the reader and one to the previous writer). If there is false sharing, all sharers are writing the block, so there is only one invalidation each time. The other situation that generates invalidations is the histogram accumulation, which is done in a tree-structured fashion and usually leads to a small number of invalidations at a time. These invalidations to multiple sharers are clearly very infrequent. In Radix too, increasing the problem size does not change the invalidation size in either phase (though it may change the relative invalidation frequencies in the two phases), whereas increasing the number of processors increases the sizes but only in some infrequent parts of the histogram accumulation phase. The dominant pattern by far remains 0 or 1 invalidations.

The nearest-neighbor, producer-consumer communication pattern on a regular grid in Ocean leads to most of the invalidations being sent to 0 or 1 processes (at the borders of a partition). At partition corners, more frequently encountered in the multigrid equation solver, two or three sharers may need to be invalidated. This does not grow with problem size or number of processors. At the highest levels of the multigrid hierarchy, the border elements of a few processors' partitions might fall on the same cache block, causing four or five sharers to be invalidated. There are also some global accumulator variables, which display a migratory sharing pattern (1 invalidation), and a couple of very infrequently used one-producer, all-consumer global variables (other than synchronization variables).

In Raytrace the dominant data structure is the scene data, which is read-only. The major read-write data consists of the image and the task queues. Each word in the image is written only once by one processor per frame. This leads to either 0 invalidations if the same processor writes a given image pixel in consecutive frames (as is usually the case) or 1 invalidation if different processors do, as might be the case when tasks are stolen or if there is write-write false sharing. The task queues themselves lead to the irregular read-write access patterns discussed earlier, with a wide-ranging distribution that is dominated in frequency by the low end (hence the very small nonzeros all along the x -axis in this case). Here too we find some infrequently written one-producer, all-consumer global variables.

In the Barnes-Hut application, the important data is the body and cell positions, the pointers used to link up the tree, and some global variables used as energy values. The position data is of the producer-consumer type. A given body's position is usually read by one or a few processors during the force calculation (tree traversal) phase. The positions (centers of mass) of the cells are read by many processes, the number increasing toward the root, which is read by all. This data thus causes a fairly wide range of invalidation sizes when it is written by its assigned processor in the update and tree construction phases that follow force calculation. The root and upper-level cells are responsible for invalidations being sent to all processors, but their frequency is quite small. The tree pointers are similar in their behavior to the cell centers of mass. The first write to a pointer in the tree-building phase invalidates the caches of the processors that read it in the previous force calculation phase; subsequent writes invalidate those processors that have read the pointer during the current tree-building phase, which is an irregularly sized but mostly small set of processors. As the number of processors is increased, the invalidation size distribution tends to shift to the right as more processors tend to read a given item, but the shift is slow and the dominant invalidations are still to a small number of processors. The reverse effect (also slow) is observed when the number of bodies is increased.

Finally, the Radiosity application has very irregular access patterns to many different types of data, including data that describes the scene (patches and elements) and the task queues. The resulting invalidation patterns show a wide distribution; however, even here the greatest frequency by far is concentrated toward 0 to 2 invalidations. Many of the accesses to the scene data behave in a migratory way, as do a few counters, and a couple of global variables are one-producer, all-consumer.

The empirical data and categorization framework indicate that in most cases the invalidation size distribution is dominated by small numbers of invalidations. The common use of parallel machines as multiprogrammed compute servers for sequential or small-way parallel applications further limits the number of sharers (process migration usually leads to invalidations of size 1). Sharing patterns that cause large numbers of invalidations are empirically found to be very infrequent at run time. A possible exception is highly contended synchronization variables, which are usually handled specially by software or hardware, as we shall see. In addition to validating the directory-based approach and suggesting its potential for performance scalability, these results suggest that limited-pointer directory representations should be successful since the frequency of overflows will be small.

8.3.2 Local versus Remote Traffic

A key property for systems with distributed memory is how much of the traffic due to cache misses or protocol activity is kept within a node (local) rather than going on the interconnect (remote). For a given number of processors and machine organization, the fraction of traffic that is local depends on the problem size. However, it is instructive to examine how the traffic and its distribution change with the number of processors even when the problem size is held fixed (i.e., under PC scaling). Figure 8.10 shows the results for the default problem sizes, breaking down the remote traffic into various categories such as sharing (true or false), capacity, cold start, write back, and overhead. A MESI rather than MSI protocol is used in this and the next subsection. Overhead includes the fixed header sent across the network with each cache block of data as well as the traffic associated with protocol transactions like invalidations and acknowledgments that do not carry any data. This protocol traffic component is different from that on a bus-based machine: each individual point-to-point invalidation consumes traffic, and acknowledgments place traffic on the interconnect too. Traffic is shown in bytes per FLOP or bytes per instruction for different applications.

We can see that both local traffic and capacity-related remote traffic tend to decrease when the number of processors increases, due to both decrease in per-processor working sets and decrease in cold misses that are satisfied locally instead of remotely. However, sharing-related traffic increases as expected. In applications with small working sets, like Barnes-Hut, LU, and Radiosity, the fraction of capacity-related traffic is very small, at least beyond a couple of processors. In irregular applications like Barnes-Hut and Raytrace, most of the capacity-related traffic is remote, all the more so as the number of processors increases, since data cannot be distributed easily at page granularity for capacity misses to be satisfied locally. In cases like Ocean, the capacity-related traffic is substantial even with the large cache and is almost entirely local when pages are placed properly (which can be done quite easily with 4D array data structures). With round-robin placement of shared pages, we would have seen most of the local capacity misses in Ocean turn to remote ones.

When we use smaller caches to capture the realistic scenario of working sets not fitting in the cache in Ocean and Raytrace, capacity traffic becomes much larger. In Ocean, most of this traffic is still local due to good data distribution, and the trend for remote traffic versus number of processors doesn't change. Poor distribution of pages would have swamped the network with traffic, but with proper distribution, remote traffic is quite low. In Raytrace, however, the capacity-related traffic is mostly remote, and the fact that it now dominates changes the slope of the curve of total remote traffic compared to that with large caches, where sharing traffic dominates. Remote traffic still increases with the number of processors but much more slowly since the working set size and, hence, capacity miss rate does not depend as much on the number of processors as the sharing miss rate.

When a miss is satisfied remotely, whether it is satisfied at the home or it needs another message to obtain the data from a dirty node depends not only on whether it is a sharing miss or a capacity/conflict/cold miss but also on the size of the cache. In

a small cache, dirty data may be replaced and written back, so a sharing miss by another processor may be satisfied at the home node rather than at the previously dirty node. For applications like Ocean that allow data to be placed easily in the memory of the node to which they are assigned (i.e., to be appropriately distributed for locality), it is often the case that only that node writes the data, so even if the data is found dirty, it is found so in a cache at the home node itself. The extent to which this is true depends on the data access patterns of the application, the granularity of data allocation in memory, and whether the data is indeed distributed properly by the program.

8.3.3 Cache Block Size Effects

The effects of block size on cache miss rates and bus traffic were assessed in Chapter 5, at least up to 16 processors. For miss rates, the trends beyond 16 processors extend quite naturally, except for threshold effects in the interaction of problem size, number of processors, and block size, as discussed in Chapter 4. This section examines the impact of block size on the components of local and remote traffic in machines with distributed memory.

Figure 8.11 shows how traffic scales with block size for 32-processor executions of the applications with 1-MB caches per processor. In Barnes-Hut, the overall traffic increases slowly until about a 64-byte block size and more rapidly thereafter primarily due to false sharing. However, the amount of traffic is small. Since the overhead per block moved through the network is fixed (as is the cost of invalidations and acknowledgments), the overhead component tends to shrink with increasing block size to the extent that there is spatial locality (i.e., if larger blocks reduce the number of blocks transferred). LU has perfect spatial locality, so the data traffic remains fixed as block size increases. Overhead is reduced, so overall traffic in fact shrinks with increasing block size. In Raytrace, the remote capacity traffic has poor spatial locality, so it grows quickly with block size. In both Barnes-Hut and Raytrace, the true sharing traffic has poor spatial locality too, as is the case in Ocean at column-oriented partition borders (spatial locality even on remote data is good at row-oriented borders). Finally, the graph for Radix clearly shows the impact of false sharing on remote traffic when it occurs past the threshold block size (here about 128 or 256 bytes). Results with smaller caches show the behavior of capacity misses playing a dominant role, as expected.

8.4

DESIGN CHALLENGES FOR DIRECTORY PROTOCOLS

Designing a correct, efficient directory protocol involves issues that are more complex and subtle than the simple organizational choices we have discussed so far, just as designing a bus-based protocol was more complex than choosing the number of states and drawing the state transition diagram for stable states. We had to deal with the nonatomicity of state transitions, split-transaction buses, serialization and ordering issues, deadlock, livelock, and starvation. Now that we understand the basics of

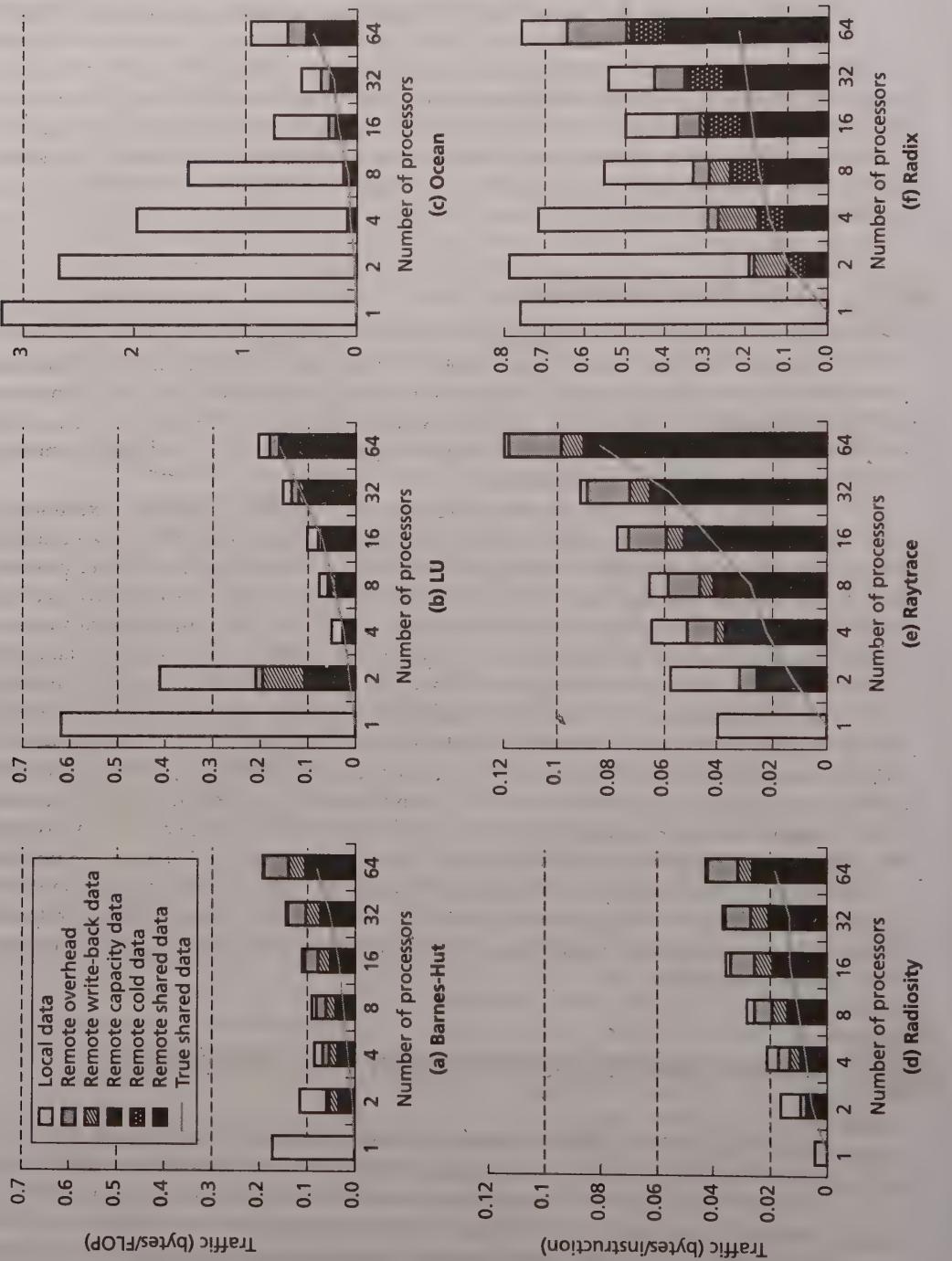


FIGURE 8.10 Traffic versus number of processors

(continued)

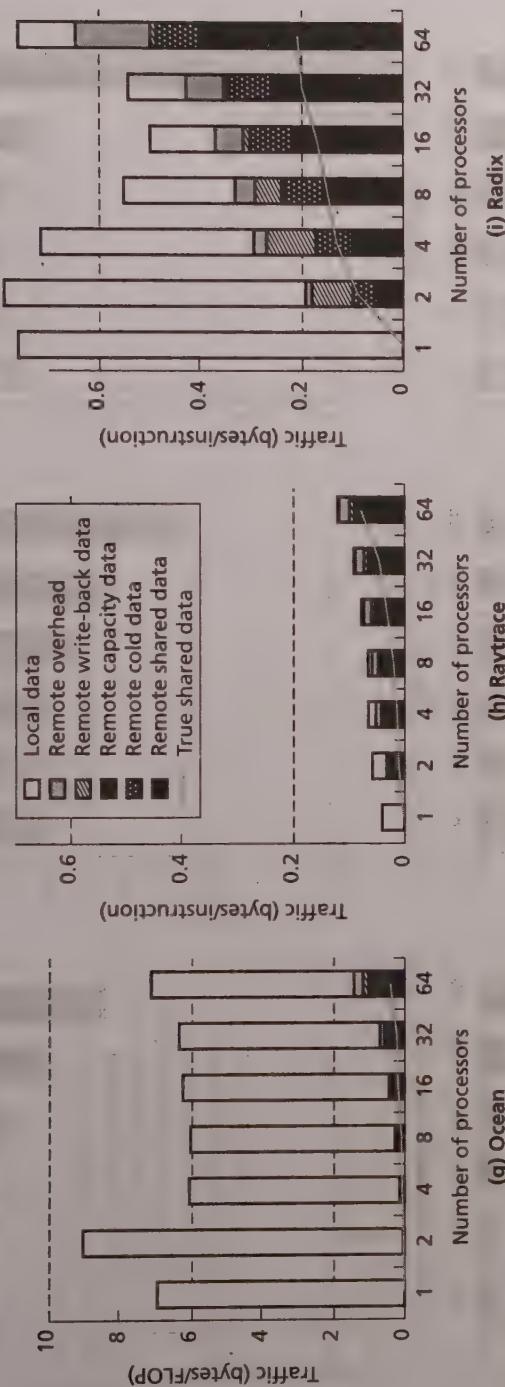


FIGURE 8.10 Traffic versus number of processors. The overhead per block transferred across the network is 8 bytes. No overhead is considered for local accesses. Graphs (a)–(f) assume 1-MB caches per processor and graphs (g)–(i) 64 KB. The caches have a 64-byte block size and are four-way set associative.

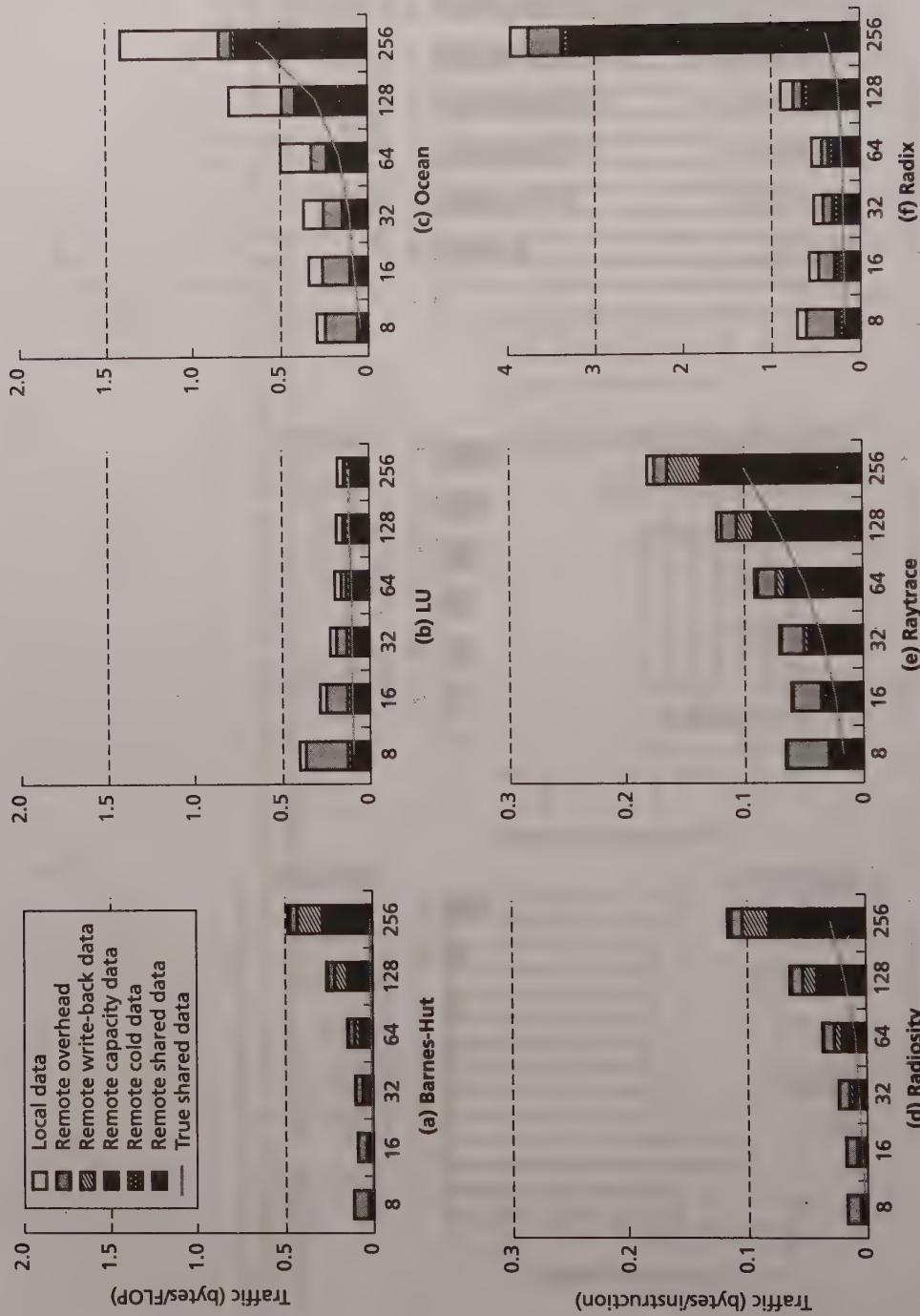


FIGURE 8.11 Traffic versus cache block size

(continued)

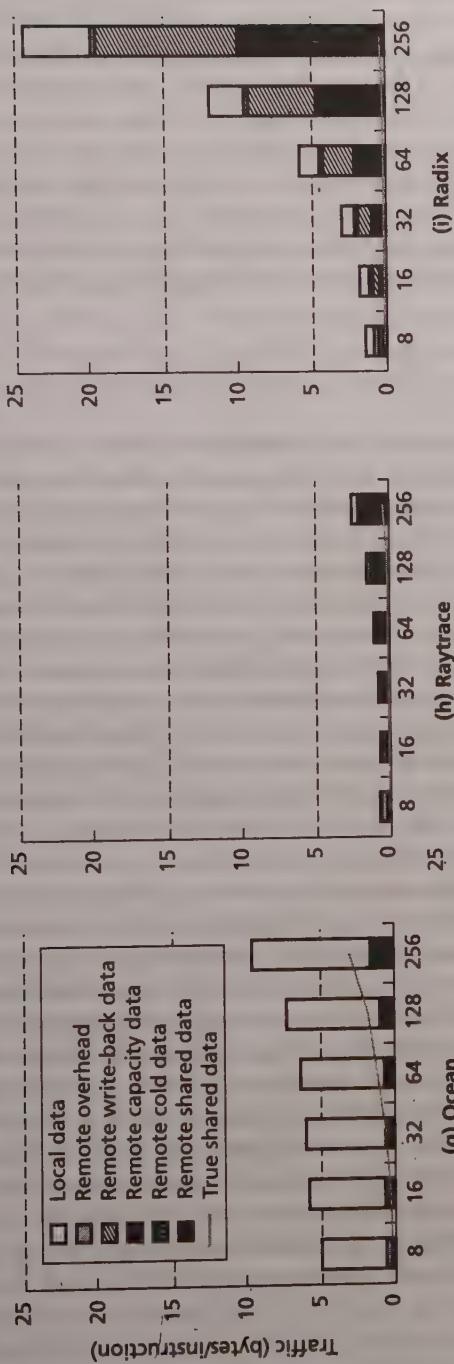


FIGURE 8.11 Traffic versus cache block size. The overhead per block transferred over the network is 8 bytes. No overhead is considered for local access. Graphs (a)–(f) show the data for 1-MB caches per processor and (g)–(i) for 64-KB caches. All caches are four-way set associative.

directories, we are ready to dive into these issues for them as well. This section discusses the new protocol-level design challenges that arise in correctly implementing directory protocols for high performance and identifies general techniques for addressing these challenges. In the next two sections, the techniques are specialized in the case studies of memory-based and cache-based directory protocols.

As always, the design challenges for scalable coherence protocols are to provide high performance while preserving correctness and to contain the complexity that results. Let us look at performance and correctness in turn, focusing on issues that were not already addressed for bus-based or noncaching systems. Since performance optimizations tend to increase concurrency and complicate correctness, let us examine them first.

8.4.1 Performance

The network transactions on which cache coherence protocols are built differ from those used in explicit message passing in two ways. First, they are automatically generated by the system—in particular, by the communication assists or controllers—in accordance with the protocol. Second, they are individually small, each carrying either a request, an acknowledgment, or a cache block of data plus some control bits. However, the basic performance model for network transactions developed in earlier chapters applies here as well. A typical network transaction incurs some overhead on the processor at its source (traversing the cache hierarchy on the way out and back in); some work or occupancy on the communication assists at its endpoints (typically looking up state, generating requests, or intervening in the cache); and some delay in the network due to transit latency, network bandwidth, and contention. Typically, the processor itself is not directly involved at the home, the dirty node, or the sharers but only at the requestor (although it may suffer at the other nodes as well due to contention).

It is useful to understand performance in terms of the layers of a multiprocessor system introduced earlier (Figure 8.2). The protocol layer of a system implements the programming model, using the network transactions provided by the communication abstraction. Thus, the protocol layer does not have much leverage on the basic communication costs of a single network transaction—transit latency, network bandwidth, assist occupancy, and processor overhead—but it can determine the number and structure of the network transactions needed to realize memory operations like reads and writes under different circumstances. In general, there are three classes of techniques for improving performance: (1) protocol optimizations, (2) high-level machine organization, and (3) hardware specialization to improve the basic communication parameters. The first two assume a fixed set of performance parameters for the communication architecture and are discussed in this section. The impact of varying the basic performance parameters will be examined in Section 8.7.

Protocol Optimizations

The two major performance goals at the protocol level are to reduce the number of network transactions generated per memory operation, which reduces the bandwidth demands placed on the network and the communication assists; and to reduce the number of actions, especially network transactions, that are on the critical path of the processor, thus reducing uncontended latency. The latter can be done by overlapping the transactions needed for a memory operation as much as possible. To some extent, protocol design can also help reduce the endpoint assist occupancy per transaction—especially when the assists are programmable—which reduces both uncontended latency as well as endpoint contention. The traffic, latency, and occupancy characteristics should not scale up quickly with the number of processing nodes used and should perform gracefully under pathological conditions like hot spots.

As we have seen, the manner in which directory information is stored determines the number of network transactions in the critical path of a memory operation. For example, a memory-based protocol can issue invalidations in an overlapped manner from the home whereas, in a cache-based protocol, the distributed list must be walked by network transactions to learn the identities of the sharers. However, even within a class of protocols, there are many ways to improve performance.

Consider a read miss to a remotely allocated block that is dirty in a third node in a flat, memory-based protocol. The strict request-response option described earlier is shown in Figure 8.12(a). The home responds to the requestor with a message containing the identity of the owner node. The requestor then sends a request to the owner, which replies to it with the data (the owner also sends a “revision” message to the home, which updates memory with the data and sets the directory state to be shared).

There are four network transactions in the critical path for the read operation and five transactions in all. One way to reduce these numbers is *intervention forwarding*. In this case, the home does not respond to the requestor but simply forwards the request as an intervention transaction to the owner, asking it to retrieve the block from its cache. An intervention is just like a request but is issued in reaction to a request and is directed at a cache rather than memory (it is similar to an invalidation in this sense but also seeks data from the cache). The owner then replies to the home with the data or an acknowledgment (if the block is in exclusive rather than modified state), at which time the home updates its directory state and replies to the requestor with the data (Figure 8.12[b]). Intervention forwarding reduces the total number of transactions needed to four, reducing bandwidth needs, but all four are still in the critical path. A more aggressive method is *reply forwarding* (Figure 8.12[c]). Here too, the home forwards the intervention message to the owner node, but the intervention contains the identity of the requestor and the owner replies directly to the requestor itself. The owner also sends a revision message to the home so that the memory and directory can be updated, but this message is not in the critical path of the read miss. This keeps the number of transactions at four but reduces

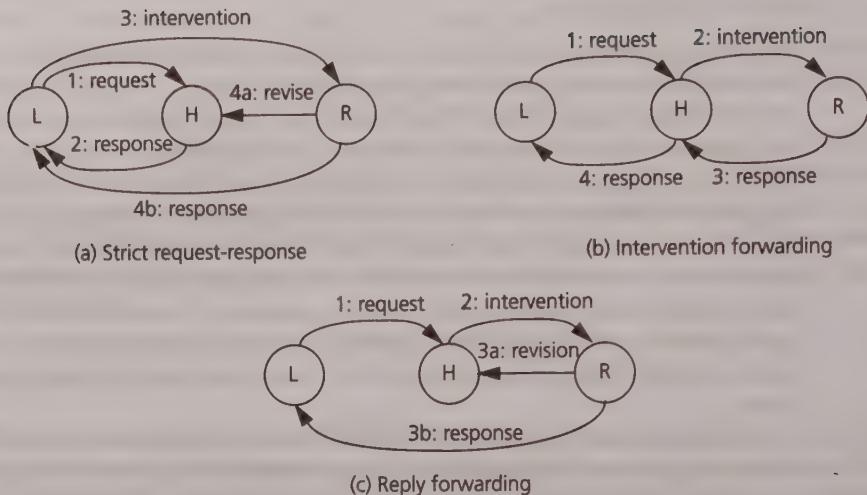


FIGURE 8.12 Reducing latency in a flat, memory-based protocol through forwarding. The case shown is of a read request to a block in exclusive state. L represents the local or requesting node, H is the home for the block, and R is the remote owner node that has the exclusive copy of the block.

the number in the critical path to three (request → intervention → reply-to-requestor); it is, therefore, called a *three-message miss*. Notice that with either of intervention forwarding or reply forwarding the protocol is no longer strictly request-response since a request to the home generates another request (to the owner node, which in turn generates a response). This can complicate deadlock avoidance, as we shall see later.

Besides being only intermediate in its latency and traffic characteristics, intervention forwarding has the disadvantage that outstanding intervention requests are kept track of at the home rather than at the requestor, since responses to the interventions are sent to the home. Because requests that cause interventions may come from any of the nodes, the home node must keep track of up to $k \cdot P$ interventions at a time, where k is the number of outstanding requests allowed per node. A requestor, on the other hand, would only have to keep track of at most k outstanding interventions. Reply forwarding does not require the home to keep track of outstanding requests and also has better performance characteristics, so systems prefer to use it. Similar forwarding techniques can be used to reduce latency in cache-based schemes at the cost of strict request-response simplicity, as shown in Figure 8.13.

In addition to forwarding, other protocol optimizations to reduce latency include overlapping transactions and activities by performing them speculatively. For example, when a request arrives at the home, the assist can read the data from memory in parallel with the directory lookup, in the hope that in most cases the block will indeed be clean at the home. If the directory lookup indicates that the block is dirty in some cache, then the memory access is wasted and must be ignored. Finally, pro-

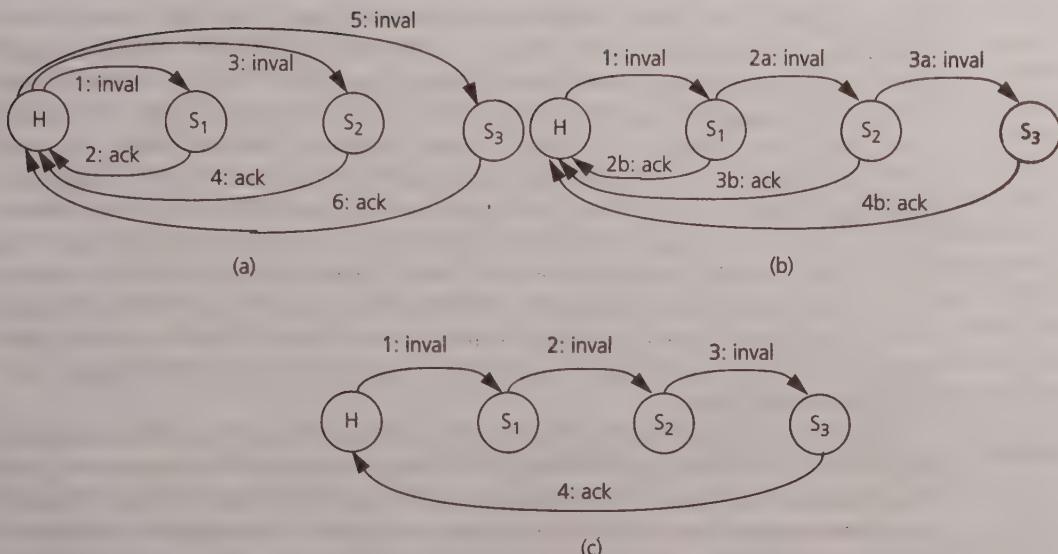


FIGURE 8.13 Reducing latency in a flat, cache-based protocol. In this scenario, invalidations are sent from the home H to the sharers S_i on a write operation. In the strict request-response case (a), every node includes in its acknowledgment (response) the identity of the next sharer on the list, and the home then sends that sharer an invalidation. The total number of transactions in the invalidation sequence is 2s, where s is the number of sharers and all are in the critical path. In (b), each invalidated node forwards the invalidation to the next sharer and in parallel sends an acknowledgment to the home. The total number of transactions is still 2s, but only s + 1 are in the critical path. In (c), only the last sharer on the list sends a single acknowledgment telling the home that the sequence is done. The total number of transactions is s + 1. (b) and (c) are not strict request-response cases.

Protocols may also automatically detect common sharing patterns to which the standard invalidation-based protocol is not ideally suited and adjust themselves at run time to interact better with these patterns (see Exercises 8.9 and 8.10).

High-Level Machine Organization

Machine organization can interact with the protocol to help improve performance as well. For example, the use of large tertiary caches within a node can reduce the number of protocol transactions generated by artifactual communication. For a fixed total number of processors, using multiprocessor rather than uniprocessor nodes in a two-level organization may be useful as well.

The potential advantages of a two-level organization are in both cost and performance. On the cost side, certain fixed per-node costs may be amortized among the processors within a node, and it is possible to use existing SMPs that may themselves be commodity parts. On the performance side, advantages may arise from sharing characteristics that reduce the number of accesses that involve the directory protocol and generate network transactions across nodes. If one processor brings a

block of data into its cache, another processor in the same node may be able to satisfy its miss to that block (for the same or a different word) more quickly through the local protocol using cache-to-cache sharing, especially if the block is allocated remotely. Requests may also be combined: if one processor has a request outstanding to the directory protocol for a block, another processor's request within the same SMP can be combined with, and obtain the data from, the first processor's response, reducing latency, network traffic, and potential hot spot contention. These advantages are similar to those of full hierarchical approaches and of shared caches. In fact, within an SMP, processors may even share a cache at some level of the hierarchy, in which case all the trade-offs for shared caches discussed in Chapter 6 apply. With fewer nodes, more of the main memory is local as well. Finally, cost and performance characteristics may be improved by using a hierarchy of packaging technologies appropriately.

Of course, the extent to which the two-level sharing hierarchy can be exploited depends on the locality in the sharing and data access patterns of applications, how well processes are mapped to processors in the hierarchy, and the cost difference between communicating within a node and across nodes. For example, applications that have wide but physically localized read-only sharing in a phase of computation, like the Barnes-Hut galaxy simulation, can benefit significantly from cache-to-cache sharing if the miss rates are high to begin with. Applications that exhibit nearest-neighbor sharing (like Ocean) can also have most of their accesses satisfied within a multiprocessor node if processes are mapped properly to nodes. However, although some processes may have all their accesses satisfied within their node, others will have accesses along at least one border satisfied remotely, so load imbalances will result and the benefits of the hierarchy will be diminished (performance will be limited by that of the most penalized processor). In all-to-all communication patterns, the savings in inherent communication is more modest. Instead of communicating with $p - 1$ remote processors in a p -processor system, a processor now communicates with $k - 1$ local processors and $p - k$ remote ones (where k is the number of processors within a node), a savings of at most $(p - k)/(p - 1)$ in internode communication. Finally, with several processes sharing a main memory unit, it may also be easier to distribute data appropriately among processors at page granularity. Some of these trade-offs and application characteristics are explored quantitatively in (Weber 1993; Erlichson et al. 1995). Of our two case study machines, the Sequent NUMA-Q uses four-processor, bus-based, cache-coherent SMPs as the nodes. The SGI Origin takes an interesting position: two processors share a bus and memory (and a board) to amortize cost, but they are not kept coherent by a snoopy protocol on the bus; rather, a single directory protocol keeps all caches in the machine coherent.

Compared to using uniprocessor nodes, the major potential disadvantage of using multiprocessor nodes is the sharing of communication resources by processors within a node. When processors share a bus, an assist, or a network interface, they amortize its cost but compete for its bandwidth. If their bandwidth demands are not reduced much by locality in sharing patterns, the resulting contention can hurt performance. The solution is to increase the throughput of these resources as well when processors are added to the node, but this compromises the cost advantages. Sharing

a bus within a node has some particular disadvantages. First, if the bus has to accommodate several processors, it becomes longer and is not likely to be contained in a single board or other packaging unit. These effects slow the bus down, increasing the latency to both local and remote data. Second, if the bus supports snooping coherence within the node, a request that must be satisfied remotely typically has to wait for local snoop results to be reported before it is sent out to the network, causing unnecessary delays. Third, with a snooping bus at the remote node too, many references that do go remote will require snoops and data transfers on the local bus as well as the remote bus, increasing latency and reducing effective data access bandwidth. Finally, snooping accesses second-level cache tags, which may cause unnecessary contention with processor accesses if the snoops are not often successful in achieving cache-to-cache sharing. Nonetheless, several directory-based systems use snoop-based coherent multiprocessors as their individual nodes (Lenoski et al. 1993; Lovett and Clapp 1996; Clark and Alnes 1996; Weber et al. 1997).

The final approach to improving protocol performance—improving the performance parameters of the communication architecture—is discussed in Section 8.7.

8.4.2 Correctness

As with snoop-based systems, correctness considerations can be divided into three classes. First, the protocol must ensure that the relevant blocks are invalidated/updated and retrieved as needed and that the necessary state transitions occur. We can assume this happens in all cases and not consider it much further. Second, the serialization and ordering relationships defined by coherence and the consistency model must be preserved. Third, the protocol and implementation must be free from deadlock, livelock, and, ideally, starvation. Several aspects of scalable protocols and systems complicate the latter two sets of issues beyond what we have seen for bus-based cache-coherent machines or scalable noncoherent machines. There are two basic problems. First, we now have multiple cached copies of a block but no single agent that can see all relevant transactions and serialize them. Second, with many processors, a large number of requests may be directed toward a single node, accentuating the input buffer problem discussed in Chapter 7. These problems are aggravated by the high latencies in the system, which push us to exploit protocol optimizations of the sort discussed previously; these optimizations allow more transactions to be in progress simultaneously and do not preserve a strict request-response nature, further complicating correctness. This subsection describes the major new issues and types of solutions that are commonly employed in each area of correctness. Some specific solutions used in the case study protocols are discussed in more detail in subsequent sections.

Serialization to a Location for Coherence

Recall the write serialization clause of coherence. Not only must a given processor be able to construct a serial order out of all the operations to a given location—at

least out of all write operations and its own read operations—but all processors must see the writes to a given location as having happened in the same order.

One mechanism we need for serialization is an entity that sees the necessary memory operations to a given location from different processors (the operations that are not contained entirely within a processing node) and determines their serialization. In a bus-based system, operations from different processors are serialized by the order in which their requests appear on the bus. In a distributed system that does not cache shared data, the consistent serializer for a location is the main memory that is the home of a location. For example, the order in which writes become visible to all processors is the order in which they reach the memory, and which write's value a read sees is determined by when that read reaches the memory. In a distributed system with coherent caching, the home memory is again a likely candidate for the entity that determines serialization to a given location, at least in a flat directory scheme, since all relevant operations first come to the home. If the home could satisfy all requests itself, then it could simply process them one by one in FIFO order of arrival and determine serialization. However, with multiple copies, visibility of an operation to the home does not imply visibility to all processors. It is easy to construct scenarios where processors may see operations to a location appear to be serialized in different orders than that in which the requests reached the home, as well as where different processors see operations complete in different orders.

As a simple example, consider an update-based protocol and a network that does not preserve point-to-point order of transactions between the same endpoints. If two write requests for shared data arrive at the home in one order, the updates they generate may arrive at the copies in different orders. As another example, suppose a block is in modified state in a dirty node and two nodes issue read-exclusive requests for it in an invalidation-based protocol. In a strict request-response protocol, the home will provide the requestors with the identity of the dirty node, and they will send requests to it. However, with different requestors, even in a network that preserves point-to-point order there is no guarantee that the requests will reach the dirty node in the same order as they reached the home. Which entity provides the globally consistent serialization in this case, and how is this orchestrated when multiple operations for this block may be simultaneously in flight and potentially needing service from different nodes?

Several types of solutions can be used to ensure serialization to a location. Most of them use additional directory states called *busy states* or *pending states*. A block being in busy state at the directory indicates that a previous request that came to the home for that block is still in progress and has not been completed. When a new request comes to the home and finds the directory state to be busy, serialization may be provided by one of the following mechanisms.

- *Buffer at the home.* The request may be buffered at the home as a pending request until the previous request that is in progress for the block has completed, regardless of whether the previous request was forwarded to a dirty node or whether a strict request-response protocol was used (the home should, of course, process requests for other blocks in the meantime). This method ensures that requests will be serviced everywhere in FIFO order of

their arrival at the home, but it reduces concurrency. It also requires that the home be notified when a write has completed or, more commonly, when the home's involvement with the write is over. Finally, it increases the danger of the input buffer at the home overflowing since this buffer holds pending requests for all blocks for which it is the home. One strategy in this case is to let the input buffer overflow into main memory, thus providing effectively infinite buffering as long as there is enough main memory and avoiding potential deadlock problems. This scheme is used in the MIT Alewife prototype (Agarwal et al. 1995).

- *Buffer at the requestors.* Pending requests may be buffered not at the home but at the requestors themselves, by constructing a distributed linked list of pending requests. This is a natural extension of a cache-based approach, which already has the support for distributed linked lists. It is used in the SCI protocol (Gustavson 1992; IEEE Computer Society 1993). Now the number of pending requests that a node may need to keep track of is small and determined only by the node itself.
- *NACK and retry.* An incoming request may be NACKed by the home (i.e., a negative acknowledgment sent to the requestor) rather than buffered when the directory state is busy. The request will be retried later by the requestor's assist and will be serialized in the order in which it is actually accepted by the directory (attempts that are NACKed do not enter in the serialization order). This is the approach used in the Origin2000 (Laudon and Lenoski 1997).
- *Forward to the dirty node.* If the directory state is busy because a request has been forwarded to a dirty node, subsequent requests for that block are not buffered at the home or NACKed. Rather, they too are forwarded to the dirty node, which determines their serialization. The order of serialization is thus determined by the home node when the block is clean at the home and by the order in which requests reach the dirty node when the block is dirty. If the block in the dirty node leaves the dirty state before a forwarded request reaches it (for example, due to a write back or a previous forwarded request), the request may be NACKed by the dirty node and retried. It will be serialized at the home or a dirty node when the retry is successful. This approach was used in the Stanford DASH protocol (Lenoski et al. 1990; Lenoski et al. 1993).

Unfortunately, with multiple copies in a distributed network, simply identifying a serializing entity is not enough. The problem is that the home or serializing agent may know (or be informed) when its involvement with a request is done, but this does not mean that the request has completed with respect to other nodes. Some transactions for the next request to that block may reach other nodes and perform with respect to them before some remaining transactions for the previous request. We see concrete examples and solutions in our case study protocols in Sections 8.5 and 8.6. Essentially, these show that, in addition to the system providing a global serializing entity for a block, individual nodes (e.g., requestors) should also preserve a local serialization with respect to each block; for example, they should not apply an incoming transaction to a block while they still have a transaction outstanding for that block.

Serialization across Locations for Sequential Consistency

Recall the two most interesting components of preserving the sufficient conditions for satisfying sequential consistency (SC): detecting write completion (needed to preserve program order) and ensuring write atomicity. In a bus-based machine, we saw that the restricted nature of the interconnect allows the requestor to detect write completion early; the write commits and can be acknowledged to the processor as soon as it obtains access to the bus, without waiting for it to actually invalidate or update other caches (Chapter 6). By providing a centralized path through which all transactions pass and ensuring FIFO ordering in the visibility of new data values beyond that path, a bus-based system also makes write atomicity quite natural to ensure.

In a machine that has a distributed network but does not cache shared data, detecting the completion of a write requires an explicit acknowledgment from the memory that holds the location (Chapter 7). In fact, the acknowledgment can be generated early, once we know the write has reached that node and been inserted in a FIFO queue to memory; at this point, the write has committed since it is clear that all subsequent reads that enter the queue will no longer see the old value, and we can use commitment as a substitute for completion to preserve program order. Write atomicity falls out naturally: a write is visible only when it reaches main memory, and at that point it is visible to all processors.

With both multiple copies and a distributed network, it is difficult to assume write completion before the invalidations or updates have actually reached all the nodes. A write cannot be acknowledged to the requestor once it has reached the home and be assumed to have effectively completed. The reason is that a subsequent write Y in program order may be issued by the same requestor after receiving such an acknowledgment for a previous write X , but Y may become visible to another processor before X , thus violating SC. This may happen because the invalidation or update transactions corresponding to Y take a different path through the network or because the network does not provide point-to-point order. Completion, or commitment, can only be assumed once explicit acknowledgments are received from all copies. Of course, a node with a copy can generate the acknowledgment as soon as it receives the invalidation—before it is actually applied to the caches—as long as it guarantees the appropriate ordering within its cache hierarchy (just as commitment is used instead of completion in Chapter 6). To satisfy the sufficient conditions for SC, a processor may wait after issuing a write until all acknowledgments for that write have been received and only then proceed past the write to a subsequent memory operation.

Write atomicity is similarly difficult when there are multiple copies and a distributed interconnect. To see this, Figure 8.14 shows how the semantics assumed by an example code fragment from Chapter 5 (Figure 5.11) that relies on write atomicity can be violated. The constraints of sequential consistency have to be satisfied by orchestrating network transactions appropriately. A common solution for write atomicity in an invalidation-based scheme is for the current owner of a block (the main

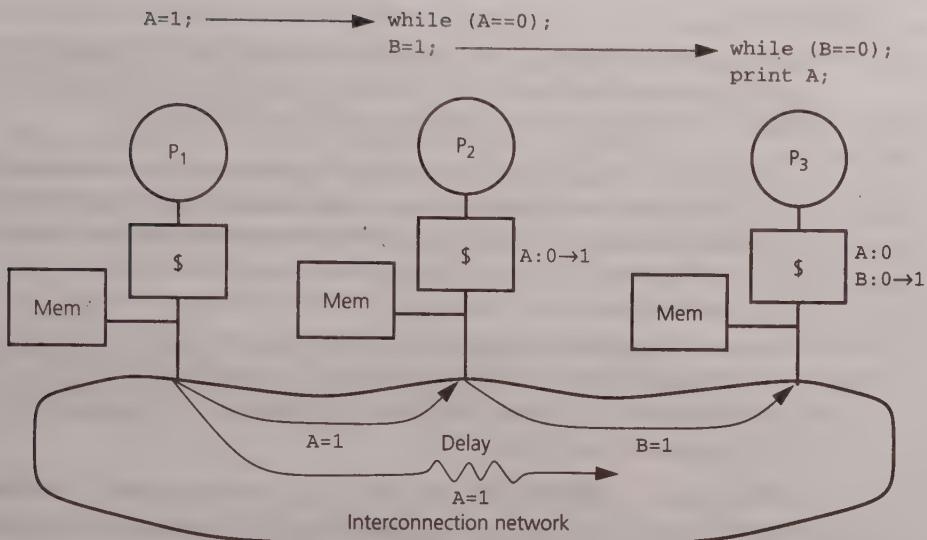


FIGURE 8.14 Violation of write atomicity in a scalable system with caches. The figure shows three processors and the code fragments that they execute. Assume that the network preserves point-to-point order and every cache starts out with copies of A and B initialized to 0. Transactions to look up directories and to satisfy read misses are ignored for simplicity. Under SC, we expect P₃ to print 1 as the value of A. However, P₂ sees the new value of A and jumps out of its while loop to write B even before it knows whether the previous write of A by P₁ has become visible to P₃. This write of B becomes visible to P₃ before the write of A by P₁, because the invalidation or update corresponding to the latter was delayed in a congested part of the network (that the other transactions did not have to go through at all). Thus, P₃ reads the new value of B but the old value of A, yielding a nonintuitive result.

memory module or the processor holding the dirty copy in its cache) to provide the appearance of atomicity by not allowing access to the new value by any process until all invalidation acknowledgments for the write that generated that value have returned. Thus, no processor can see the new value until it is visible to all processors. Maintaining the appearance of atomicity is much more difficult for update-based protocols since the data is sent to the sharers and, hence, is accessible immediately. Ensuring that no sharer reads the value until it is visible to all sharers requires a two-phase interaction. In the first phase, the copies of that memory block are updated in all relevant processors' caches, but those processors are prohibited from accessing the new value. In the second phase, after the first phase is known to have completed through acknowledgments as above, those processors are sent messages that allow them to use the new value. This difficulty and its performance implications help to make update protocols less attractive for scalable directory-based machines than for bus-based machines:

Deadlock

In Chapter 7, we discussed an important source of potential deadlock in request-response protocols such as those of a shared address space: the filling up of a finite input buffer. Three solutions were proposed for buffer deadlock:

1. Provide enough buffer space, either by buffering requests at the requestors using distributed linked lists or by providing enough input buffer space (in hardware or main memory) for the maximum number of possible incoming transactions.
2. Use NACKs.
3. Provide separate request and response networks, whether physically separate or multiplexed with separate buffers, to prevent backups in the potentially poorly behaved request network from blocking the progress of well-behaved response transactions.

Two separate networks would suffice in a protocol that is strictly request-response; that is, in which all transactions can be separated into requests and responses such that a request transaction generates only a response (or nothing) and a response generates no further transactions (and is, in this sense, better behaved since it does not generate further dependences). However, we have seen that in the interest of performance many practical coherence protocols use forwarding and are not always strictly request-response, breaking the deadlock avoidance assumption. In general, we need as many networks (physical or virtual) as the longest chain of different transaction types needed to complete a given operation so that the transaction at the end of a chain (that does not generate further transactions) is always guaranteed to make progress. However, using multiple networks is expensive and many of them will be underutilized. In addition to the approaches that provide enough buffering (as in the HAL S1 and MIT Alewife) or use NACKs throughout, two different approaches deal with deadlock in protocols that are not strict request-response. Both initially pretend that the protocol is strict request-response and provide two real or virtual networks, then rely on detecting situations when deadlock appears possible and resort to a different mechanism to avoid deadlock in these cases. That mechanism may be NACKs or reverting to a strict request-response protocol.

The detection of potential deadlock situations may be done in many ways. In the Stanford DASH machine, a node conservatively assumes that deadlock may be about to happen when both its input request and output request buffers fill up beyond a threshold and the request at the head of the input request buffer is one that may need to generate further requests like interventions or invalidations (i.e., that request is a violator of strict request-response operation and hence capable of causing deadlock). An alternative strategy is to assume the potential for deadlock when the output request buffer is full and has not had a transaction removed from it for T cycles. When potential deadlock is detected, the DASH system takes the first, NACK-based approach to avoiding deadlock: the node takes such requests off from the head of the input queue one by one and sends NACK messages back for them to

the requestors. It does this until the request at the head is no longer one that can generate further requests or until it finds that the output request queue is no longer full. The NACKed requestors will retry their requests later.

A different deadlock avoidance approach is taken by the Origin2000. When potential deadlock is detected, instead of sending a NACK to the requestor, the node sends it a response asking it to send the intervention or invalidation requests directly to the sharers; that is, the system dynamically backs off from a forwarding protocol to a strict request-response protocol, compromising performance temporarily but not allowing deadlock cycles. The advantage of this approach is that NACKing is a statistical rather than robust solution to such congestion-related problems: requests may have to be retried several times in bad situations, leading to increased network traffic and increased latency to the time the operation completes. Dynamic backoff also has advantages related to livelock, as we shall see next.

Livelock

In protocols that avoid deadlock by providing enough buffering of requests, whether centralized or through distributed linked lists, livelock and starvation are taken care of automatically as long as the buffers are FIFO. The other cases do not, in themselves, address livelock and starvation. In these cases, the classic livelock problem due to the race condition of multiple processors trying to write a block at the same time is often taken care of by letting the first request to get to the home go through but NACKing all the others.

NACKs are useful mechanisms for resolving race conditions like the preceding without livelock. However, when used to avoid deadlock in the face of input buffering limitations, as in the DASH solution outlined previously, they have, in fact, the potential to cause livelock. For example, when the node that detects a possible deadlock situation NACKs some requests, it is possible that all those requests are retried at the same time. With extreme pathology, the same situation could repeat itself continually and livelock could result.² The alternative solution to deadlock, of switching to a strict request-response protocol in potential deadlock situations, does not cause this livelock problem. It guarantees forward progress and removes the request-request dependence at the home once and for all.

Starvation

The occurrence of starvation is unlikely in well-designed protocols; however, it is not ruled out as a possibility. The fairest solution to starvation is to buffer all requests in FIFO order, which also solves deadlock and livelock. However, this can

2. While the DASH architecture is designed to use NACKs, the actual prototype implementation steps around this problem by using a large enough request input buffer since both the number of nodes and the number of possible outstanding requests per node are small. However, this is not a robust solution for larger, more aggressive machines that cannot provide enough buffer space.

have performance disadvantages, and for protocols that do not do this, avoiding starvation can be difficult to guarantee. Deadlock or livelock solutions that use NACKs and retries are often more susceptible to starvation, which is most likely when many processors repeatedly compete for a resource. Some may keep succeeding while one or more may be very unlucky in their timing and may always get NACKed.

A protocol could decide to do nothing about starvation and rely on the variability of delays in the system not to allow such an indefinitely repeating pathological situation to occur. The DASH machine uses this solution and times out with a bus error if the situation persists beyond a threshold time. Alternatively, a random delay can be inserted between retries to further reduce the small probability of starvation. Finally, requests may be assigned priorities based on the number of times they have already been NACKed, a technique that is used in the Origin2000 protocol.

Having an understanding of the basic directory organizations and high-level protocols as well as the key performance and correctness issues in a general context, we are now ready to dive into actual case studies of memory-based and cache-based protocols. We will see what protocol states and activities look like in actual realizations, how directory protocols interact with and are influenced by the underlying processing nodes, what scalable cache-coherent machines look like, and how actual protocols trade off performance with the complexity of maintaining correctness and of debugging or validating the protocol.

8.5

MEMORY-BASED DIRECTORY PROTOCOLS: THE SGI ORIGIN SYSTEM

Our discussion begins with flat, memory-based directory protocols, using the SGI Origin architecture as a case study. At least for moderate-scale systems, this machine uses essentially a full bit vector directory representation. A similar directory representation but slightly different protocol was also used in the Stanford DASH research prototype (Lenoski et al. 1990), which was the first distributed-memory machine to incorporate directory-based coherence. We follow a similar discussion template for both this and the next case study (the SCI protocol as used in the Sequent NUMA-Q). We begin with the basic coherence protocol, including the directory structure, the directory and cache states, how operations such as reads, writes, and write backs are handled, and the performance enhancements used. Then we will briefly discuss the position taken on the major correctness issues, followed by some prominent protocol extensions for extra functionality. Next, we will examine the rest of the machine as a multiprocessor and how the coherence machinery fits into it. This includes the processing node, the interconnection network, the input/output system, and any interesting interactions between the directory protocol and the underlying node. The case study ends with some important implementation issues (illustrating how it all works and the important data and control pathways), the basic performance characteristics (latency, occupancy, bandwidth) of the protocol, and the resulting application performance for our sample applications.

8.5.1 Cache Coherence Protocol

The Origin system is composed of a number of processing nodes connected by a switch-based interconnection network (see Figure 8.15). Every processing node contains two MIPS R10000 processors, each with first- and second-level caches, a fraction of the total main memory on the machine, an I/O interface, and a single-chip communication assist or coherence controller, called the Hub, that implements the coherence protocol. The Hub is integrated into the memory system. It sees all (second-level) cache misses issued by the processors in that node, whether they are to be satisfied locally or remotely; it receives transactions coming in from the network (in fact, the Hub implements the network interface); and it is capable of retrieving data from the local processor caches.

In terms of the performance issues discussed in Section 8.4.1, at the protocol level, the Origin2000 uses reply forwarding as well as speculative memory operations in parallel with directory lookup at the home. At the machine organization level, the decision in Origin to have two processors per node is driven mostly by cost: several other components on a node (the Hub, the system bus, and so on) are shared between the processors, thus amortizing their cost while hopefully still providing substantial bandwidth per processor. The Origin designers believed that the latency and bandwidth disadvantages of interacting with a snooping bus within a node outweighed its advantages and chose not to maintain snooping coherence between the two processors within a node. Rather, the SysAD (system address and data) bus is simply a shared physical link that is multiplexed between the two processors in a node. This sacrifices the potential advantage of cache-to-cache sharing within the node but eliminates the latency, occupancy, and cache tag contention added by snooping. In particular, with only two processors per node, the likelihood of successful cache-to-cache sharing is small, so the disadvantages may dominate. With a Hub shared between two processors, the combining of requests to the network (not to the directory protocol) could nonetheless have been supported, but it is not, due to the additional implementation cost. When discussing the protocol in this section, let us assume for simplicity that each node contains only one processor, together with its cache hierarchy, a Hub, and main memory. We consider the impact of using two processors per node on the directory structure and protocol later in this section.

Other than reply forwarding, the most interesting aspects of the Origin protocol are its use of busy states and NACKs to resolve race conditions and provide serialization to a location, its deadlock and livelock solution, the way in which it handles race conditions caused by write backs, and its nonreliance on any order preservation among transactions in the network (not even point-to-point order among transactions between the same endpoint nodes). To show how a complete protocol works in the presence of races as well as to illustrate the performance enhancement techniques used in different cases, we will look at how the Origin puts the techniques together to process read and write operations.

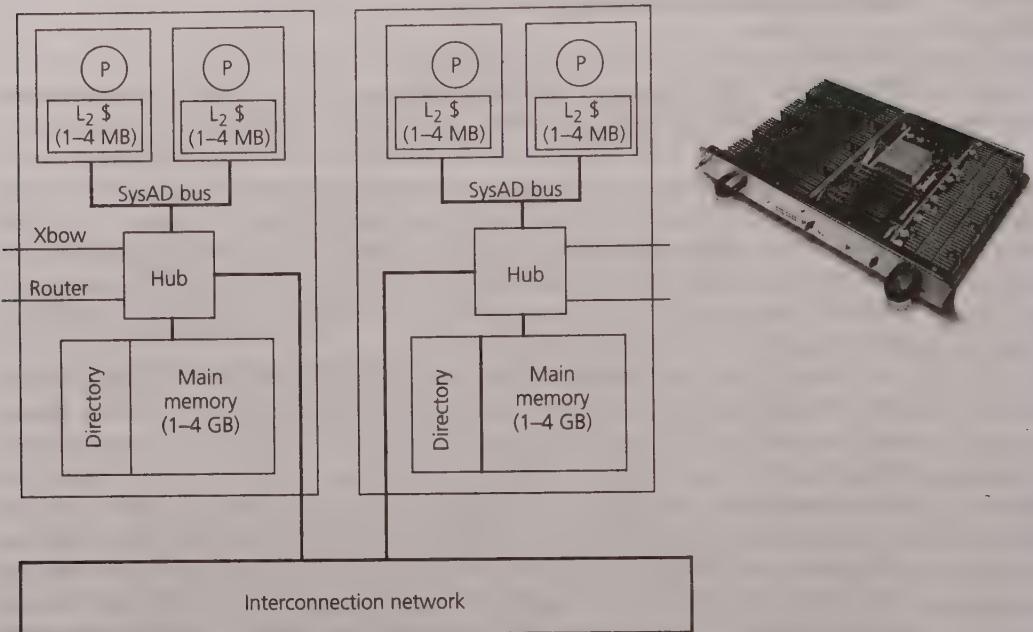


FIGURE 8.15 Block diagram of the Silicon Graphics Origin2000 multiprocessor. Each node contains two processors, a communication assist or controller called the Hub, and main memory with the associated directory. The photograph shows a single node board. Source: Photo courtesy of Silicon Graphics, Inc.

Directory Structure and Protocol States

The directory information for a memory block is maintained at the home node for that block. We assume a full bit vector approach for now and examine how the directory organization changes with machine size later.

In the caches, the protocol uses the same MESI states as used in Chapter 5. At the directory, a block may be in one of seven states. Three of these are stable states: *unowned*, or no cached copies in the system; *shared*, that is, zero or more read-only cached copies whose whereabouts are indicated by the presence vector; and *exclusive*, or one read-write cached copy in the system, indicated by the presence vector. An exclusive directory state means the block may be in either dirty or (clean) exclusive state in the cache (i.e., either the M or E states of the MESI protocol). Three other states are *busy* states. As discussed earlier, these imply that the home has received a previous request for that block but was not able to complete that operation itself (e.g., the block may have been dirty in a cache in another node); transactions to complete the request are still in progress in the system, so the directory at the home is not yet ready to handle a new request for that block. The three busy states correspond to three different types of requests that might still be in progress: a read, a read exclusive or upgrade, and an uncached read (a read whose result does

not enter the processor caches and is not kept coherent thereafter). Busy states and NACKs (rather than large amounts of buffering) are used by this protocol to avoid race conditions and provide serialization to a location. The seventh state is a *poison* state, which is used to implement a lazy TLB shootdown method for migrating pages among memories. (Protocol extensions like uncached operations and page migration are discussed in Section 8.5.4.) Given these states, let us see how the coherence protocol handles read, write, and write-back requests from a node.

Handling Read Requests

Suppose a processor issues a read that misses in its cache hierarchy. The address of the miss is examined by the local Hub to determine the home node, and a read request transaction is sent to the home node to look up the directory entry. If the home is local, the directory is looked up by the local Hub itself. At the home, the data for the block is accessed speculatively in parallel with looking up the directory entry. The directory entry lookup, which completes a cycle earlier than the speculative data access, may indicate that the memory block is in one of several different states—and different actions are taken in each case.

- *Shared or unowned.* This means that main memory at the home has the latest copy of the data (so the speculative access was successful). If the state is shared, the bit corresponding to the requestor is set in the directory presence vector; if it is unowned, the directory state is set to exclusive (achieving the functionality provided by the shared signal in snooping systems). The home then sends the data for the block back to the requestor in a reply transaction. These cases satisfy a strict request-response protocol. Of course, if the home node is the same as the requesting node, then no network transactions or messages are generated and it is a locally satisfied miss.
- *Busy.* This means that the home should not handle the request at this time since a previous request for the block is still in progress. The requestor is sent a negative acknowledge (NACK) message, asking it to try again later. A NACK is categorized as a response, but like an acknowledgment it does not carry data.
- *Exclusive.* This is the most interesting case. If the home is not the owner of the block, the valid data for the block must be obtained from the owner and must find its way to the requestor as well as to the home (since the state will change to shared). The Origin protocol uses reply forwarding: the request is forwarded to the owner, which replies directly to the requestor, sending a revision message to the home. If the home itself is the owner, then the home can simply reply to the requestor, change the directory state to shared, and set the requestor's bit in the presence vector. In fact, in general the directory treats a cache at the home just like any other cache; the only difference is that a “message” between the home directory and a cache at the home does not translate to a network transaction.

Let us look in a little more detail at what really happens when a read request arrives at the home and finds the state exclusive. (This and several other cases we discuss are illustrated in Figure 8.16.) The main memory block is accessed speculatively in parallel with the directory as usual. When the directory state is discovered to be exclusive, it is set to the busy-exclusive state to deal with subsequent requests, and the request is forwarded to the exclusive node. We cannot set the directory state to shared yet since memory does not yet have an up-to-date copy, and we do not want to leave it as exclusive since then a subsequent request might chase the same exclusive copy of the block as the current request does, requiring that serialization be determined by the current owner node rather than by the home.

Having set the directory entry to a busy state, the presence vector is changed to set the requestor's bit and unset the current owner's. Why this is done at this time becomes clear when we examine write-back requests. Now we see an interesting aspect of the protocol: even though the directory state is exclusive, the home optimistically assumes that the block will be in the (clean) exclusive rather than dirty state in the owner's cache and sends the speculatively accessed memory block at the home as a *speculative reply* (i.e., a reply with data that may or may not be useful) to the requestor. At the same time, the home forwards the intervention request to the owner. The owner checks the state in its cache and performs one of the following actions. If the block is in dirty state, it sends a reply with the data directly to the requestor and a revision message containing the data to the home. At the requestor, the response overwrites the stale speculative reply that was sent by the home. The revision message with data sent to the home is called a *sharing write back* since it writes the data back from the owning cache to main memory and tells it to set the block to shared state. If the block is in exclusive state, the reply to the requestor and the revision message to the home do not contain data since both already have the latest copy (the requestor has it via the speculative reply from the home). The response to the requestor is simply a completion acknowledgment, and the revision message is called a *downgrade* since it asks the home to downgrade the state of the block from (busy) exclusive to shared. In either case, when the home receives the revision message, it changes the state from busy to shared.

You may have noticed that the use of speculative replies does not have any significant performance advantage in this case since the requestor has to wait to know the real state at the exclusive node anyway before it can use the data. In fact, a simpler alternative to this scheme would be to simply assume that the block is dirty at the owner, not send a speculative reply, and always have the owner send back a reply with the data regardless of whether it has the block in dirty or (clean) exclusive state. Why then does the Origin protocol use speculative replies? There are two reasons, which illustrate how a protocol is influenced by the quirks of existing processors and how different protocol optimizations influence each other. First, the cache controller of the R10000 processor that the Origin uses happens not to return data when it receives an intervention to an exclusive (rather than dirty) cached block since memory is assumed to have a valid copy. Second, speculative replies enable a different optimization in the protocol, which is to allow a processor to simply drop a (clean) exclusive block when it is replaced from the cache, rather than notify main

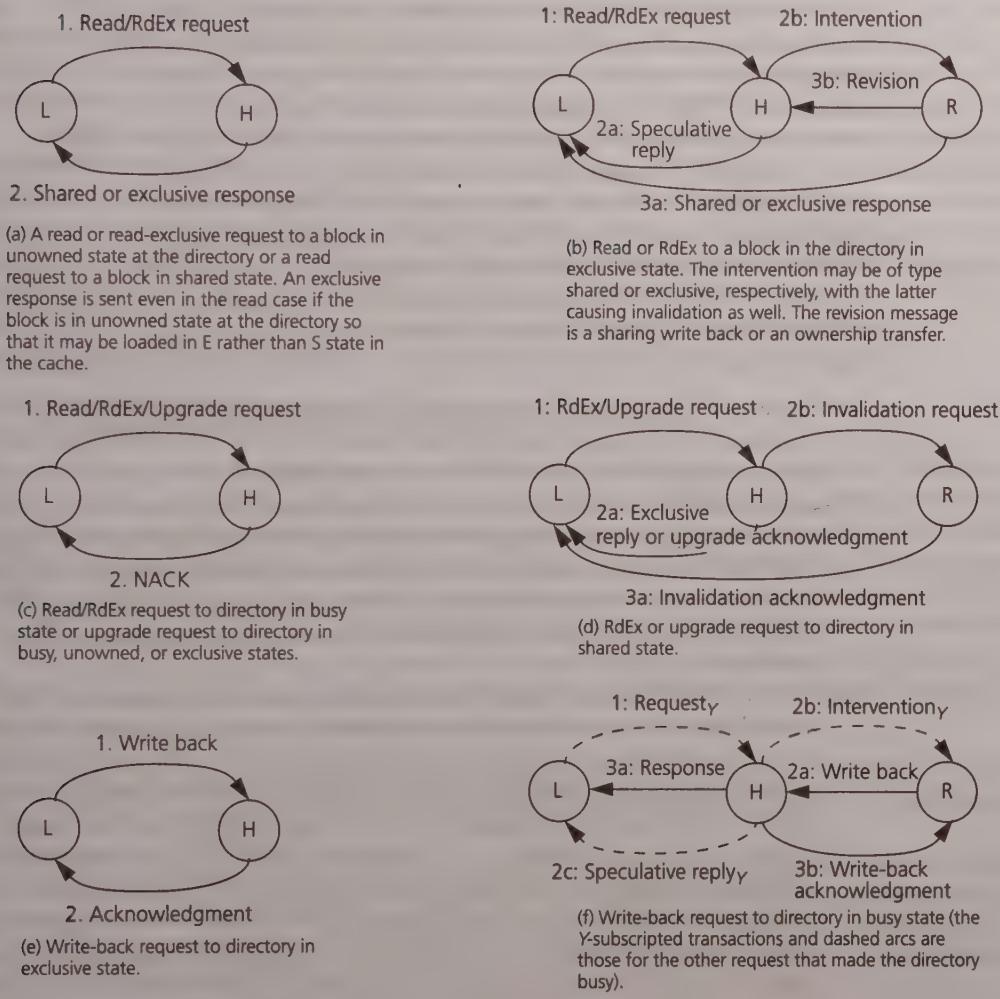


FIGURE 8.16 Protocol actions in response to requests in the Origin multiprocessor. The case or cases under consideration appear below the diagram, indicating the type of request and the state of the directory entry when the request arrives at the home. The messages or types of transactions are listed next to each arc. Since the same diagram represents different combinations of request type and directory state, different message types are listed on each arc.

memory that it now has the only copy and should reply to subsequent requests since main memory will in any case send a speculative reply when needed.

Handling Write Requests

As we saw in Chapter 5, write misses that invoke the protocol may generate either read-exclusive requests, which request both data and ownership, or upgrade

requests that request only ownership since the requestor's data is valid. In either case, the request goes to the home where the directory state is looked up to determine what actions to take. If the state at the directory is anything but unowned (or busy, which NACKs the request), the copies in other caches must be invalidated. To preserve the ordering model, invalidations must be explicitly acknowledged.

As in the read case, a strict request-response protocol, intervention forwarding, or reply forwarding can be used (see Exercise 8.4). Origin chooses reply forwarding to reduce latency: the home updates the directory state and sends the invalidations directly; it also includes the identity of the requestor in the invalidations so that they are acknowledged directly back to the requestor itself. The actual handling of the read-exclusive and upgrade requests depends on the state of the directory entry when the request arrives; that is, whether it is unowned, shared, exclusive, or busy.

- *Unowned.* If the request is an upgrade, the state at the directory is expected to be shared. The state being unowned means that the block has been replaced from the requestor's cache and the directory notified since it sent the upgrade request (this is possible since the Origin protocol does not assume point-to-point network order). An upgrade is no longer the appropriate request, so it is NACKed. The write operation will be retried, presumably as a read exclusive. If the request is a read exclusive, the directory state is changed to exclusive and the requestor's presence bit is set. The home replies with the data from memory.
- *Shared.* The block must be invalidated in the caches that have copies. The Hub at the home first makes a list of sharers that are to be sent invalidations, using the presence vector. It then sets the directory state to exclusive and sets the presence bit for the requestor. This ensures that the next request for the block will be forwarded to the requestor. If the request was a read exclusive, the home next sends a response to the requestor (called an “exclusive reply with invalidations pending”) that also contains the number of sharers from whom to expect invalidation acknowledgments. If the request was an upgrade, the home sends an “upgrade acknowledgment with invalidations pending” to the requestor, which is similar but does not carry the data for the block. In either case, the home next sends invalidation requests to all the sharers, which in turn send acknowledgments to the requestor (not the home). The requestor waits for all acknowledgments to come in before it “closes” or completes the operation. If a new request for the block comes to the home in the meantime, it will see the directory state as exclusive and will be forwarded as an intervention to the current requestor. This current requestor will not handle the intervention immediately but will buffer it until it has received all acknowledgments for its own request and closed that operation. (Further, requests coming to the home in the meantime will find the block in busy-exclusive state, as discussed earlier.)
- *Exclusive.* If the request is an upgrade, then an exclusive directory state means another write has beaten this request to the home. An upgrade is no longer the appropriate request and is NACKed. For a read-exclusive request, the following actions are taken. As with reads, the home sets the directory to a busy

state, sets the presence bit of the requestor, and sends a speculative reply to it. An invalidation request is sent to the owner, containing the identity of the write requestor (if the home is the owner, this is just an invalidation to the local cache and not a network transaction). If the owner has the block in dirty state, it sends a “transfer of ownership” revision message to the home (no data) and a reply with the data to the requestor. This reply overrides the speculative reply that the requestor receives from the home. If the owner has the block in (clean) exclusive state, it relies on the speculative reply from the home and simply sends an acknowledgment to the requestor and a “transfer of ownership” revision message to the home.

- **Busy.** The request is NACKed as in the read case and must try again.

Handling Write-Back Requests and Replacements

When a node replaces a block that is dirty in its cache, it generates a write-back request. This request carries data and is replied to with an acknowledgment by the home. The directory cannot be in unowned or shared state when a write-back request arrives because the write-back requestor has a dirty copy. (A read request cannot change the directory state to shared in between the generation of the write back and its arrival at the home since such a request would have been forwarded to the very node that is requesting the write back and the directory state would have been set to busy.) Let us see what happens when the write-back request reaches the home for the two possible directory states: exclusive and busy.

- **Exclusive.** The directory state transitions from exclusive to unowned (since the only cached copy has been replaced from its cache), and an acknowledgment is returned.
- **Busy.** This indicates an interesting race condition. The directory state can only be busy because an intervention for the block (due to a request from another node Y , say) has been forwarded to the very node X that is doing the write back. The intervention and write back have crossed each other in the interconnect. Now we are in a funny situation. The other operation from Y is already in progress and cannot be undone. We cannot let the write back be dropped, or we would lose the only valid copy of the block. Nor can we NACK the write back and retry it after the operation from Y completes, since then Y 's cache will have a valid copy while a different dirty copy is being written back to memory from X 's cache! This protocol solves the problem by essentially combining the two operations, using the write back as the response to Y 's request (see Figure 8.16[f]). The write back that finds the directory state busy changes the state to either shared (if the state was busy-shared, i.e., the request from Y was for a read copy) or exclusive (if it was busy-exclusive). The data returned in the write back is then forwarded by the home to the requestor Y . This serves as the response to Y instead of the response it would have received directly from X if there were no write back. When X receives an intervention for the block due to Y 's request, it simply ignores it (see Exercise 8.13). The directory also

sends a write-back acknowledgment to X. Node Y's operation is complete when it receives the response, and the write back is complete when X receives the write-back acknowledgment. We will see an exception to this treatment in a more complex case when we discuss the serialization of operations. In general, write backs introduce many subtle situations into directory-based coherence protocols.

If the block being replaced from a cache is in shared state, the node may or may not choose to send a replacement hint message back to the home, asking the home to clear its presence bit in the directory. Replacement hints avoid the next useless invalidation to that block and can reduce the occurrence in limited-pointer directory representations, but they incur assist occupancy and do not reduce traffic. In fact, if the block is not written again by another node, then the replacement hint is a waste. The Origin protocol does not use a limited-pointer representation and does not use replacement hints.

In all, the number of transaction types for coherent memory operations in the Origin protocol is 9 requests, 6 invalidations and interventions, and 39 responses. For noncoherent operations such as uncached memory operations, I/O operations, and special synchronization support, the number of transactions is 19 requests and 14 replies (no invalidations or interventions since there is no coherent caching).

8.5.2 Dealing with Correctness Issues

So far, we have seen what happens at different nodes upon read and write misses and how some important race conditions are resolved. Let us now take a different cut through the Origin protocol, examining the specific solutions it adopts for the correctness issues discussed in Section 8.4.2 and the features that the machine provides to deal with errors that may occur.

Serialization to a Location for Coherence

The entity designated to serialize cache misses from different processors is the home. As we have seen, serialization is provided not by buffering requests at the home until previous ones have completed or forwarding them to the owner node even when the directory is in a busy state but by NACKing requests from the home when the state is busy and causing them to be retried. Requests are forwarded only from stable directory states. Serialization is determined by the order in which the home *accepts* the requests—that is, satisfies them itself or forwards them—not the order in which they first arrive at the home.

The general discussion of serialization techniques in Section 8.4.2 suggested that more was needed for serialization to a given location than simply a global serializing entity since the serializing entity does not have full knowledge of when transactions related to a given operation are completed at all the relevant nodes. With a sufficiently in-depth understanding of a protocol, we now examine some concrete examples of this problem (Lenoski 1992) and see how it might be addressed (see Examples 8.1 and 8.2).

EXAMPLE 8.1 Consider the following simple piece of code.

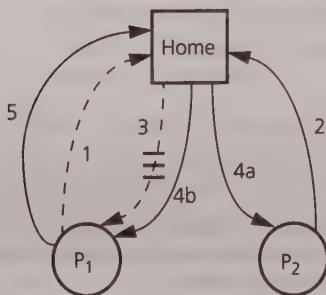
P_1	P_2
rd A (i)	wr A
BARRIER	BARRIER
rd A (ii)	

The write of A may happen either before the first read of A or after it, but it should be serializable with respect to that first read. The second read of A should in any case return the value written by P_2 . However, it is quite possible for the effect of the write to get lost if we are not careful. Show how this might happen in a protocol like the Origin's, and discuss possible solutions.

Answer Figure 8.17 shows how the problem can occur, with the text in the figure explaining the transactions, the sequence of events, and the problem. There are two possible solutions. An unattractive one is to have read replies themselves be acknowledged explicitly and let the home go on to process the next request only after it receives this acknowledgment. This further violates the request-response nature of the protocol, causes buffering and potential deadlock problems, and leads to long delays. The more likely solution is to ensure that a node that has a request outstanding for a block, such as P_1 , does not allow access by another request, such as the invalidation, to be applied to that block in its cache until its outstanding request completes. P_1 may buffer the incoming invalidation request and apply it only after the read reply is received and completed. Or P_1 can apply the invalidation even before the read reply is received and then consider the reply invalid (a NACK) when it returns and retry the read. Origin uses the former solution whereas the latter is used in DASH. The order of P_1 's (first) read with respect to P_2 's write is different in the two machines, but both orders are valid. The buffering needed is small and does not cause deadlock problems. ■

EXAMPLE 8.2 In addition to the requestor, the home too may have to disallow new operations from actually being applied to a block (or its directory state) before previous ones have completed as far as it is concerned. Otherwise, directory information may be corrupted. Show an example illustrating this need and discuss solutions.

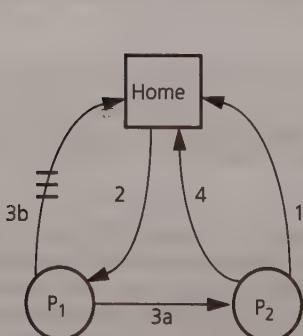
Answer This example is more subtle and is shown in Figure 8.18. The node issuing the write request detects completion of the write (as far as its involvement is concerned) through acknowledgments before processing another request for the block. The problem is that the home does not wait for its involvement in the write operation—which includes waiting for the revision message and directory update—to complete before it allows another access (here the write back) to be applied to the block. The Origin protocol prevents this from happening by using its busy state: the directory will be in busy-exclusive state when the write back arrives before the revision message. When the directory detects that the write back is coming from the same node whose request put the directory into busy-exclusive state, the write back is NACKed and must be retried. (Recall from the discussion of handling write backs that the write back was treated differently if the request that set the state to busy came from a different node than from the one doing the write back; in that case, the write back was not NACKed but was sent on as the response to the requestor.) ■



1. P_1 sends read request to home node for A.
2. P_2 sends read-exclusive request to home (for the write of A). Home (serializer) won't process it until it is done with read from P_1 , which it receives first.
3. In response to (1), home sends reply to P_1 (and sets directory presence bit). Home now thinks read is complete (there are no acknowledgments for a read reply). Unfortunately, the reply does not get to P_1 right away.
- 4a. In response to (2), home sends data reply to P_2 corresponding to request 2.
- 4b. In response to (2), home sends invalidation to P_1 ; it reaches P_1 before transaction 3 (no point-to-point order is assumed in Origin, and in general the invalidation is a request and 3 is a response, so they may travel on different networks).
5. P_1 receives and applies invalidation, sends acknowledgment to home.

Finally, the read reply (3) reaches P_1 and overwrites the invalidated block. When P_1 reads A after the barrier, it reads this old value rather than seeing an invalid block and fetching the new value. The effect of the write by P_2 is lost as far as P_1 is concerned.

FIGURE 8.17 Example illustrating the need for local serialization of operations at a requestor. The example shows how a write can be lost even though home thinks it is doing things in order. Transactions associated with the first read operation are shown with dotted lines, and those associated with the write operation are shown in solid lines. The three solid bars through a transaction indicate that it is delayed in the network.



Initial condition: block is in dirty state in P_1 's cache.

1. P_2 sends read-exclusive request to home.
2. Home forwards request to P_1 (dirty node).
3. P_1 sends data reply to P_2 (3a) and "ownership transfer" revision message to home to change owner to P_2 (3b).
4. P_2 , having received its reply, considers write complete. Proceeds, but incurs a replacement of the just dirtied block, causing it to be written back in transaction 4.

This write back is received by the home before the ownership transfer revision message from P_1 (even point-to-point network order wouldn't help), and the block is written into memory. Then when the revision message arrives at the home, the directory is made to point to P_2 as having the dirty copy. But this is untrue, and our protocol is corrupted.

FIGURE 8.18 Example illustrating the need for local serialization of operations at a home node. The example shows how directory information can be corrupted if a home node does not wait for its involvement with a previous request to be over (e.g., a revision message to be received from the owner node) before it allows a new access to the same block.

These examples illustrate the importance of another general requirement that nodes must locally fulfill for proper serialization, beyond the existence of a global serializing entity for a block: any node, not just the serializing entity, should not apply a transaction corresponding to a new memory operation to a block until a previously outstanding memory operation on that block (that the node has begun to handle) is complete as far as that node's involvement is concerned.

Preserving the Memory Consistency Model

The dynamically scheduled R10000 processor allows independent memory operations to issue out of program order, allowing multiple operations to be outstanding at a time and achieving some overlap among them. However, it ensures that operations complete in program order and, in fact, that writes leave the processor environment and become visible to the memory system in program order with respect to other operations, thus preserving sequential consistency (Chapter 11 discusses the necessary processor mechanisms further). The processor does not satisfy the sufficient conditions for sequential consistency spelled out in Chapter 5 in that it does not wait to issue the next operation until the previous one completes, but a system that uses this processor and provides atomicity satisfies the model itself.³

Since the processor guarantees visibility and completion in program order, the extended memory hierarchy can perform any reorderings to different locations that it desires without violating this property. The Origin protocol provides write atomicity as discussed earlier: a node does not allow any incoming accesses to a block for which invalidations are outstanding until the acknowledgments for those invalidations have returned (i.e., the write is committed). Nonetheless, one implementation consideration is important in maintaining SC that is due to the Origin protocol's interactions with the processor. Recall from Figure 8.16(d) what happens on a write request (read exclusive or upgrade) to a block that is in shared state at the directory. The requestor receives two types of responses: an *exclusive reply* from the home, discussed earlier, whose role is to indicate that the write has been serialized at memory with respect to other operations for the block and perhaps to return data; and *invalidation acknowledgments*, indicating that the other copies have been invalidated and the write has completed. The microprocessor, however, expects only a single response to its write request, as in a uniprocessor system, so these different responses have to be dealt with by the requesting Hub. To ensure sequential consistency, the Hub must pass the response on to the processor—allowing it to declare completion of the write—only when both the exclusive reply and the invalidation acknowledgments have been received. It must not pass on the response simply when the exclusive reply has been received since that would allow the processor to complete later accesses to other locations even before all invalidations for this one have been

3. This is true for accesses that are under the control of the coherence protocol. The processor also supports memory operations that are not visible to the coherence protocol, called noncoherent memory operations, for which the system does not guarantee any ordering: it is the user's responsibility to insert synchronization to preserve a desired ordering in these cases.

acknowledged, violating sequential consistency. We see in Section 9.1 that such violations are useful when more relaxed memory consistency models than SC are used.

Deadlock, Livelock, and Starvation

The Origin uses finite input buffers and a protocol that is not strict request-response. As discussed in Section 8.4.2, to avoid deadlock, it uses the technique of reverting to a strict request-response protocol when it detects a high-contention situation that may cause deadlock. Since NACKs are not used to alleviate the contention, livelock is avoided in these situations too. The classic livelock problem due to multiple processors trying to write a block at the same time is avoided by using busy states and NACKs (recall that NACKs avoid rather than cause livelock in this case). The first of these requests to get to the home sets the state to busy and makes forward progress while others are NACKed and must retry.

In general, the philosophy of the Origin protocol is twofold: (1) to be “memoryless,” that is, every node reacts to incoming events using only current local state and no history of previous events; and (2) not to allow an operation to hold globally shared resources while it is requesting other resources. The latter leads to the choices of NACKing rather than buffering for a busy resource and helps prevent deadlock. These decisions greatly simplify the hardware yet provide high performance in most cases. However, since NACKs are used rather than FIFO ordering, the problem of starvation still exists. This is addressed by associating a priority with a request, which is a function of the number of times the request has been NACKed.⁴

Error Handling

Despite a correct protocol, hardware and software errors can occur at run time. These can corrupt memory or write data to different locations than expected (e.g., if the address on which to perform a write becomes corrupted). The Origin system provides many standard mechanisms to handle hardware errors on components. All caches and memories are protected by error correction codes (ECCs), and all router and I/O links are protected by cyclic redundancy checks (CRCs) and a hardware link-level protocol that automatically detects and retries failures. In addition, the system provides mechanisms to contain failures within the part of the machine in which the program that caused the failure is running. Access protection rights are

4. The priority mechanism works as follows. The directory entry for a block has a “current” priority associated with it. Incoming transactions that will not cause the directory state to become busy are always serviced. Other transactions will potentially be serviced only if their priority is greater than or equal to the current directory priority. If such a transaction is NACKed (e.g., because the directory is in busy state when it arrives), the current priority of the directory is set to be equal to that of the NACKed request. This ensures that the directory will no longer service another request of lower priority until this one is serviced upon retry. To prevent a monotonic increase and “topping out” of the directory entry priority, it is reset to zero whenever a request of priority greater than or equal to it is serviced.

provided on both memory and I/O devices, preventing unauthorized nodes from making modifications. These access rights allow the operating system to be structured into cells or partitions, an organization called a *cellular operating system*. A cell is a number of nodes, configured at boot time. If an application runs within a cell, it may be disallowed from writing memory or I/O outside that cell. If the application fails and corrupts memory or I/O, it can only affect other applications or the system running within that cell and cannot harm code running in other cells. Thus, a cell is the unit of fault containment in the system.

8.5.3 Details of Directory Structure

While we have assumed a full bit vector directory organization so far for simplicity, the actual structure of the Origin directory entry is a little more complex for two reasons: first, to deal with the two processors per node and, second, to allow the directory structure to scale to more than 64 nodes with a 64-bit entry. There are, in fact, three possible formats or interpretations of the directory bits. If a block is in an exclusive state (i.e., modified or exclusive) in a processor cache, then the rest of the directory entry is not a bit vector with one bit turned on but rather contains an explicit pointer to that specific processor (not node). This means that interventions forwarded from the home are targeted to a specific processor. Otherwise, for example, if the directory state is shared, the directory entry is interpreted as a bit vector. Bits in the bit vector correspond to nodes, so even though the two processor caches within a node are not kept coherent by the bus, the unit of visibility to the directory in this format is a node or Hub, not a processor. If an invalidation is sent to a Hub, unlike an intervention, it is broadcast to both processors in the node over the SysAD bus that connects the two processors and the Hub. There are two sizes for presence bit vectors: 16 bit and 64 bit (in the 16-bit case, the directory entry is stored in the same DRAM as the main memory whereas in the 64-bit case the rest of the bits are in an extended directory memory module that is looked up in parallel). The 16-bit vector therefore supports up to 32 processors, and the 64-bit vector supports up to 128 processors.

For larger systems, the interpretation of the bits changes to the third format. In a p -node system, each bit now corresponds to a fixed set of $p/64$ nodes. The bit is set when any one (or more) of the nodes in the corresponding set has a copy of the block. If a bit is set when a write happens, then invalidations are sent to all the $p/64$ nodes represented by that bit (and are then broadcast to both processors in each of those nodes). For example, with the maximum supported size of 1,024 processors (512 nodes), each bit corresponds to 8 nodes. This is called a *coarse vector representation*, and we see it again when we discuss overflow strategies for directory representations as an advanced topic in Section 8.10. In fact, the system dynamically chooses between the bit vector and coarse vector representation on a large machine: if all the nodes sharing the block are within the same 64-node octant of the machine, a bit vector representation is used; otherwise, a coarse vector is used.

8.5.4 Protocol Extensions

In addition to the protocol optimizations discussed earlier, the Origin protocol provides some extensions to support special operations and activities that interact with the protocol. These include input/output and DMA operations, page migration, and synchronization.

Support for Input/Output and DMA Operations

To support memory reads by a DMA device, the protocol provides “uncached read-shared” requests. Such a request returns to the DMA device a snapshot of a coherent copy of the data, but that copy is then no longer kept coherent by the protocol. The request is used primarily by the I/O system and the block transfer engine provided in the Hub and as such is intended for use by the operating system. For writes to memory from a DMA device, the protocol provides “write invalidate” requests. A write invalidate simply blasts the new value of a word into memory, overwriting the previous value. It also invalidates all existing cached copies of the block in the system, thus returning the directory entry to unowned state. From a protocol perspective, it behaves much like a read-exclusive request, except that it modifies the block in memory and leaves the directory in unowned state.

Support for Automatic Page Migration

As we discussed in Chapter 3, on a machine with physically distributed memory it is often important to allocate data appropriately across physical memories so that most capacity, conflict, and cold misses are satisfied locally. On CC-NUMA machines like the Origin, data is allocated in memory at the granularity of a page (16 KB, in this case). Despite the very aggressive communication architecture in the Origin, the latency of an access satisfied by remote memory is at least 2–3 times that of a local access even without contention. The appropriate distribution of pages among memories might change dynamically at run time, either because a parallel program’s access patterns change or because the operating system decides to migrate an application process from one processor to another for better resource management across multiprogrammed applications. It is therefore useful for the system to detect the need for moving pages at run time and migrate them automatically to where they are needed.

For every page in main memory, Origin provides an array of miss counters, one per node, to help determine when most of the misses to a page are coming from a nonlocal processor so that the page should be migrated. The miss counters are stored in directory memory at the home. When a request comes in for a page, the miss counter for that node is incremented and compared with the miss counter for the home node. If it exceeds the latter by more than a threshold, then the page can be migrated to that remote node. (Sixty-four counters are provided per page, and in a system with more than 64 nodes, 8 nodes share a counter.) Page migration is typi-

cally very expensive, which often annuls the advantage of doing the migration. The major reason for the high cost is not so much moving the page (which with the block transfer engine in the Hub takes about 25–30 μ s for a 16-KB page) as changing the virtual-to-physical mappings in the TLBs of all processors that have referenced the page. Migrating a page keeps the virtual address the same but changes the physical address, so the old mappings in the page tables of processes are now invalid. As page table entries are changed, it is important that the cached versions of those entries in the TLBs of processors be invalidated (much like TLB shootdown discussed in Chapter 6). In fact, all processors must be sent a TLB invalidation message since we don't know which ones have a mapping for the page cached in their TLB. The processors are interrupted, and the invalidating processor has to wait for the last among them to respond before it can update the page table entry and continue. This process typically takes over 100 μ s, in addition to the cost to move the page itself.

To reduce this cost, Origin uses a distributed, lazy TLB invalidation mechanism supported by its seventh directory state, the poisoned state. The idea is not to invalidate TLB entries when the page is moved but rather to invalidate a processor's TLB entry only when that processor next references the page. Not only is the time to invalidate all TLBs removed from the critical path of the processor that manages the migration, but TLB entries end up being invalidated for only those processors that subsequently reference the page. Let's see how this works. To migrate a page, a block transfer engine reads all cache blocks from the source page location using special "uncached read-exclusive" requests. This request type returns the latest coherent copy of the data and invalidates any existing cached copies (like a regular read-exclusive request), but it also causes the destination main memory to be updated with the latest version of the block and puts the directory in the poisoned state. The migration itself takes only the time to do this block transfer. When a processor next tries to access a block from the old physical page, using its stale TLB entry, it will miss in the cache and will find the block in poisoned state at the directory. At that time, the poisoned state will cause the requesting processor to see a bus error. The special OS handler for this bus error invalidates the processor's TLB entry so that it will obtain the new mapping from the page table when it retries the access. Of course, the old physical page must be reclaimed by the system at some point to avoid wasting storage. Once the block transfer has completed, the OS invalidates one TLB entry per time quantum of the OS scheduler so that after some fixed amount of time the old page can be moved on to the free list.

Support for Synchronization

Origin provides two types of support for synchronization. First, the load-locked store-conditional (LL-SC) instructions of the R10000 processor are available to compose synchronization operations, as we saw in the previous chapters. Second, for situations in which many processors contend to update a location, such as a global counter or a barrier, uncached fetch&op primitives are provided. These fetch&op operations are performed at the main memory; the block is not replicated in the

caches, so successive nodes trying to update the location do not have to retrieve it from the previous writer's cache. The cacheable LL-SC is better when the same node tends to repeatedly access the shared (synchronization) variable, and the uncached fetch&rop is better when different nodes tend to update in an interleaved or contended way. Producer-consumer communication of event synchronization can also be aided by uncached write and read operations since at most two network transactions are needed instead of four and since the producer and consumer transactions may even overlap on their way to the home node. However, spinning on a remote uncached location may cause a lot of traffic.

8.5.5 Overview of the Origin2000 Hardware

The preceding protocol discussion has provided us with a fairly complete picture of how a flat, memory-based directory protocol is implemented out of network transactions and state transitions, just as a bus-based protocol was implemented out of bus transactions and state transitions. Let us now turn our attention to the actual hardware of the Origin2000 machine that implements this protocol. This subsection provides an overview of the system hardware organization and is followed by a deeper examination of how the Hub controller is actually implemented (in Section 8.5.6). Finally, the performance of the machine is discussed in Section 8.5.7. (Readers interested in only the protocol can skip the rest of this section without loss of continuity.)

In addition to the two MIPS R10000 processors connected by a system bus, each node of the Origin2000 contains a fraction of the main memory on the machine (1–4 GB per node), the Hub (which is the combined communication/coherence controller and network interface), and an I/O interface called the Xbow. All components but the Xbow are on a single 16" × 11" printed circuit board. Each processor in a node has its own separate L₁ and L₂ caches, with the L₂ cache configurable from 1 to 4 MB with a cache block size of 128 bytes and two-way set associativity. There is one directory entry per main memory block. Memory is interleaved from 4 ways to 32 ways, depending on the number of modules plugged in (4-way interleaving at 4-KB granularity within a module and up to 32-way at 512-MB granularity across modules). The system has up to 512 such nodes, that is, up to 1,024 processors. With a 195-MHz R10000 processor, the peak performance per processor is 390 MFLOPS or 780 MIPS (four instructions per cycle), leading to an aggregate peak performance of almost 500 GFLOPS in a maximally sized machine. The peak bandwidth of the SysAD bus that connects the two processors is 780 MB/s, as is that of the Hub's connection to memory. Memory bandwidth itself for data is about 670 MB/s. The Hub connections to the off-board network router chip and Xbow I/O interface are 1.56 GB/s each, using the same link technology. A detailed picture of the node board is shown in Figure 8.19.

The Hub chip is the heart of the machine. It sits on the system bus of the node and connects the processors, local memory, network, and Xbow, which communicate with one another through it. All cache misses, whether to local or remote memory, go through the Hub (which implements the coherence protocol), as do all

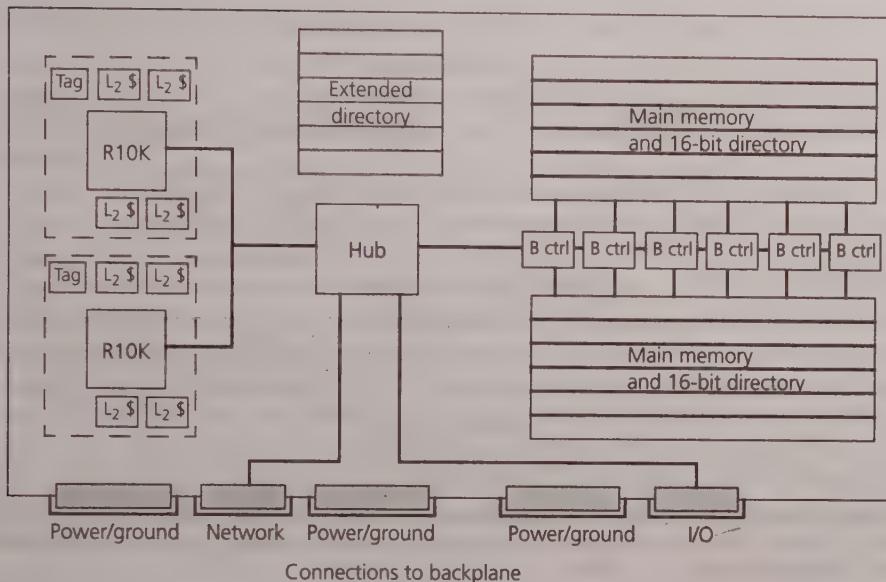


FIGURE 8.19 A node board on the Origin multiprocessor. “L₂ \$” stands for secondary cache chips and “B ctrl” for memory bank controller.

uncached operations. It is a highly integrated, 500-K gate standard-cell design in 0.5- μ CMOS technology. It contains outstanding transaction buffers for each of its two processors (each processor itself allows four outstanding requests), a pair of block transfer engines that support block memory copy and fill operations at full system bus bandwidth, and interfaces for the network, the SysAD bus, the memory/directory, and the I/O subsystem. The Hub also implements the at-memory, uncached fetch&rop instructions and page migration support discussed earlier.

The interconnection network has a hypercube topology for machines with up to 64 processors but a different topology, called a *fat cube*, beyond that. (This topology is discussed in Chapter 10.) Each router supports six links. The network links have high bandwidth (1.56 GB/s total per link in the two directions) and low latency (41 ns pin-to-pin through a router) and can use flexible cabling up to three feet long for the links. Each link supports four virtual channels. Virtual channels are described in Chapter 10; for now, we can think of the machine as having four distinct networks such that each has about one-fourth of the physical link bandwidth. One of these virtual channels is reserved for request network transactions, one for responses. Two can be used for congestion relief and high-priority transactions, thereby violating point-to-point order, or can be reserved for I/O as is usually done.

The Xbow chip connects the Hub to other I/O interfaces. It is itself implemented as a crossbar with eight ports. Typically, two nodes (Hubs) might be connected to one Xbow and, through it, to six external I/O cards as shown in Figure 8.20. The Xbow is quite similar to the router chip (called SPIDER) but with simpler buffering

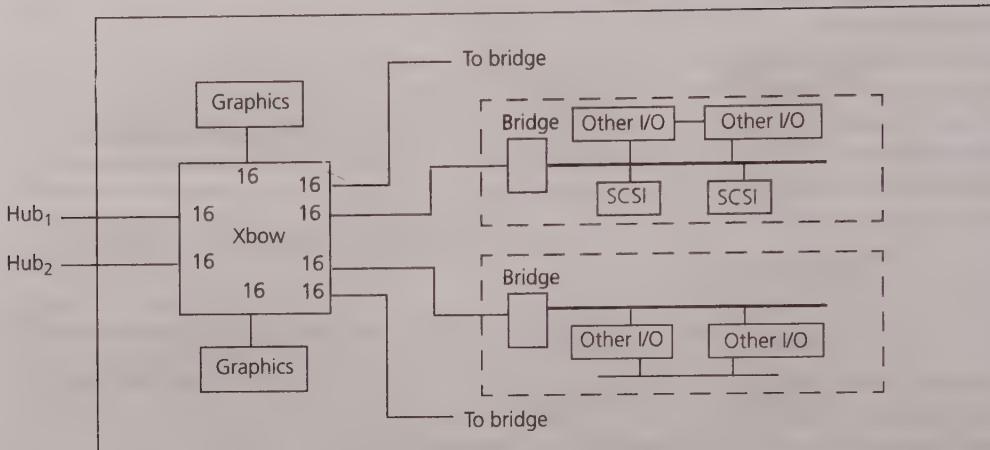


FIGURE 8.20 Typical Origin I/O configuration shared by two nodes. High-performance graphics devices connect directly to the Xbow, while other I/O devices connect to I/O buses that are linked to the Xbow through bridges.

and arbitration that allow eight ports to fit on the chip rather than six. The arbiter also supports the reservation of bandwidth for certain devices to support real-time needs like video I/O. High-performance I/O cards like graphics connect directly to the Xbow ports, but most other ports are connected through a bridge and an I/O bus that allows multiple cards to plug into it. Any processor can reference any physical I/O device in the machine, either through uncached references to a special I/O address space or through coherent DMA operations. An I/O device, too, can transfer data to and from any memory in the system, not just the memory on the node to which it is directly connected through the Xbow, thus taking advantage of the shared address space. Communication between the processor and the appropriate Xbow is handled transparently by the Hubs and network routers. Thus, like memory, I/O is physically distributed but globally accessible, so locality in I/O distribution is also a performance rather than correctness issue.

8.5.6 Hub Implementation

The communication assist—the Hub—must have certain basic abilities to implement the coherence protocol. It must be able to observe all cache misses, synchronization events, and uncached operations; keep track of outgoing requests while moving on to handle other outgoing and incoming transactions; guarantee the sinking of responses coming in from the network; invalidate cache blocks; and intervene in the caches to retrieve data. It must also coordinate the activities and dependences of all the different types of transactions that flow through it from different components and implement the necessary pathways and control. The design of such controllers is, therefore, challenging. This subsection briefly describes the major

components of the Hub controller used in the Origin2000 and points out some of its salient features used to implement the coherence protocol. Further details of the actual data and control pathways through the Hub, as well as the mechanisms used to actually control the interactions among messages, are also useful for understanding how scalable cache coherence is implemented and can be read elsewhere (Singh 1997).

The Hub is divided into four major interfaces, one for each type of external entity that it connects together: the processor interface or PI, the memory/directory interface or MI, the network interface or NI, and the I/O interface or II (see Figure 8.21). These interfaces communicate with one another through an on-chip crossbar switch. Each interface is divided into a few major structures, including FIFO queues to buffer messages to/from other interfaces and to/from external entities. A key property of the design is for each interface to shield its external entity from the details of other interfaces and entities (and vice versa). For example, the PI hides the processors from the rest of the world, so any other interface must only know the behavior of the PI and not of the processors and SysAD bus themselves. Let us discuss the structures of the PI, MI, and NI briefly, as well as some examples of the shielding provided by the interfaces.

The Processor Interface (PI)

The PI has the most complex control mechanisms among the interfaces since it keeps track of outstanding protocol requests and responses and must match them. The PI interfaces with the memory (SysAD) buses of the two R10000 processors on one side and with incoming and outgoing FIFO queues connecting it to each of the other Hub interfaces on the other side (Figure 8.21). Each physical FIFO is logically separated into independent request and response “virtual FIFOs” by providing separate logic and staging buffers. In addition, the PI itself contains three pairs of coherence control buffers that keep track of outstanding transactions, control the flow of messages through the PI, and implement the interactions among messages dictated by the protocol. These buffers do not, however, hold the messages themselves. There are two read request buffers (RRBs) that track outstanding read requests from each processor, two write request buffers (WRBs) that track outstanding write requests, and two intervention request buffers (IRBs) that track incoming invalidation and intervention requests. Access to the three sets of buffers is through a single bus, so all messages contend for access to them.

A message that is recorded in one type of buffer may also need to look up another type to check for conflicting accesses or interventions to the same address from the processor. For example, an outgoing read request performs an associative lookup in the WRB to see if a write back to the same address is pending as well. If there is a conflicting WRB entry, a read request is not placed in the PI's outgoing request FIFO; rather, a bit is set in the RRB entry to indicate that when the WRB entry is freed, the read request should be reissued (i.e., when the write back is acknowledged or is canceled by an incoming invalidation as per the protocol). Buffers are also looked up to close an outstanding PI transaction in them when a completion response comes in

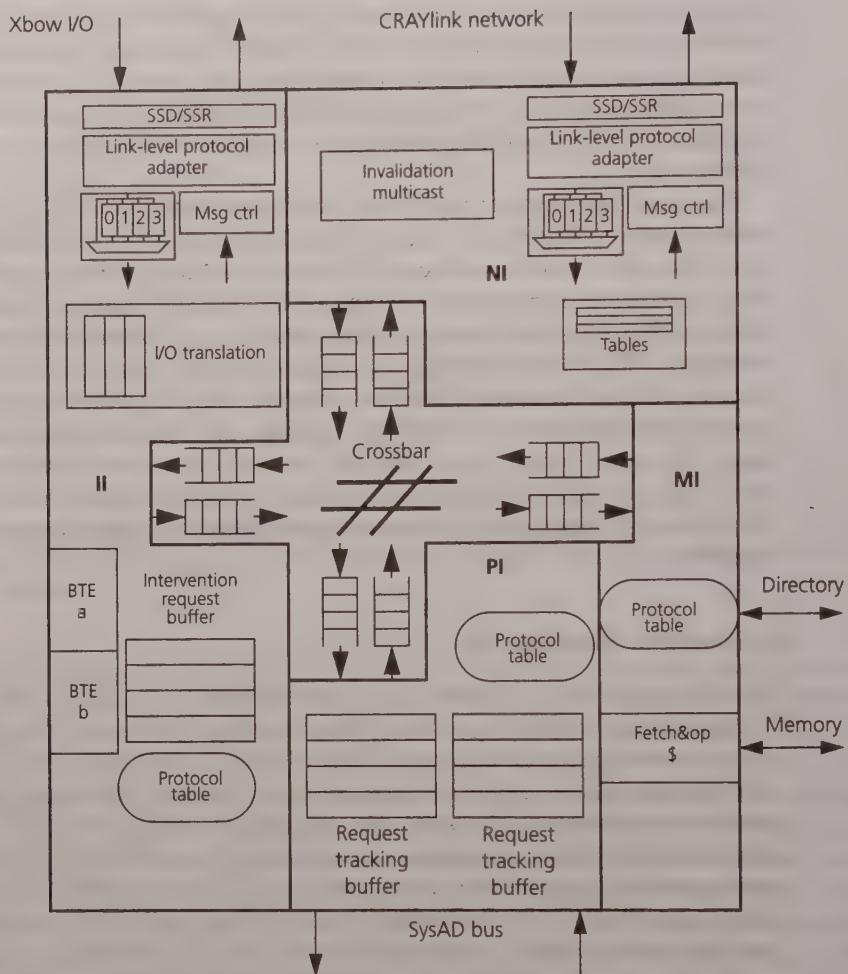


FIGURE 8.21 Layout of the Hub chip. The crossbar at the center connects the buffers of the four different interfaces. Clockwise from the bottom left, the BTEs are the block transfer engines. The top left corner is the I/O interface or II (the SSD and SSR translate signals to and from the I/O ports). Next is the network interface (NI), including the routing tables. The bottom right is the memory/directory interface (MI), and at the bottom is the processor interface (PI) with its request tracking buffers.

from either the processors in the node or from another interface. Since the order of transactions closing is not deterministic, a new transaction must go into any available slot, so these tracking buffers are implemented as fully associative rather than FIFO buffers (the queues that hold the actual messages are FIFO). The buffer look-ups determine whether the PI should issue a request to either a processor or the other interfaces.

The PI is a good example of the shielding provided by interfaces. If the processor (or cache) provides data as a reply to an incoming intervention, it is the logic in the PI's outgoing FIFO that expands the reply into the two responses required by the protocol, one to the home as a sharing write-back revision message and one to the requestor. The processor itself does not have to be modified to generate two replies. Another example is in the mechanisms used to keep track of and match incoming and outgoing requests and responses. All requests passing through the PI in either direction are given request numbers, and responses carry these request numbers as well. However, the processor itself does not know about request numbers, and it is the PI's job to ensure that when it passes on incoming requests (interventions or invalidations) to the processor, it can match the processor's responses to the outstanding interventions/invalidations without the processor having to deal with request numbers.

The Memory/Directory Interface (MI)

The MI also has FIFOs between it and the Hub crossbar. The FIFO from the Hub crossbar to the MI separates headers from data so that the header of the next message can be examined by the directory while the current one is being serviced; this allows writes to be pipelined and performed at peak memory bandwidth. The MI also contains a directory interface, a memory interface, and a controller. The directory interface contains the logic and tables that determine what protocol actions to take and hence implement the coherence protocol. It also contains the logic that generates outgoing message headers, while the memory interface contains the logic that generates outgoing message data. Both the memory and directory RAMS have their own address and data buses. Some messages, like revision messages coming to the home, may not access the memory but only the directory.

On a read request, the read is issued to memory at the home speculatively, simultaneously with starting the directory operation. The directory state is available a cycle before the memory data, and the controller uses this (plus the message type and initiator) to look up the directory protocol table. This hardwired table directs the controller to the action to be taken and the message to send. The directory block sends the latter information to the memory interface, where the message headers are assembled and inserted into the outgoing FIFO together with the data returning from memory. The directory lookup itself is a read-modify-write operation. For this, the MI provides support for partial writes of memory blocks and a one-entry merge buffer to hold the bytes from the time they are read from memory to the time they are written back. Finally, to speed up the at-memory fetch&op accesses provided for synchronization, the MI contains a four-entry LRU fetch&op cache to hold the data for recent fetch&op variables and, hence, to avoid a memory or directory access. This reduces the best-case serialization time at memory for a fetch&op to 41 ns, about four 100-MHz Hub cycles.

The Network Interface (NI)

The NI interfaces the Hub crossbar to the network router for that node. The router and the Hub internals use different data transport formats, protocols, and speeds (100 MHz in the Hub versus 400 MHz in the router), so one major function of the NI is to translate between the two. Toward the router side, the NI implements a flow control mechanism to avoid network congestion (Singh 1997). The FIFOs between the NI and the network also implement separate virtual FIFOs for requests and responses, thus implementing separate virtual networks. The outgoing FIFO also has an invalidation destination generator that takes the bit vector of nodes to be invalidated and generates individual messages for them, a routing table that pre-determines the routing decisions based on source and destination nodes, and virtual channel selection logic.

8.5.7 Performance Characteristics

The peak hardware bandwidths of the Origin2000 system were stated earlier: 780-MB/s SysAD bus, 670-MB/s local memory, and 780-MB/s node-to-network each way. The occupancy of the Hub at the home for a transaction on a cache block is about 20 Hub cycles (about 40 processor cycles), though it varies between 18 and 30 Hub cycles depending on whether successive directory pages accessed are in the same bank of the directory RAM and on the exact pattern of successive transactions. The latencies of memory operations depend on many factors, such as the type of operation, whether the home is local or not, where and in what state the data is currently cached, and how much contention there is for resources along the way. The latencies can be measured using microbenchmarks. Let us examine microbenchmark results for latency and bandwidth first, followed by the performance and scaling of our six parallel applications.

Characterization with Microbenchmarks

Unlike the MIPS R4400 processor used in the SGI Challenge, the Origin's MIPS R10000 processor is dynamically scheduled and does not stall on a read miss. This makes it more difficult to measure read latency, raising an interesting methodological issue. We cannot, for example, measure the unloaded latency of a read miss by simply executing the microbenchmark from Chapter 4 that reads the elements of an array with stride greater than the cache block size. Since the misses are to different locations, subsequent misses will simply be overlapped with one another and the processor will not see their full latency. Instead, this microbenchmark will give us a measure of the throughput that the system can provide on successive read misses issued from a processor. The throughput is the inverse of the latency remaining after overlap, which we can call the *pipelined latency*.

To measure the full latency, we need to ensure that subsequent operations are dependent on each other. To do this, we can use a microbenchmark that chases pointers down a linked list: the address for the next read is not available to the pro-

Table 8.1 Back-to-Back and True Unloaded Latencies for Different System Sizes

Where Miss is Satisfied	Network Routers Traversed	Back-to-Back Latency (ns)	True Unloaded Latency (ns)
L ₁ cache	0	5.5	5.5
L ₂ cache	0	56.9	56.9
Local memory	0	472	329
4P remote memory	1	690	564
8P remote memory	2	890	759
16P remote memory	3	991	862

The first column shows where in the extended memory hierarchy the misses are satisfied. For the 8P case, for example, the misses are satisfied in the node furthest away from the requestor in a system of 8 processors. Given the Origin2000 topology, this means traversing through two network routers in this case.

cessor until the previous read (of the pointer) completes, so the reads cannot be overlapped. However, it turns out this is a little pessimistic in determining the unloaded read latency. The reason is that the processor implements critical word restart; that is, it can use the value returned by a read as soon as that word is returned to the processor, without waiting for the rest of the cache block to be loaded in the caches. With the pointer-chasing microbenchmark, the next read will be issued before the previous block has been loaded and will contend for cache access with the loading of the rest of that block. The latency obtained from this microbenchmark, which includes this contention, can be called *back-to-back latency* (one read miss issued just as the previous one completes). Avoiding this contention between successive accesses requires that we put some computation between the read misses; the computation should depend on the data being read, so it cannot execute in parallel with the read miss, but should not access the cache between two misses. The goal is to have this computation overlap the time it takes for the rest of the cache block to load into the caches after a read miss so that the next read miss will not have to stall on cache access. The time for this overlap computation must, of course, be subtracted from the elapsed time of the microbenchmark to measure the true unloaded read-miss latency, assuming critical word restart. We can call this the *true unloaded latency*. Table 8.1 shows the back-to-back and true unloaded latencies measured on the Origin2000. Only one processor executes the microbenchmark, but the data that is accessed is distributed among the memories of different numbers of processors. The back-to-back latency is usually about 13 SysAD bus cycles (133 ns) longer because the L₂ cache block size (128 B) is 12 double words longer than the L₁ cache block size (32 B) and there is one cycle for bus turnaround.

Table 8.2 lists the back-to-back latencies for different initial states of the block being referenced (Hristea, Lenoski, and Keen 1997). Recall that the owner node is the home node when the block is in unowned or shared state at the directory and is the node that has a cached copy when the block is in exclusive state. The true unloaded latency for the case where both the home and the owner are the local node

Table 8.2 Back-to-Back Latencies (in ns) for Different Initial States of the Block

Home	Owner	State of Block		
		Unowned	Clean-Exclusive	Modified
Local	Local	472	707	1,036
Remote	Local	704	930	1,272
Local	Remote	472	930	1,159
Remote	Remote	704	917	1,097

The first column indicates whether the home of the block is local or not, the second indicates whether the current owner is local or not, and the last three columns give the latencies for the block being in different states. Of course, the owner node should be ignored for the unowned state.

(i.e., if the block is owned by main memory, the other processor in the same node) is 338 ns for the unowned state, 656 ns for the clean-exclusive state, and 892 ns for the modified state. Note that no contention is encountered with operations from other processors in this microbenchmark; latencies under real workloads will be larger.

Application Speedups

Figure 8.22 shows the speedups for the six parallel applications on a 32-processor Origin2000, using two problem sizes for each application. We see that most of the applications speed up well, especially once the problem size is large enough. The dependence on problem size is particularly stark in applications like Ocean and Raytrace. The exceptions to good speedup at this scale are Radiosity and, most notably, Radix. In the case of Radiosity, even the larger problem is relatively small for a machine of this size and power. We can expect to see better speedups for larger scenes. For Radix, the problem is the highly scattered, bursty pattern of writes in the permutation phase. These writes are mostly to locations that are allocated remotely, and the flood of requests to and from the directories, invalidations, acknowledgments, and replies that they generate causes tremendous contention and hot spotting at Hubs and memories. Running larger problems doesn't alleviate the situation since there is no other computation than the data permutation during this phase, and the communication-to-computation ratio is essentially independent of problem size; in fact, the situation worsens once a processor's partition of the keys does not fit in its cache, at which point frequent write-back transactions are also thrown into the mix. For applications like Radix (and an FFT, not shown) that exhibit all-to-all bursty communication, the fact that two processors share a Hub and two Hubs share a router also causes contention at these resources, despite their high peak bandwidths (Jiang and Singh 1998). For these applications, the machine would perform better if it had only a single processor per Hub and per router. However, the sharing of resources does reduce cost and does not get in the way of the other applications.

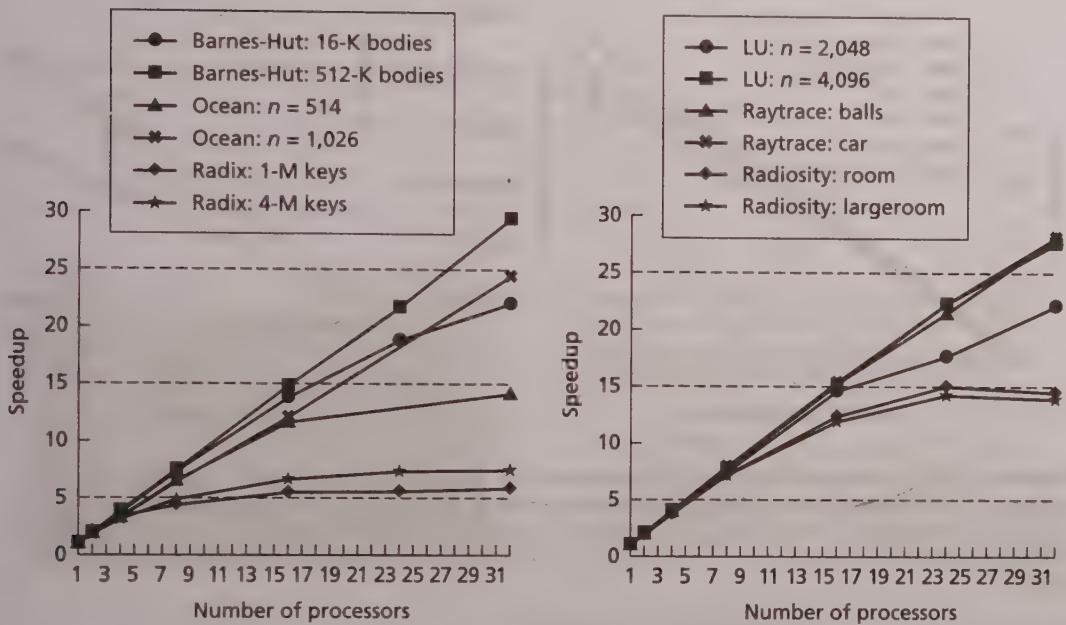


FIGURE 8.22 Speedups for the parallel applications on the Origin2000. Two problem sizes are shown for each application. The Radix sorting program does not scale well, and the Radiosity application is limited by the available input problem sizes. The other applications speed up quite well when reasonably large problem sizes are used.

Breakdowns of execution time into components on a per-processor basis on this machine were shown in Chapters 3 and 4, giving us a good idea of where time is spent.

Scaling

Figure 8.23 shows the speedups under different scaling models for the Barnes-Hut galaxy simulation on the Origin2000. The results are quite similar to those on the SGI Challenge in Chapter 6—although extended to more processors—and the analysis there largely applies. For applications like Ocean (not shown), in which an important working set is proportional to the data set size per processor, machines like the Origin2000 display an interesting effect in comparing scaling models when we start from a problem size where the working set does not fit in the cache on a uniprocessor. Under PC and TC scaling, the data set size per processor diminishes with an increasing number of processors. Thus, although the communication-to-computation ratio increases, we observe superlinear speedups once the working set starts to fit in the cache (since the performance within each node becomes much

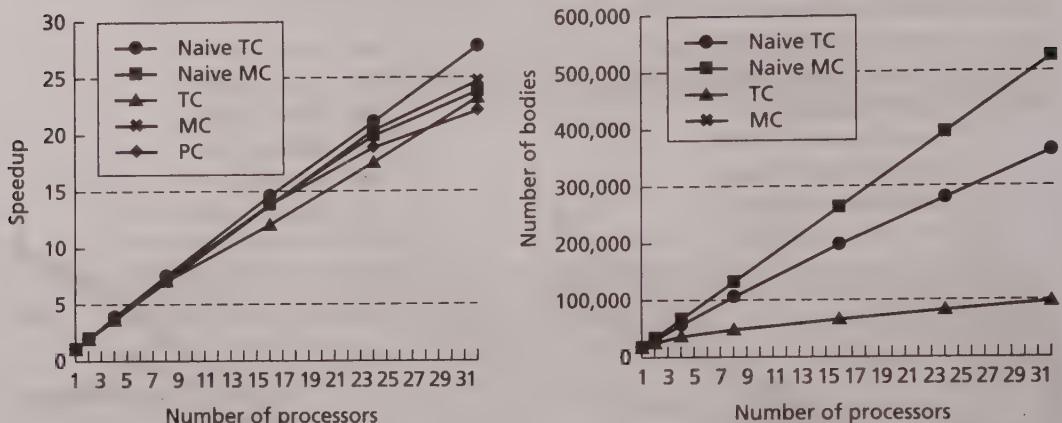


FIGURE 8.23 Scaling of speedups and number of bodies simulated under different scaling models for the Barnes-Hut galaxy simulation on the Origin2000. As with the results for bus-based machines in Chapter 6, the speedups are very good under all scaling models, and the number of bodies that can be simulated grows much more slowly under realistic TC scaling than under MC or naive TC scaling.

better when the working set fits in the cache). Under MC scaling, the communication-to-computation ratio does not change, but neither does the working set size per processor. As a result, although the demands on the communication architecture scale more favorably under MC scaling than under TC or PC scaling (the capacity misses due to the working sets are almost entirely local), speedups are not so good because the beneficial effect on node performance of the working set suddenly fitting in the cache is no longer observed. Also, even local capacity misses occupy the Hub and memory, contributing to contention.

8.6 CACHE-BASED DIRECTORY PROTOCOLS: THE SEQUENT NUMA-Q

The flat, cache-based directory protocol described in our second case study is the IEEE standard Scalable Coherent Interface (SCI) protocol (Gustavson 1992). As a case study of this protocol, we examine the NUMA-Q machine from Sequent Computer Systems, Inc., a machine targeted toward commercial workloads such as databases and transaction processing (Lovett and Clapp 1996). This machine relies heavily on third-party commodity hardware, using stock Intel SMPs as the processing nodes, stock I/O links, and the DataPump network interface from Vitesse Semiconductor Corporation to move data between the node and the network. The only customization is in the IQ-Link board used to implement the SCI directory protocol. A similar directory protocol is also used (with much more customization) in the Convex Exemplar series of machines (Convex Computer Corporation 1993; Thekkath et al. 1997), which, like the SGI Origin, is targeted more toward scientific computing.

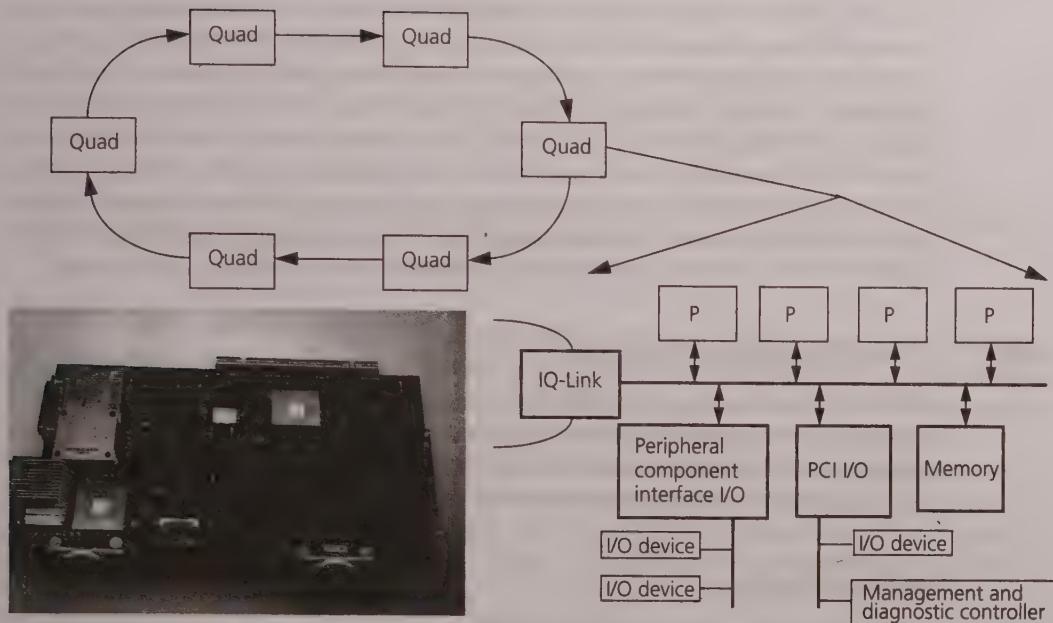


FIGURE 8.24 Block diagram of the Sequent NUMA-Q multiprocessor. The diagram shows the high-level organization of the machine, both across nodes and within a node. The photograph shows an IQ-Link board. Source: Photo courtesy of Sequent Computer Systems, Inc.

NUMA-Q is a collection of homogeneous processing nodes interconnected by high-speed links in a ring configuration (Figure 8.24). Each processing node is an inexpensive Intel quad bus-based multiprocessor with four Intel Pentium Pro microprocessors, which illustrates the use of high-volume SMPs as building blocks for larger systems. Systems from Data General (Clark and Alnes 1996) and from HAL Computer Systems (Weber et al. 1997) also use Pentium Pro quads as their processing nodes, the former also using an SCI protocol similar to NUMA-Q across quads and the latter using a memory-based protocol inspired by the Stanford DASH protocol. (In the Convex Exemplar series, the individual nodes connected by the SCI protocol are not bus based but are small directory-based multiprocessors kept internally coherent by a different directory protocol.) We described the quad SMP node in Chapter 1 (see Figure 1.17) and so do not discuss it further.

The IQ-Link board in each quad plugs into the quad memory bus and takes the place of the Hub in the SGI Origin. In addition to the directory logic and storage and the datapath between the quad bus and the network, it also contains a large (expandable) 32-MB, four-way set-associative remote access cache for blocks that are fetched to the node from remote memory. This *remote access cache*, hereafter called the *remote cache*, represents the quad to the cross-node SCI directory protocol. It is

the only cache in the quad that is visible to that protocol; the individual processor caches are kept coherent with the remote cache through the snooping bus protocol within the quad. The directory protocol is for the most part oblivious to how many processors there are within a node and even to the bus protocol itself. Inclusion is preserved between the remote cache and the processor caches within the node, so if a block is replaced from the remote cache it is invalidated in the processor caches, and if a block is placed in modified state in a processor cache then the state in the remote cache reflects this. The cache block size of the remote cache is 64 bytes, which is therefore the granularity of both communication and coherence across quads.

8.6.1 Cache Coherence Protocol

While two interacting coherence protocols are used in the Sequent NUMA-Q machine, this section focuses on the SCI directory protocol across remote caches and ignores the multiprocessor nature of the quad nodes. Interactions with the snooping MESI protocol within the quads are discussed in Section 8.6.5.

Directory Structure

The directory structure of SCI is the flat, cache-based distributed doubly linked-list scheme that was described in Section 8.2.3 and illustrated in Figure 8.8. There is a linked list of sharers per block, and the pointer to the head of this list is stored with the main memory that is the home of the corresponding memory block. An entry in the list corresponds to a remote cache in a quad. The remote cache is stored in synchronous DRAM memory in the IQ-Link board of that quad, together with the forward and backward pointers for the list. Figure 8.25 shows a simplified representation of a list. The first element (node) is called the head of the list and the last node the tail. The head node has both read and write permission on its cached block whereas the other nodes have only read permission (except in a special-case extension, called pairwise sharing, that we discuss briefly in Section 8.6.3). The pointer in a node that points to its neighbor in the direction toward the tail of the list is called the forward or downstream pointer, and the other is called the backward or upstream pointer. Let us see how the cross-node SCI coherence protocol uses this directory representation.

States

Since processor caches are not visible to the directory protocol, and since a block never enters the remote cache at its home node, unlike in the Origin, the directory protocol in the NUMA-Q does not keep track of cached copies at the home. Keeping the copy in the home memory coherent with these cached copies is the job of the bus protocol. A block in main memory can be in one of three directory states whose names are defined by the SCI protocol as follows. The states are similar to but not the same as the directory states in the Origin protocol.



FIGURE 8.25 An SCI sharing list. Each element of the list in NUMA-Q is a multiprocessor node, represented by its remote cache.

- **Home:** No remote cache (quad) in the system contains a copy of the block (of course, a processor cache in the home quad itself may have a copy since this is not visible to the SCI coherence protocol but is managed by the bus protocol within the quad). This is like the unowned directory state in the Origin.
- **Fresh:** One or more remote caches may have a read-only copy, and the copy in memory is valid. This is like the shared state in the Origin.
- **Gone:** Another remote cache contains a writable (exclusive or dirty) copy. No valid copy exists on the local node. This is like the exclusive directory state in the Origin.

Consider the cache states for blocks in a remote cache. While the processor caches within a quad use the standard MESI stable states, the SCI scheme that governs the remote caches has a large number of possible cache states. In fact, 7 bits are used to represent the state of a block in a remote cache, and the standard describes 29 stable states and many pending (busy) or transient states. Each stable state can be thought of as having two parts, which is reflected in the naming structure of the states. The first part describes where that cache entry is located in the sharing list for that block. This may be ONLY (for a single-node list), HEAD, TAIL, or MID (which means neither the head nor the tail of a multiple-node list). The second part describes the actual state of the cached block. This includes states like dirty (modified and writable); clean (unmodified, same contents as memory, but writable, like the exclusive state in MESI); fresh (data may be read but may not be written until memory is informed); copy (unmodified and readable); and several others. A full description can be found in the SCI standards document (IEEE Computer Society 1993). We shall encounter some of these states (such as HEAD-DIRTY, TAIL-CLEAN, etc.) as we go along.

The SCI standard defines three primitive operations that can be performed on a distributed sharing list. Memory operations such as read misses, write misses, write backs, and replacements are implemented using these three primitive operations:

1. **List construction:** adding a new node (sharer) to the head of a sharing list.
2. **Rollout:** removing a node from a list, which requires that a node communicate with its upstream and downstream neighbors, informing them of their new neighbors so they can update their pointers.
3. **Purging (invalidation):** the node at the head may purge or invalidate all other nodes, thus resulting in a single-element list. Only the head node of a list can issue a purge.

The SCI standard also describes three levels of increasingly sophisticated SCI protocols. The *minimal* protocol does not permit even read sharing; that is, only one node at a time can have a cached copy of a block. The *typical* protocol is what most systems are expected to implement. It has provisions for read sharing (multiple copies), efficient access to data that is in FRESH state in memory, as well as options for efficient DMA transfers and robust recovery from errors. Finally, the *full* protocol implements all of the options defined by the standard, including optimizations for pairwise sharing between only two nodes and queue-on-lock-bit (QOLB) synchronization (to be discussed later). The NUMA-Q system implements the typical protocol, and this is the one we discuss. Let us see how different types of memory operations—read misses, write misses, and replacements (including write backs)—are handled. In each case, the identity of the home node is first determined from the address of the block.

Handling Read Requests

Suppose the read request needs to be propagated off quad. We can think of this node's remote cache as the requesting cache as far as the SCI protocol is concerned. The requesting cache first allocates an entry for the block if necessary and sets the cache state of the block to a pending (busy) state; in this state, it will not process other requests for that block that come to it. (The SCI protocol often puts cached blocks in busy states at requestors in this way, to keep transactions for a block atomic and to facilitate serialization, much like the Origin protocol did with its busy states at the directory. However, it does not use NACKs, as we shall see.) It then begins a list construction operation to add itself to the head of the sharing list by sending a request to the home node. When the home receives the request, its block may be in one of the three directory states identified earlier: HOME, FRESH, or GONE.

If the directory state is HOME, there are no cached copies and the copy in memory is valid. On receiving the read request, the home updates its state for the block to FRESH and sets its head pointer to point to the requesting node. The home then replies to the requestor with the data, which upon receipt updates its state from PENDING to ONLY_FRESH. All actions at a node in response to a given transaction are atomic (the processing for one is completed before the next one is handled), and a strict request-response protocol is followed in all cases (unlike in Origin).

If the directory state is FRESH, there is already a sharing list, but the copy at the home is also valid. The home changes its head pointer to point to the requesting cache instead of the previous head of the list. It then sends back a transaction to the requestor containing the data as well as a pointer to the previous head. On receipt, the requestor moves to a different pending state and sends a transaction to that previous head asking to be attached as the new head of the list (the list construction operation). The previous head reacts to this message by changing its state from HEAD_FRESH to MID_VALID or from ONLY_FRESH to TAIL_VALID as the case may be, updating its backward pointer to point to the requestor and sending an acknowledgment to the requestor. When the requestor receives this acknowledgement, it sets its forward pointer to point to the previous head and changes its state

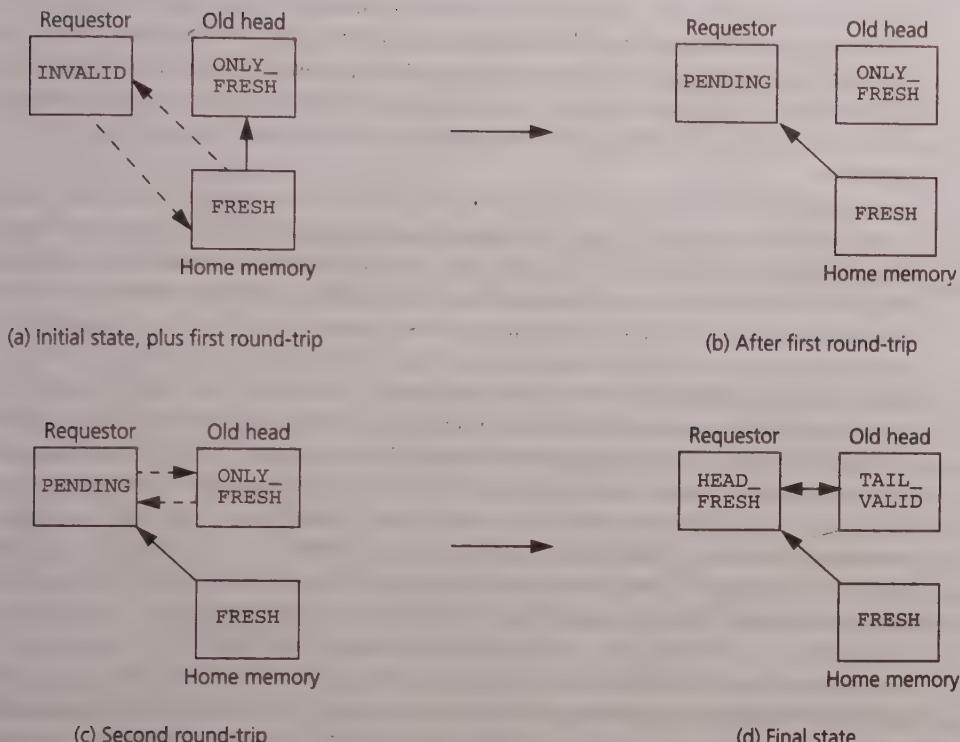


FIGURE 8.26 An example of a read miss in the SCI protocol. The figure shows the messages and state transitions for a read miss to a block that is initially in the FRESH state at home, with one node on the sharing list. Solid lines are the pointers in the sharing list, whereas dotted lines represent network transactions. Null pointers are not shown.

from the pending state to HEAD_FRESH. The sequence of transactions and actions is shown in Figure 8.26 for the case where the previous head is in state HEAD_DIRTY when the request comes to it.

If the directory state is GONE, the cache at the head of the sharing list has an exclusive (clean or modified) copy of the block. Now, the memory does not reply with the data but simply stays in the GONE state and sends a pointer to the previous head back to the requestor. The requestor goes to a new pending state and sends a request to the previous head, asking both for the data and to attach to the head of the list (list construction). The previous head changes its state from HEAD_DIRTY to MID_VALID or from ONLY_DIRTY to TAIL_VALID (or whatever is appropriate), sets its backward pointer to point to the requestor, and returns the data to the requestor. (The data may have to be retrieved from a processor cache in the previous head node.) The requestor then updates its copy, sets its state to HEAD_DIRTY, and sets its forward pointer to point to the new head, all in a single atomic action as always. Note that even though the reference was a read, the head of the sharing list is

left in HEAD_DIRTY state. This does not have the standard meaning of dirty that we are familiar with; that is, that the head node can write that data without having to invalidate any other caches. It means that it can indeed write the data into the cache without communicating with the home (and even before sending out the invalidations), but it must invalidate the other nodes in the sharing list since they are in valid state.

It is possible to fetch a block in HEAD_DIRTY state even when the directory state is not GONE, for example, when the requesting node is expected to write that block soon afterward. In this case, if the directory state is FRESH the memory returns the data to the requestor, together with a pointer to the old head of the sharing list, and then puts itself in GONE state. The requestor then prepends itself to the sharing list by sending a request to the old head and puts itself in the HEAD_DIRTY state. The old head changes its state from HEAD_FRESH to MID_VALID or from ONLY_FRESH to TAIL_VALID as appropriate, and other nodes on the sharing list remain unchanged.

In the preceding cases, a requestor is always directed by the home to the old head. It is possible that the old head (let's call it A) is in a pending state when the request from the new requestor (B) reaches it since it may itself have a memory operation outstanding on that block. This is dealt with not by buffering the request at the old head or NACKing it but by extending the sharing list backward into a (still distributed) *pending list*. That is, node B will indeed be physically attached to the head of the list but in a pending state waiting to truly become the head. If another node C now makes a request to the home, it will be forwarded to node B and will also attach itself to the pending list (the home will now point to C, so subsequent requests will be directed there, and so on). At any time, we call the “true head” (here A) simply the *head* of the sharing list, we call the part of the list before the true head the *pending list*, and we call the latest element to have joined the pending list (here C) the *pending head* (see Figure 8.27). When A leaves the pending state and completes its operation, it will pass on the “true head” status to B, which will in turn pass it on to C when its request is completed. Note also that, unlike in the Origin, no pending or busy state exists at the directory, which always simply takes atomic actions to change its state and head pointer and returns the previous state/pointer information to the requestor, a point we will revisit when discussing how correctness issues are addressed.

Handling Write Requests

The head node of a sharing list is assumed to always have the latest copy of the block (unless the head node is in a pending state). Thus, only the head node is allowed to write a block and issue invalidations. When a node incurs a write miss, three cases are possible. In the first case, the writer is already at the head of the list, but it does not have the sole modified copy (e.g., there may be other sharers). It first ensures that it is in the appropriate state for this case, by communicating with the home if necessary (and in the process ensuring that the home block is already in or transitions to the GONE state). It then modifies the data locally and invalidates the rest of the nodes in the sharing list. (This case is elaborated on in the next two para-

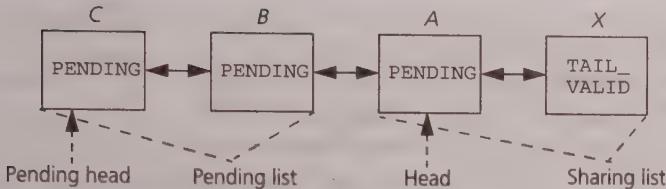


FIGURE 8.27 Pending lists in the SCI protocol. The pending list is a continuation (in the reverse direction) of the regular sharing list. The true head (called the head) and the nodes in the pending list are in pending states.

graphs.) In the second case, the writer is not in the sharing list at all. The writer must first allocate space for and obtain a copy of the block, then add itself to the head of the list using the list construction operation, and then perform the preceding steps to complete the write. The third case is when the writer is in the sharing list but not at the head. In this case, it must remove itself from the list (rollout), then add itself to the head (list construction), and finally perform the preceding steps. We discuss rollout further in the context of replacement, where it is also needed, and we have already seen list construction. Let us focus now on the case where the writing node is already at the head of the list.

If the block is in the HEAD_DIRTY state in the writer's cache, it is modified right away (since the directory must already be in GONE state) and then the writing node purges the rest of the sharing list. The purge operation is done in a serialized request-response manner: an invalidation request is sent to the next node in the sharing list, which rolls itself out from the list and sends back to the head a pointer to the next node in the list. The head then sends this node a similar request, and so on until all entries are purged (i.e., until the response to the head contains a null pointer; see also Figure 8.28). The writer, or head node, stays in a pending state while the purging is in progress. During this time, new attempts to add to the sharing list are delayed in a pending list as usual. The latency of purging a sharing list is a few serialized round-trips (invalidation request, acknowledgment, and the rollout transactions) plus the associated actions per sharing list entry, so it is important that long sharing lists are not encountered often on writes. It is possible to reduce the number of network transactions in the critical path by having each node pass on an invalidation request to the next node and perhaps acknowledge the previous node rather than return the identity to the writer. This is not part of the SCI standard since it distributes the state of the invalidation progress and hence complicates protocol-level recovery from errors; however, practical systems may be tempted to take advantage of this shortcut, especially if sharing lists are long.

If the writer is the head of the sharing list but has the block in HEAD_FRESH state, then it must be changed to HEAD_DIRTY before the block can be modified and the rest of the entries purged. The writer goes into a pending state and sends a request to the home, the home changes from FRESH to GONE state and replies to the message, and then the writer goes into a different pending state and purges the rest

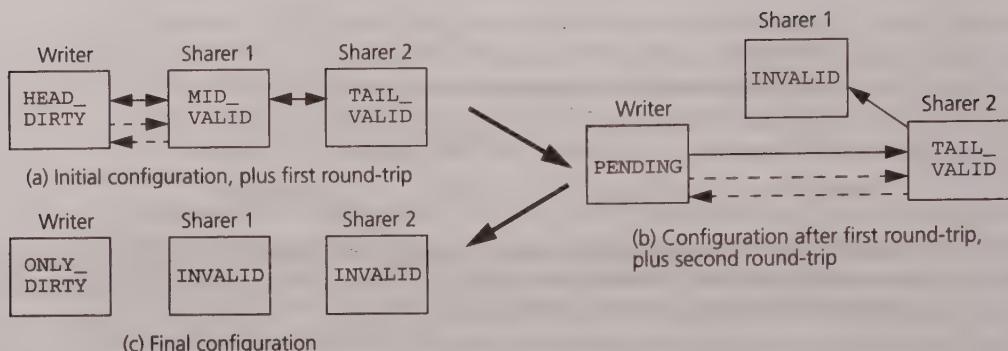


FIGURE 8.28 Purging a sharing list from a **HEAD_DIRTY** node in SCI. Solid arrows connecting list nodes are list pointers, while dashed arrows indicate network transactions that implement the transition to the next configuration.

of the blocks as was just described. It may be that when the request reaches the home the home is no longer in FRESH state, but it points to a newly queued node that got there in the meantime and has been directed to the writer. When the home looks up its state, it detects this situation and sends the writer a corresponding response that is like a NACK. When the writer receives this response, based on its local pending state it deletes itself from the sharing list (how it does this, given that a request is coming at it, is discussed in the next subsection) and tries to reattach as the head in HEAD_DIRTY or ONLY_DIRTY state by sending the appropriate new request to the home. This is not a retry, in the sense that the writer does not try the same request again, but is a suitably modified request to reflect the new state of itself and the home (similar to modifying an upgrade to a read exclusive in the race condition due to nonatomic state transitions discussed in Chapter 6). The last case for a write by a head node is if the writer has the block in ONLY_DIRTY state, in which case it can modify the block without generating any network transactions.

Handling Write-Back and Replacement Requests

A node that is in a sharing list for a block may need to delete itself, either because it must become the head in order to perform a write operation, or because it must be replaced in its remote cache for capacity or conflict reasons, or because it is being invalidated. In the case of a replacement, even if the block is in shared state and does not have to write data back, the space in the cache (and the pointers) will now be used for another block and its list pointers, so to preserve a correct representation the block being replaced must be removed from its sharing list. These replacements and list removals use the rollout operation.

Consider the general case of a node trying to roll out from the middle of a sharing list. The node first sets itself to a pending state, then sends a request each to its upstream and downstream neighbors asking them to update their forward and back-

ward pointers, respectively, to skip that node. The pending state is needed since there is nothing to prevent two adjacent nodes in a sharing list from trying to roll themselves out at the same time, which can lead to a race condition in the updating of pointers. Even with the pending state, if two adjacent nodes indeed try to roll out at the same time, they may set themselves to pending state simultaneously and send messages to each other. This can cause deadlock since neither will respond while it is in pending state. A simple priority system is used to avoid such deadlock: by convention, the node closer to the tail of the list has priority and is rolled out first. The roll-out operation is completed by setting the state of the rolled-out cache entry to invalid when both the neighbors have replied. The neighbors of the node that is rolling out do not have to change their state except when the node being rolled out is the second in a two-node list; in that case, the head of the list may change its state from HEAD_DIRTY or HEAD_FRESH to ONLY_DIRTY or ONLY_FRESH as appropriate.

If the entry to be rolled out is the head of the list, then the entry may be in dirty state (a write back) or in fresh state (a replacement). The same set of transactions is used in either case. The head puts itself in a pending state and first sends a transaction to its downstream neighbor. This causes the latter to set its backward pointer to the home memory and change its state appropriately (e.g., from TAIL_VALID or MID_VALID to HEAD_DIRTY or from MID_FRESH to HEAD_FRESH). When the replacing (head) node receives a response, it sends a transaction to the home, which updates its pointer to point to the new head but need not change its state. The home sends a response to the replacer, which is now out of the list and sets its state to INVALID. Of course, if the replacer is the only node in the list, then it needs to communicate only with memory, which will set its state to HOME.

This scenario of a head node rolling out provides another example of the state at the recipient of a request not being compatible with that request when it arrives. By the time the message from the replacer gets to the home, the home may have set its head pointer to point to a different node X from which it has received a request for the block in the interim. In general, whenever a transaction comes in, the recipient looks up its local state and the incoming request type; if it detects a mismatch, the general strategy adopted by the protocol is as we saw earlier in the example of a write to a block in HEAD_FRESH state: the recipient does not perform the operation that the request solicits but issues a response that is a lot like a NACK. The requestor will then check its local state again and take an appropriate action. In this specific case, the home detects that the incoming transaction type requires that the requestor be the current head; this is not true, so it NACKs the request. The replacer keeps retrying the request to the home and keeps being NACKed. At some point, the request from node X that was redirected to the replacer will reach the replacer, asking to be prepended to the list. The replacer will look up its (pending) state and send a response to that requestor, telling it to instead go to the downstream neighbor (the real head since the replacer is rolling out of the list). The replacer is now off the list and in a different pending state; it is waiting to go to INVALID state, which it will do when the next NACK from the home reaches it. Thus, the SCI protocol does include NACKs, but not in the traditional sense of asking requests to retry when a node or

resource is busy. NACKs are used just to indicate inappropriate requests and facilitate changes of state at the requestor; the difference is that in this case a request that is NACKed will never succeed in its original form but may cause a new type of request to be generated, which may succeed.

Finally, when a block needs to be written back upon a miss, an important performance question is whether the miss should be satisfied first or the block should be written back first. In discussing bus-based protocols, we saw that most often the miss is serviced first and the block to be written back is put in a write-back buffer. In NUMA-Q, the simplifying decision is made to service the write back (rollout) first and only then satisfy the miss. Although this slows down the miss, the complexity of the buffering solution is greater here than in bus-based systems (where the write-back buffer can simply be snooped). Also, the replacements and hence write backs we are concerned with here are from the remote cache, which is large enough (tens of megabytes) that replacements are likely to be very infrequent.

8.6.2 Dealing with Correctness Issues

A major emphasis in the SCI standard is providing well-defined, uniform mechanisms for preserving serialization, resolving race conditions, and avoiding deadlock, livelock, and starvation. The standard takes a stronger position on starvation and fairness than many other coherence protocols. It was mentioned earlier that most of the correctness considerations are satisfied by the use of distributed lists of sharers as well as pending requests, but let us look at how this works in more detail.

Serialization of Operations to a Given Location

In the SCI protocol, the home node is the entity that determines the order in which cache misses to a block are serialized. However, unlike in the Origin protocol, here the order is that in which the requests first arrive at the home, and the mechanism used for ensuring this order is very different. There is no busy state at the home. Generally (except for some race conditions described earlier), the home accepts every request that comes to it, either satisfying it wholly by itself or directing it to the node that it sees as the current head of the sharing list (the pending head if there is a pending list). Before it directs the request to another node, it first updates its head pointer to point to the current requestor. The next request for the block from any node will see the updated state and pointer (i.e., to the current requestor) even though the operation corresponding to the current request is not globally complete. This ensures that the home does not direct two conflicting requests for a block to the same node at the same time, avoiding race conditions. As we have seen, if a request cannot be satisfied at the head node to which it was directed—that is, if that node is in pending state—the requestor will attach itself to the distributed pending list for that block and await its turn as long as necessary (see Figure 8.27). Nodes in the pending list obtain access to the block in FIFO order, ensuring that the order in which they complete is indeed the same as that in which they first reached the home.

While the home may NACK requests when some race conditions are encountered, those requests will never succeed in their current form, so they do not count in the serialization. They may be modified to new, different requests that will succeed, and in that case those new requests will be serialized in the order in which they first reach the home.

Memory Consistency Model

The SCI standard defines both a coherence protocol and a transport layer, including a network interface design. However, it does not specify many other aspects, like details of the physical implementation or even the memory consistency model. Such matters are left to the system implementor. NUMA-Q does not satisfy sequential consistency but uses a more relaxed memory consistency model called *processor consistency* that we shall discuss in Section 9.1. Interestingly, as in Origin, the consistency model chosen for the system is the one supported by the underlying microprocessor.

Deadlock, Livelock, and Starvation

The fact that a distributed pending list is used to hold waiting requests at the requestors themselves, rather than a hardware queue shared at the home node by all blocks allocated in it, implies that there is no danger of input buffers filling up and, hence, no deadlock problem at the protocol level. A strict request-response protocol is used as well. Since requests are not NACKed from the home to alleviate blockages or contention (only under certain race conditions when they must be altered) but will simply join the pending list and always make progress, livelock does not occur. The list mechanism also ensures that the requests are handled in FIFO order as they first come to the home, thus preventing starvation.

The total number of pending lists that a node can be a part of is the number of requests it can have outstanding, and the storage for the pending lists is already available in the cache entries, so there is little need for extra buffering at the protocol level. (Replacement of a pending entry is not allowed; the memory operation that causes the replacement stalls until the entry is no longer pending.) While the SCI standard does not take a position on queuing and buffering issues at the lower transport level, most implementations, including NUMA-Q, use separate request and response queues on each of the incoming and outgoing paths.

Error Handling

The SCI standard provides some options in the typical protocol to recover from errors at the hardware link level. NUMA-Q does not implement these but, rather, assumes that the hardware links are reliable. Standard ECC and CRC checks are provided to detect and recover from hardware errors in the memory and network links. Robustness to errors at the protocol level often comes at the cost of performance.

For example, SCI's decision to have the writer send all the invalidations one by one, serialized by responses, simplifies error recovery since the writer knows how many invalidations have been completed when an error occurs; however, it but compromises performance. While NUMA-Q retains this feature, other systems may choose not to.

8.6.3 Protocol Extensions

While the SCI protocol is fair and quite robust to errors, many types of operations can generate several serialized network transactions and therefore become quite expensive. A read miss requires two network transactions with the home, at least two with the head node if there is one, and perhaps more with the head node if it is in pending state. A replacement requires a rollout, which requires communication with both neighbors. But, potentially, the most troublesome operation from a scalability viewpoint is invalidation on a write since the cost of the invalidation scales linearly with the number of nodes on the sharing list with a fairly large constant (more than a round-trip time). The use of distributed pending lists can increase latency too, and, in general, the latency of misses tends to be larger than in memory-based protocols. Extensions have been proposed to SCI to deal with widely shared data through a combination of hardware organization and protocol. For example, instead of a single large ring interconnect, the SCI standard envisions building large systems by connecting many smaller rings together in a hierarchy using bridges and switches; the protocol can exploit combining transactions in this hierarchy. Some extensions require changes to the basic protocol and hardware structures whereas others are compatible with the basic SCI protocol and only require new implementations of the bridges. The complexity of the extensions may reduce performance for low degrees of sharing. They are not finalized in the standard and are beyond the scope of this discussion. More information can be found in (IEEE Computer Society 1995; Kaxiras and Goodman 1996; Kaxiras 1996). One extension that is included in the standard specializes the protocol for the case in which only two nodes share a cache block and they ping-pong ownership of it back and forth between themselves by both writing it repeatedly. This is described in the SCI protocol document (IEEE Computer Society 1993). NUMA-Q includes another protocol extension that is a special protocol operation that enables a processor to obtain a copy of a block even while it is invalidating the (nonhome) source of the block.

Unlike Origin, NUMA-Q does not provide hardware or OS support for dynamic page migration. With the very large remote caches, capacity misses in the processor caches to remotely allocated data are almost always satisfied in the remote cache in the local node. However, proper page placement can still be useful when a processor writes and has to obtain ownership for data. If nobody else has a copy (e.g., in the interior portion of a processor's partition in the equation solver kernel or in Ocean), then if the home is local, obtaining ownership does not generate network traffic; however, if home is remote, a round-trip to the home is needed to look up directory state. The NUMA-Q position is that data migration in main memory is the responsibility of user-level software. The exception is when a process migrates, in which case

the OS uses a heuristic to possibly migrate that process's active pages as well, making them local at the new location. The designers considered this to be the important context for page migration. Similarly, little hardware support is provided for synchronization beyond simple atomic exchange primitives like test&set.

8.6.4 Overview of NUMA-Q Hardware

Within a quad multiprocessor node, the second-level caches per processor currently shipped in NUMA-Q systems are 512 KB or 1 MB large and four-way set associative with a 32-byte block size. The quad bus is a 532-MB/s split-transaction in-order bus, with limited facilities for out-of-order responses that are needed by a two-level coherence scheme. (Even if the bus within an SMP node provides in-order responses, when a request must go to a remote node it is infeasible to have its response be in-order with respect to responses generated within the local node.) A quad also contains up to 4 GB of globally addressable main memory; two 32-bit-wide 133-MB/s peripheral component interface (PCI) buses connected to the quad bus by PCI bridges and to which I/O devices and a memory and diagnostic controller can attach; and the IQ-Link board that plugs into the memory bus and includes the communication assist and the network interface.

In addition to the directory information for locally allocated data and the tags for remotely allocated but locally cached data (which it keeps on both the bus side and the directory side), the IQ-Link board consists of four major functional blocks as shown in Figure 8.29: the bus interface controller, the DataPump, the SCI link interface controller, and the RAM arrays. The *Orion bus interface controller* (OBIC) provides the interface to the shared quad bus, managing the remote cache data arrays and the bus snooping and requesting logic. It acts as both a pseudo memory controller that snoops and translates accesses to nonlocal data as well as a pseudo-processor that puts incoming transactions from the network onto the bus. The *DataPump*, a gallium arsenide chip built by Vitesse Semiconductor Corporation, provides the link and packet-level transport protocol of the SCI standard. It provides an interface to a ring interconnect, pulling off packets that are destined for its quad node and letting other packets go by. The *SCI link interface controller* (SCLIC) interfaces to the DataPump and the OBIC as well as to the interrupt controller and the directory tags. Its main function is to manage the SCI coherence protocol, using one or more programmable protocol engines. The *RAM arrays* implement the data and the different tags needed for the remote cache. These components are described further when we discuss the implementation of the IQ-Link in Section 8.6.6.

For the interconnection across quads, the SCI standard defines both a transport layer and a cache coherence protocol. The transport layer defines a functional specification for a node-to-network interface and a network topology that consists of rings made of point-to-point links. In particular, it defines a 1-GB/s ring interconnect and the transactions that can be generated on it. The NUMA-Q system is initially a single-ring topology of up to eight quads as shown in Figure 8.24. Cables from the quads connect to the ports of a ring that is contained in a single box called the IQ-Plus. Larger systems will include multiple eight-quad systems connected

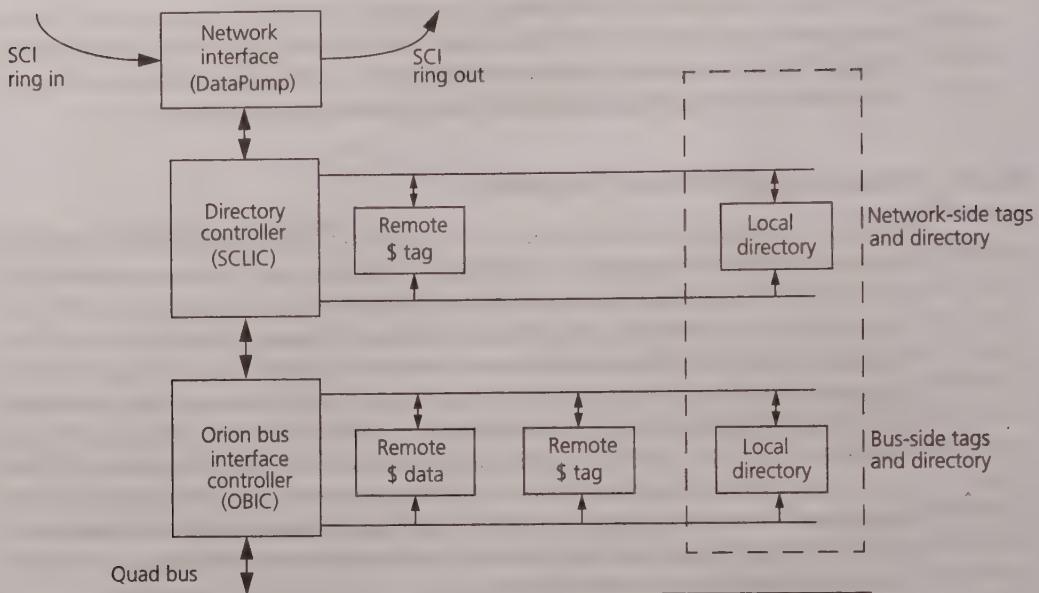


FIGURE 8.29 Functional block diagram of the NUMA-Q IQ-Link board. The remote cache data is implemented in synchronous DRAM (SDRAM). The bus-side tags and directory are implemented in Static RAM (SRAM) whereas the network-side tags and directory can afford to be slower and are therefore implemented in SDRAM.

with local area networks. As mentioned earlier, the SCI standard envisions that, because of the high latency of long rings, larger systems will generally be built out of multiple rings interconnected by switches. With a small number of outstanding requests per node, the latency of a long ring severely limits the node-to-network bandwidth that a node can achieve (see Chapter 11). The transport layer of SCI will be discussed further in Chapter 10.

Since the machine is targeted toward database and transaction processing workloads, I/O is an important focus of the NUMA-Q design. As in Origin, I/O is globally addressable, so any processor can directly write to or read from any I/O device, not just those attached to the local quad. A nonlocal processor does not have to send an explicit message to the quad to which the device is attached and have a processor on that quad issue the access. This is very convenient for commercial applications, which are not often structured so that a processor need only access its local disks. I/O devices are connected to the two PCI buses that attach through PCI bridges to the quad bus. Each PCI bus is clocked at half the speed of the memory bus and is half as wide, yielding roughly one-quarter the bandwidth. Physically, there are two ways for a processor to access I/O devices on other quads. One is through the SCI rings, whether through the cache coherence protocol or through uncached writes, just as Origin does through its Hubs and network. However, bandwidth is a precious resource on a

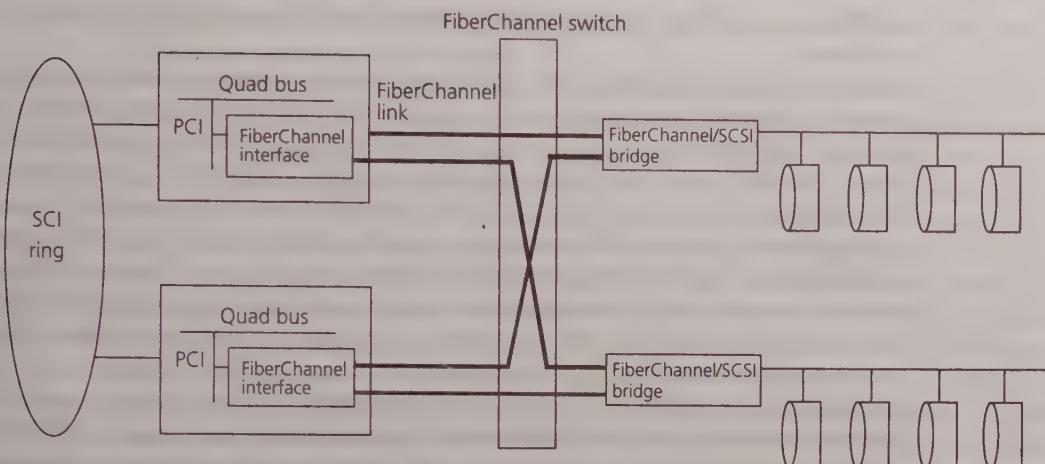


FIGURE 8.30 I/O subsystem of the Sequent NUMA-Q. I/O is globally addressable, and I/O data transfers among nodes can travel through FiberChannel via PCI buses or through the SCI ring used for memory operations.

ring network. I/O transfers can occupy substantial bandwidth, interfering with memory accesses. NUMA-Q therefore provides a separate communication substrate through the PCI buses for interquad I/O transfers, which is the default I/O path. A “FiberChannel” link connects to a PCI bus on each node. These links are connected to all the shared disks in the system through either point-to-point connections, an arbitrated FiberChannel loop, or a FiberChannel switch, depending on the scale of the processing and I/O systems (Figure 8.30).

FiberChannel talks to the disks at over 50 MB/s sustained through a bridge that converts the FiberChannel data format to the SCSI format that the disks accept. I/O to any disk in the system usually takes a path through the local PCI bus and the FiberChannel switch; however, if this path fails for some reason, the operating system causes I/O transfers to go through the SCI ring to another quad and through its PCI bus and FiberChannel link to the disk. FiberChannel may also be used to connect multiple NUMA-Q systems in a loosely coupled fashion and to have multiple systems share disks. Finally, a management and diagnostic controller connects to a PCI bus on each quad; these controllers are linked with one another and to a system console through a private local area network like Ethernet for system maintenance and diagnosis.

8.6.5 Protocol Interactions with SMP Node

The earlier discussion of the SCI protocol ignored the multiprocessor nature of the quad node and the bus-based protocol within it. Now that we understand the hardware structure of the node and the IQ-Link, let us examine the interactions of the two protocols, the requirements that the interacting protocols place upon the quad

and IQ-Link, and some particular problems raised by the use of an off-the-shelf SMP as a node.

A read request illustrates some of the interactions. A read miss in a processor's second-level cache first appears on the quad bus. In addition to being snooped by the other processor caches, it is snooped by the OBIC bus controller on the IQ-Link board. The OBIC looks up the remote cache as well as the directory state bits for locally allocated blocks to see if the read can be satisfied within the quad or if it must be propagated off node. In the former case, main memory or one of the other caches satisfies the read, and the appropriate MESI state changes occur. (Snoop results are reported, in order, after a fixed number of bus cycles [four]; if a controller cannot finish its snoop within this time, it asserts a stall signal for another two bus cycles, after which memory checks for the snoop result again. This continues until all snoop results are available.) The quad bus implements in-order data responses to requests. However, if the OBIC detects that the request must be propagated off node, then it must intervene. It does this by asserting a *deferred response* signal, telling the bus to violate its in-order response property and proceed with other transactions and that the OBIC will take responsibility for responding to this request. This would not have been necessary if the quad bus implemented out-of-order responses. The OBIC then passes on the request to the SCLIC to engage the directory protocol. When the response comes back, it is passed from the SCLIC back to the OBIC, which places it on the bus and completes the deferred transaction. Note that when extending any bus-based system to be the node of a larger cache-coherent machine, it is essential that the bus be split transaction, not only for performance but also to simplify correctness. Otherwise, the bus will be held up for the entire duration of a remote transaction, not allowing even local misses to complete and not allowing incoming network transactions to be serviced by processor caches (potentially causing deadlock).

Writes take a similar path out of and back into a quad. The state of the block in the remote cache, snooped by the OBIC, indicates whether the block is owned by the local quad or a request must be propagated to the home through the SCLIC. Putting the node at the head of the sharing list and invalidating other nodes, if necessary, is taken care of by the SCLIC. When the SCLIC is done, it places a response on the quad bus (via the OBIC), which completes the operation. An interesting situation arises due to a limitation of the quad itself. Consider a read miss or write miss to a locally allocated block that is cached remotely in a modified state. When the response returns and is placed on the bus as a deferred response, it should update the main memory. However, the quad memory was not implemented to deal with deferred requests and responses and does not update itself on seeing a deferred response. Thus, when a deferred response is passed down to the bus through the OBIC, the OBIC must also ensure that it updates the memory through a special action before it gives up the bus. Another limitation arises from how the OBIC uses the quad bus protocol. If two processors in a quad issue read-exclusive requests back to back, and the first one propagates to the SCLIC, we would like the second one to be buffered and accept the response from the first in the appropriate state. However,

the implementation NACKs the second request, which will then have to retry until the first one returns and it succeeds.

Finally, consider serialization. Since serialization at the SCI protocol level is done at the home, incoming transactions at the home have to be serialized not only with respect to one another but also with respect to accesses by the processors in the home quad. For example, suppose a block is in the HOME state at the home. At the SCI protocol level, this means that no remote cache in the system (which must be on some other node) has a valid copy of the block. However, unlike the unowned state in the Origin protocol, this does not mean that no processor cache in the home node has a copy of the block. In fact, the directory will be in HOME state even if one of the processor caches at the home has a dirty copy of the block. Even to obtain the right value, a request coming in for a locally allocated block at a home node must therefore be broadcast on the quad bus as well and cannot be handled entirely by the SCLIC and OBIC. Similarly, an incoming request that makes the directory state change from HOME or FRESH to GONE must be put on the quad bus so that the copies in the processor caches can be invalidated. Since both incoming requests and local misses to data at the home appear on the quad bus, it is natural to let this bus be the actual serializing agent at the home.

Similarly, serialization issues need to be addressed in a requesting quad for accesses to remotely allocated blocks. Activities within a quad relating to remotely allocated blocks are serialized at the local SCLIC rather than the local bus. Thus, requests from local processors for a block in the remote cache and incoming requests from the SCI interconnect for the same block are serialized at the local SCLIC. Similarly, the SCLIC takes care of the local serialization between outstanding invalidations at a requestor and incoming requests. Other interactions with the node protocol are discussed once we have considered the implementation of the IQ-Link board components.

8.6.6 IQ-Link Implementation

Unlike the single-chip Hub in Origin, the SCLIC directory controller, the OBIC bus interface controller, and the DataPump are separate chips on the IQ-Link board, which also contains some SRAM and SDRAM chips for tags, state, and remote cache data (see Figure 8.29).

The data in the remote cache is directly accessible by the OBIC. Two sets of tags are used to reduce communication between the SCLIC and the OBIC: the network-side tags for access by the SCLIC and the bus-side tags for access by the OBIC. The same is true for the directory state for locally allocated blocks. The bus-side tags and directory state contain only the information that is needed for the bus snooping and are implemented in SRAM so they can be looked up at bus speed. The network-side tags and state need more information and can be slower, so they are implemented in synchronous DRAM (SDRAM). The bus-side local directory SRAM contains only the 2 bits of directory state per 64-byte block (to distinguish the HOME, FRESH, and GONE states) whereas the network-side directory contains the 6-bit SCI head pointer

as well. The bus-side remote cache tags also have only 4 bits of state and do not contain the SCI forward and backward list pointers. They keep track of 14 states, some of which are transient states that ensure forward progress within the quad (e.g., that keep track of blocks that are being rolled out or of the particular bus agent that has an outstanding retry on the bus and so must get priority for that block). The network-side remote cache tags, which are part of the directory protocol, contain 7 bits to represent all protocol states plus two 6-bit pointers per block (as well as the 13-bit cache tags themselves).

Unlike the hardwired protocol tables in Origin, the SCLIC coherence controller in NUMA-Q is programmable. This means the protocol can be written in software or firmware rather than hardwired into a finite state machine. Every protocol-invoking operation from a local processor, as well as every incoming transaction from the network, invokes a software “handler” or task that runs on the protocol engine. These software handlers, written in microcode, may manipulate directory state, put interventions on the quad bus, generate network transactions, and so on. The SCLIC engine has multiple register sets to support 12 read/write/invalidate transactions and 1 interrupt transaction concurrently. To allow the standard intraquad interrupt interface to be used across quads, the SCLIC provides a bridge for routing standard intraquad interrupts between quads and provides some extra bits to include the destination quad number when generating such interrupts.

A programmable protocol engine has several potential advantages. It allows the protocol to be debugged in software and corrected by simply downloading new protocol code. It provides the flexibility to experiment with or change protocols even after the machine is built and bottlenecks are discovered, and allows multiple protocols to be supported by the machine. And it enables code to be inserted into the handlers to monitor chosen events for performance debugging, which is especially valuable given the implicit nature of communication and the potential impact of artifactual communication in a shared address space. The disadvantage is that a programmable protocol engine has higher occupancy per transaction than a hardwired one, so a performance cost is associated with this decision. Attempts are made to reduce this performance impact in the NUMA-Q SCLIC. The protocol processor has a three-stage pipeline and issues up to two instructions (a branch and another instruction) every cycle. It uses a cache to hold recently used directory state and tag information rather than accessing the directory RAMs every time. Finally, it is specialized to support the kinds of bit-field manipulation operations that are commonly needed in directory protocols as well as useful instructions that speed up handler dispatch and management, like “queue on buffer full” and “branch on queue space available” instructions. A somewhat different programmable protocol engine is used in the Stanford FLASH multiprocessor (Kuskin et al. 1994), the successor to the hardwired Stanford DASH machine.

Each Pentium Pro processor can have up to four requests outstanding. The quad bus can have eight requests outstanding at a time and ensures that snoop and data responses come in order (except when deferred responses are used, as discussed earlier). The OBIC can have four external requests outstanding to the SCLIC and can buffer two incoming transactions to the quad bus at a time. If a fifth request from the

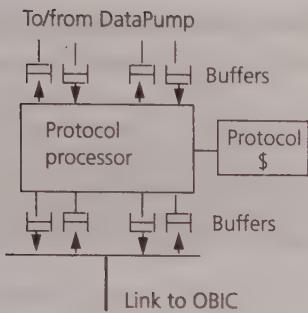


FIGURE 8.31 Simplified block diagram of the SCLIC chip. It contains a programmable protocol processor, a cache for directory information, and buffers to interface with the OBIC (bus) and DataPump (network).

quad bus needs to go off quad, the OBIC will NACK it until a buffer entry is free but will not cause the quad bus to stall for local operations. The SCLIC can have up to eight requests outstanding and can buffer four incoming requests at a time. A simplified illustration of the SCLIC is shown in Figure 8.31. Finally, the DataPump request and response buffers are each two entries deep outgoing to the network and four entries deep incoming. All request and response buffers, whether incoming or outgoing, are physically separate in this implementation.

In addition to the ability to instrument protocol handlers in software, all three components of the IQ-Link board also provide performance counters to enable non-intrusive measurement of various events and statistics. There are three 40-bit memory-mapped counters in the SCLIC and four in the OBIC. Each can be set in software to count any of a large number of events, such as protocol engine utilization, memory and bus utilization, queue occupancies, the occurrence of SCI command types, and the occurrence of transaction types on the quad bus. The counters can be read by software on the main processors at any time or can be programmed to generate interrupts when they cross a predefined threshold value. The Pentium Pro processor module itself provides a number of performance counters to count first- and second-level cache misses as well as the frequencies of request types and the occupancies of internal resources, among other properties. Together with the programmable handlers, these counters can provide a wealth of information about the behavior of the machine when running workloads.

8.6.7 Performance Characteristics

The quad bus has a peak bandwidth of 532 MB/s, and the SCI ring interconnect can transfer 500 MB/s in each direction across the node-to-network interface. The IQ-Link board can transfer data between these two interconnects at about 30 MB/s in each direction (note that only a small fraction of the transactions appearing on the quad bus or on the SCI ring are expected to be relevant to the other interconnect). The latency for a local read miss satisfied in main memory (or the remote cache) is expected to average about 250 ns under ideal conditions. The latency for a read satisfied in remote memory in a two-quad system is expected to be about 2.5 μ s, a ratio

Table 8.3 Characteristics of Microbenchmarks and Workloads Running on an Eight-Quad NUMA-Q

Workload	Latency of L ₂ Misses		SCLIC Utilization	Percentage of L ₂ Misses Satisfied in			
	All	Remotely Satisfied		Local Memory	Other Local Cache	Local "Remote Cache"	Remote Node
Remote Read Misses	8,020 ns	8,300 ns	95%	1.5%	0%	2%	96.5%
Remote Write Misses	9,350 ns	9,625 ns	95%	1%	0%	2%	97%
TPC-B-like	630 ns	4,300 ns	54%	80%	2%	11.5%	6.5%
TPC-D (Q9)	580 ns	3,950 ns	40%	85%	5.5%	4%	5.5%

of about 10 to 1. However, the inclusion of a remote access cache keeps the frequency of artifactual communication very low. The latency through the DataPump network interface for the first 18 bits of a transaction is 16 ns and then 2 ns for every 18 bits thereafter. In the network itself, it takes about 26 ns for the first bit to get from the DataPump output of a quad into the IQ-Plus box that implements the ring and back out to the DataPump of the next quad along the ring.

The designers of the NUMA-Q have performed several experiments on the machine with microbenchmarks and with database and transaction processing workloads. To obtain a flavor for the microbenchmark performance capabilities of the machine, how latencies vary under load, and the characteristics of such workloads, let us take a brief look at the results. For a single-quad system with all four processors simultaneously generating cache misses as quickly as they can, back-to-back read misses are found to take 600 ns each and obtain a combined transfer bandwidth to the processors of 290 MB/s. Under similar conditions, back-to-back write misses, which cause a read followed by a write back, take 585 ns, and sustain 195 MB/s. For a single-quad system with multiple I/O controllers on each PCI I/O bus generating inbound writes from the I/O devices to the local memory as quickly as possible, each cache block transfer takes 360 ns at 111 MB/s sustained bandwidth.

Table 8.3 shows the latencies and characteristics under load as seen in various workloads running on multiple-quad systems. The first two rows are for microbenchmarks designed to have all quads simultaneously issuing read misses that are satisfied in remote memory. The third row is for the Transaction Processing Council's on-line transaction processing benchmark TPC-B (see Appendix). The last row is for Query 9 of the TPC-D benchmark suite, which represents decision support applications. The latencies are measured using the performance counters embedded in the OBIC and SCLIC and are measured not from the processor but from the bus request to the first data response. All workloads are run with four quads (16 processors), except the decision support workload, which is run with eight. Write misses to locally allocated

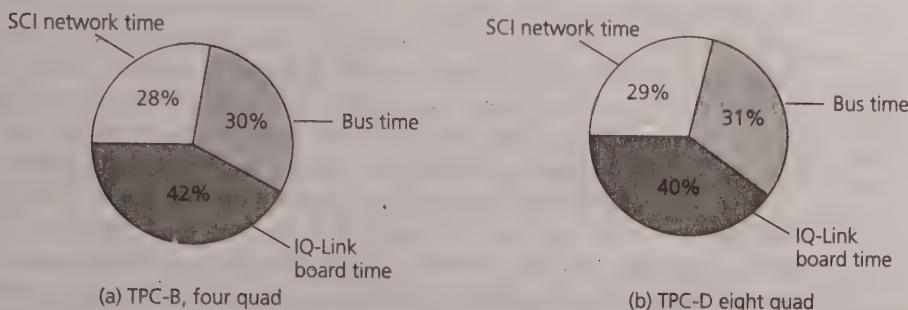


FIGURE 8.32 Components of average remote miss latency in two workloads on an eight-quad NUMA-Q. In both cases, most of the time is spent in the IQ-Link board, which includes data transfers between the SCLIC and the DataPump or the OBIC. Time in the OBIC chip itself is included in bus time in this figure.

data that cause invalidations to be sent remotely are very few and are included in the last column.

Remote data access latencies are clearly significantly higher than the unloaded latencies. In general, the SCI ring and protocol have higher latencies than those of more distributed networks and memory-based protocols, as discussed earlier. However, at least in these transaction processing and decision support workloads, much of the time in a remote access is spent passing through the IQ-Link board itself and not in the bus or ring. Figure 8.32 shows the breakdowns of average remote latency into three components for two workloads on four- and eight-quad systems. The path to improved remote access performance, both under load and not under load, is to make the IQ-Link board more efficient. The designers are considering a number of opportunities, including redesigning the SCLIC, perhaps using two instruction sequencers instead of one in the programmable SCLIC, and optimizing the OBIC, with the hope of reducing the remote access latency to about 2 μ s under heavy load in the next generation. The remote cache is found to be very useful in keeping capacity misses local. The TPC-D (Q9) workload has lower SCLIC utilization than the TPC-B workload because it generates fewer invalidations.

8.6.8

Comparison Case Study: The HAL S1 Multiprocessor

The S1 multiprocessor from HAL Computer Systems is an interesting combination of some features of the NUMA-Q and the Origin2000. Like the NUMA-Q, the S1 also uses Pentium Pro quads as the processing nodes; however, it uses a memory-based directory protocol like that of the Origin2000 across quads rather than the cache-based SCI protocol. In addition, to reduce latency and assist occupancy, it integrates the coherence machinery more tightly with the node than the NUMA-Q does, coming closer to the Origin in this regard. Instead of using separate chips for

the directory protocol controller (SCLIC), bus interface controller (OBIC), and network interface (DataPump), the S1 integrates the entire communication assist and the network interface into a single chip called the mesh coherence unit (MCU), with separate chips used for storage. On the other hand, the cache-coherent design scales to only four quads, does not have the flexibility of a programmable controller, and does not include a remote access cache to reduce remote capacity misses.

Since the memory-based protocol does not require the use of forward and backward pointers with each cache entry, there is no need for a quad-level remote data cache to provide this functionality (which processor caches do not provide); in memory-based protocols, remote caches are useful only to reduce capacity misses, and the S1 does not use them. The directory information is maintained in separate SRAM chips, but the directory storage needed is greatly reduced by maintaining directory information not for all memory blocks but only for those blocks that are in fact cached remotely, organizing the directory itself as a cache (as discussed in Section 8.10.1). The MCU also contains a DMA engine to support explicit message passing as well as block data transfers in a cache-coherent shared address space (see Chapter 11). Message passing or explicit data transfers can be implemented either through the DMA engine (preferred for large messages) or through the transfer mechanism used for cache blocks (preferred for small messages). The MCU is hard-wired instead of programmable, which reduces its occupancy for protocol processing and hence improves its performance under contention. The MCU also has substantial hardware support for performance monitoring. Other than the MCU, the only custom chip used is the network router, which is a six-ported crossbar with 1.9 million transistors, optimized for speed. The network is clocked at 200 MHz. The latency through a single router is 42 ns, and the usable per-link bandwidth is 1.6 GB/s in each direction—both similar to that of the Origin2000 network. The initial S1 interconnect implementation scales to 32 nodes (128 processors).

A major goal of integrating all the assist functionality into a single chip in S1 was to reduce remote access latency and increase remote bandwidth. From the designers' simulated measurements, the best-case unloaded latency for a read miss that is satisfied in local memory is 240 ns, for a read miss to a block that is clean at a nearby remote home is 1,065 ns, and for a read miss to a block that is dirty in a (nearby) third node is 1,365 ns. The remote-to-local latency ratio ranges from 4 to 5 (including contention), which is a little worse than on the SGI Origin2000 but better than on the NUMA-Q. However, microbenchmark comparisons of latencies are not very meaningful as predictors of overall performance on workloads since they ignore important considerations like remote caches and flexibility that can greatly affect the frequency of communication.

The bandwidths achieved by the HAL S1 in copying a single 4-KB page are instructive. The achieved bandwidth is 105 MB/s from local memory to local memory through processor reads and writes (limited primarily by the quad memory controller that has to handle both the reads and writes of memory), about 70 MB/s between local memory and a remote memory (in either direction) when accomplished through processor reads and writes, and about 270 MB/s in either direction between local and remote memory when performed through the DMA engines in the

MCUs. The case of remote transfers through processor reads and writes is limited primarily by the limit on the number of outstanding memory operations from a processor, which is not an issue for the DMA case. The DMA case has the additional advantage that it requires only one bus transaction at the initiating end for each memory block rather than two split-transaction pairs in the case of processor reads and writes (once for the read and once for the write). At least in the absence of contention across transfers, the local quad bus becomes a bandwidth bottleneck long before the interconnection network does.

Now that we understand the protocol layer that implements the coherent shared address space programming model in some depth for both memory-based and cache-based protocols, let us briefly examine some key interactions of protocols with the basic performance parameters of the communication architecture in determining the performance of applications.

8.7

PERFORMANCE PARAMETERS AND PROTOCOL PERFORMANCE

Recall that there are four major performance parameters in a communication architecture: overhead on the main processor, occupancy of the communication assist, network transit delay, and network bandwidth. Processor overhead is usually quite small on cache-coherent machines (unlike on message-passing systems, where it often dominates) and is determined entirely by the underlying node. In the best case, the portion that we can call processor overhead, and which cannot be hidden from the processor through overlap, is the cost of issuing the memory operation. In the worst case, it is the cost of traversing the processor's cache hierarchy and reaching the assist (which can be quite significant). All other protocol processing actions are off-loaded to the communication assist (e.g., the Hub or the IQ-Link). Network link bandwidth, too, is usually adequate for most applications in high-performance multiprocessor networks (Holt et al. 1995). The more critical issues under the control of the communication architecture are, therefore, network delay and assist occupancy.

As we have seen, the communication assist has many roles in protocol processing, including generating a request, looking up the directory state, accessing the data for a response, and sending out and receiving invalidations and acknowledgments. The occupancy of the assist for processing a transaction not only contributes to the uncontended latency of that transaction but can also cause contention at the assist and hence increase the cost of other transactions. This is especially true in cache-coherent machines because of the large number of small transactions—both data-carrying transactions and others like requests, invalidations, and acknowledgments—which implies that the occupancy is incurred very frequently and not amortized very well. The situation is better than in shared address space machines that are not cache coherent, where a transaction transfers only the referenced word rather than a whole cache block because replication and coherence must be managed by the programmer (see the discussion in Section 3.6), but the amortization is still small. In fact, assist occupancy very often dominates the data transfer bandwidth of the node-to-network interface as the key bottleneck to throughput at the

endpoints (Holt et al. 1995). It is therefore very important to keep assist occupancy small. At the protocol level, it is important both to ensure that the assist is not tied up by an outstanding transaction while other unrelated transactions are available for it to process and to reduce the amount of processing needed from the assist per transaction. For example, if the home forwards a request to a dirty node, the home assist should not be held up until the dirty node returns a response—which would dramatically increase its effective occupancy—but should go on to service the next transaction and deal with the response later when it comes. At the hardware design level, it is important to specialize the assist enough and integrate it tightly with the node's memory system so that its effective occupancy per transaction is low. The tighter the integration and the greater the specialization, the less commodity oriented the design but the lower the occupancy.

Impact of Network Delay and Assist Occupancy

Figure 8.33 shows the impact of assist occupancy and network latency on performance, assuming an efficient memory-based directory protocol similar to that of the SGI Origin2000. In the absence of contention, assist occupancy behaves just like network transit delay or any other component of the latency in a transaction's path: increasing occupancy by d cycles would have the same impact as keeping occupancy constant but increasing network delay by d cycles. Since the x -axis is total uncontended round-trip latency for a remote read miss (including the cost of network delay and assist occupancies incurred along the way), if no contention is induced by increasing occupancy, then all the curves for different values of occupancy will be identical. In fact, they are not, and the separation of the curves indicates the impact of the contention induced by increasing assist occupancy.

The smallest value of occupancy (o) in the graphs is intended to represent that of an aggressive hardwired assist that is tightly integrated with the cache or memory controller, such as the one used in the Origin2000. The least aggressive one represents placing a slow general-purpose processor on the memory bus to play the role of communication assist. The most aggressive network delays used represent modern high-end multiprocessor interconnects whereas the least aggressive ones are closer to using commodity system area networks like asynchronous transfer mode (ATM). We can see that for an aggressive occupancy, the latency curves take the expected $1/l$ shape. The contention induced by assist occupancy has a major impact on performance for applications that stress communication throughput (especially those in which communication is bursty), particularly for the low-delay networks used in multiprocessors. Thus the curves for higher occupancies are far apart from one another toward their left ends. For reasonable occupancies, the curves become closer to one another at larger network delays, since the greater time spent by transactions in the network keeps the assist less busy and hence keeps contention at the assist smaller. For higher occupancies, the curve almost flattens, at least with lower network delays, indicating that the assist is saturated. The problem is especially

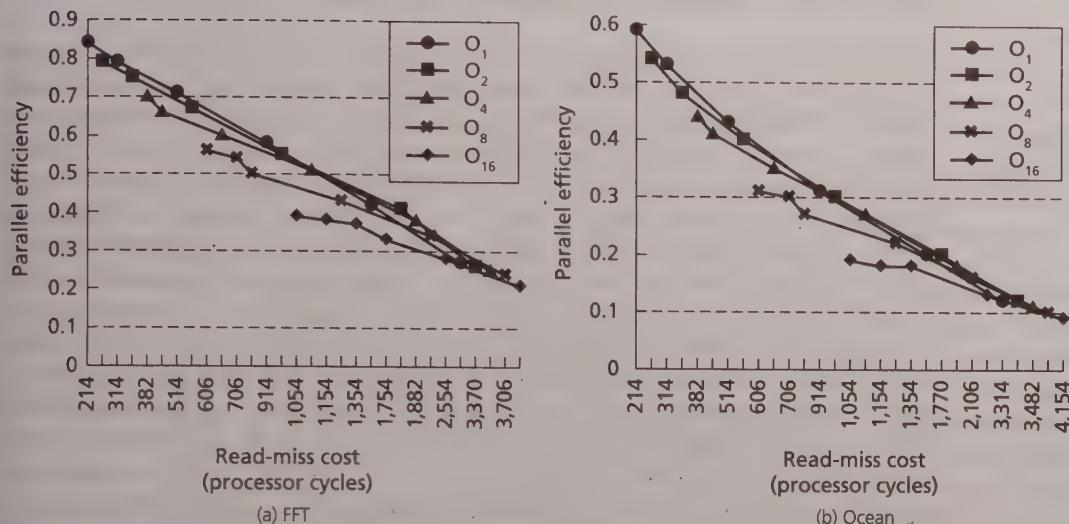


FIGURE 8.33 Impact of assist occupancy and network latency on the performance of memory-based cache coherence protocols. The y-axis is the parallel efficiency, which is the speedup over a sequential execution divided by the number of processors used (1 is ideal speedup). The x-axis is the uncontended round-trip latency of a read miss that is satisfied in main memory at the home, including all components of cost (occupancy, transit latency, time in buffers, and network bandwidth). Each curve is for a different value of assist occupancy (o), while along a curve the only parameter that varies is the network transit delay (l). The lowest occupancy assumed is 7 processor cycles, which is labeled O_1 . O_2 corresponds to twice that occupancy (14 processor cycles) and so on. All other costs, such as the time to propagate through the cache hierarchy and through buffers and the node-to-network bandwidth, are held constant. The graphs are for simulated 64-processor executions. The main conclusion is that the contention induced by assist occupancy is very important to performance, especially in low-latency networks.

severe for applications with bursty communication, such as sorting and FFTs, since there the rate of communication relative to computation during the communication phase does not change much with problem size, so larger problem sizes do not help alleviate the contention during that phase. Assist occupancy is a less severe problem for applications in which communication events are separated by significant computation and whose communication bandwidth demands are small (e.g., Barnes-Hut). When latency tolerance techniques are used (discussed in Chapter 11), bandwidth is stressed even further, so the impact of assist occupancy is much greater even at higher transit latencies, and the curves at the highest occupancies are almost completely flat for FFT and sorting (Holt et al. 1995). This data shows that it is very important to keep assist occupancy low in machines that communicate and maintain coherence at a fine granularity such as that of cache blocks. The impact of contention due to assist occupancy tends to increase with the number of processors used to solve a given problem since the communication-to-computation ratio tends to increase.

Effects of Assist Occupancy on Protocol Trade-Offs

The occupancy of the assist has an impact not only on the performance of a given protocol but also on the trade-offs among protocols. We have seen that cache-based protocols can have higher latency on write operations than memory-based protocols since the transactions needed to invalidate sharers are serialized. The SCI cache-based protocol also tends to have more protocol processing to do on a given memory operation than a memory-based protocol, so the effective occupancy of the assist tends to be significantly higher, especially when assists are programmable rather than hardwired. Combined with the higher latency on writes, this would tend to cause memory-based protocols to perform better. This difference between the performance of the protocols will become greater as assist occupancy and its performance impact increase. On the other hand, the protocol processing occupancy for a given memory operation (e.g., a write) in SCI is distributed over more nodes and assists, so, depending on the communication patterns of the application, it may experience less contention at a given assist. For example, when hot spotting becomes a problem due to bursty irregular communication in memory-based protocols (as in radix sorting), it may be somewhat alleviated in SCI. How these trade-offs play out in practice will depend on the characteristics of real programs and machines, although overall we might expect memory-based protocols to perform better in optimized implementations.

Improving Performance Parameters in Hardware

There are many ways to use more aggressive, specialized hardware to improve performance characteristics such as delay, occupancy, and bandwidth. Some notable techniques include the following. First, an SRAM directory cache may be placed close to the assist to reduce directory lookup cost, as is done in NUMA-Q and in the Stanford FLASH multiprocessor (Kuskin et al. 1994). Second, a single bit of SRAM can be maintained per memory block at the home to keep track of whether or not the block is in clean state in the local memory. If it is, then on a read miss to a locally allocated block, there is no need to invoke the communication assist any further. Third, if the assist occupancy is high, it can be pipelined into stages of protocol processing, as is also done in the NUMA-Q and Stanford FLASH (e.g., decoding a request, looking up the directory, generating a response), or its occupancy can be overlapped with other actions. Pipelining the assist reduces contention but not the uncontended latency of individual memory operations; the opposite (and complementary) result can be achieved by having the assist generate and send out a response or a forwarded request even before all the cleanup it needs to do is done.

8.8 SYNCHRONIZATION

Software algorithms for synchronization on scalable non-cache-coherent shared address space systems using atomic exchange instructions or LL-SC are discussed in Section 7.9. Recall that the major focus of these algorithms compared to those for

bus-based machines is to exploit the parallelism of independent paths in the interconnect and to ensure that processors will spin on local rather than nonlocal variables. The same algorithms are applicable to scalable cache-coherent machines. However, there are two differences. First, the performance implications of spinning on remotely allocated variables are likely to be much less significant since a processor caches the variable and then spins on it locally until it is invalidated. Having processors spin on different variables rather than the same one is of course useful in preventing all processors from rushing out to the same home memory when the variable is written and invalidated, thereby reducing contention. And good placement of synchronization variables has the benefit of converting the misses that occur after invalidation into two-hop misses from three-hop misses. However, there is only one (very unlikely) situation when it may actually be very important to performance that the variable a processor spins on be allocated locally: if all levels of the cache hierarchy are unified and direct mapped and the instructions for the spin loop conflict with the variable itself, in which case conflict misses will be satisfied locally. Second, while these performance aspects of synchronization algorithms are less critical, implementing atomic primitives and LL-SC is more interesting when it interacts with a coherence protocol. This section examines the performance and implementation aspects, first comparing the performance of the different synchronization algorithms for the locks described in Chapters 5 and 7 on the SGI Origin2000 and then discussing some new implementation issues for atomic primitives beyond the issues already encountered in Chapter 6 for bus-based machines.

8.8.1

Performance of Synchronization Algorithms

The experiments used here to illustrate synchronization performance are the same as those used on the bus-based SGI Challenge in Section 5.5, again using LL-SC as the primitive to construct atomic operations. The delays used are the same in processor cycles and therefore different in actual microseconds. The results for the lock algorithms described in Chapters 5 and 7 are shown in Figure 8.34 for 16-processor executions. Here again, three different sets of values are used for the delays within and after the critical section for which processors repeatedly contend.

Here too, until we use delays between critical sections, the simple locks behave unfairly and yield higher throughput. Exponential backoff often helps the simple LL-SC lock in the event of a null critical section since this is the case where significant contention needs to be alleviated. The ticket lock scales quite poorly in this case, as it did on a bus, but scales very well when proportional backoff is used. The array-based lock also scales very well. With coherent caches, the better placement of lock variables in main memory afforded by the software queuing lock is not particularly useful, and in fact the queuing lock incurs contention on its compare&swap operations (implemented with LL-SC) and scales worse than the array lock. If we force the simple locks to behave fairly, they behave much like the ticket lock without proportional backoff.

If we use a non-null critical section and a delay between lock accesses (Figure 8.34[c]), all locks behave fairly. Now the simple LL-SC locks don't have

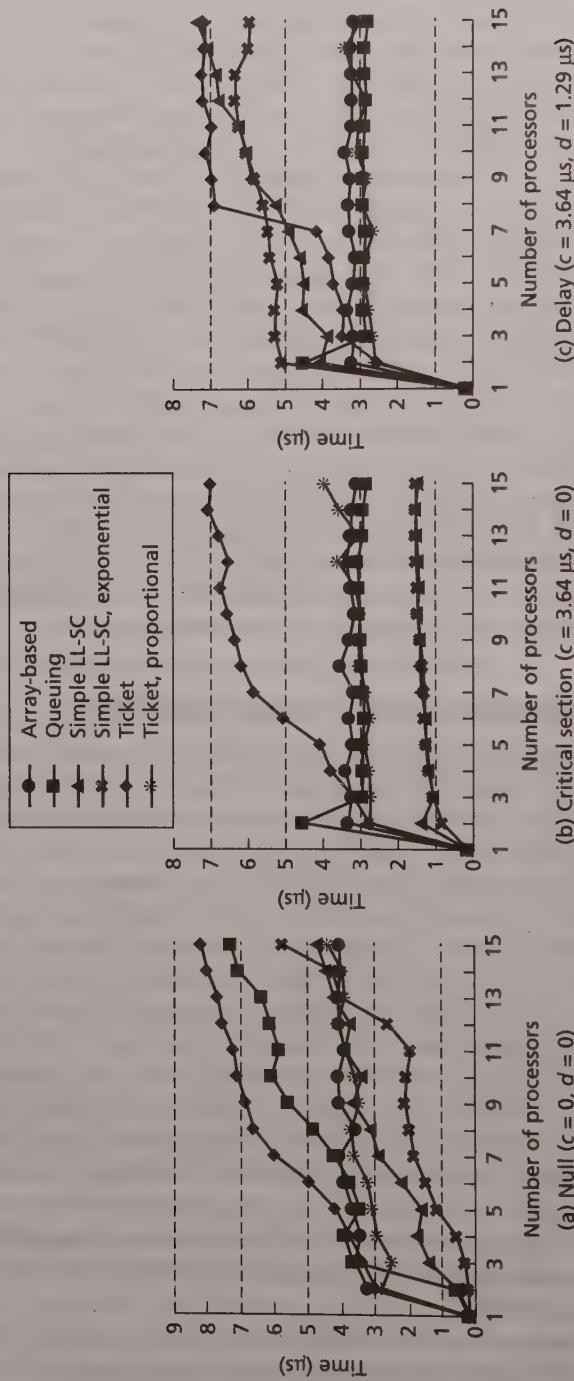


FIGURE 8.34 Performance of locks on the SGI Origin2000 for three different scenarios

their advantage, and their scaling disadvantage shows through. The array-based lock, the queuing lock, and the ticket lock with proportional backoff all scale well (at least to this small number of processors). The better data placement of the queuing lock does not matter, but neither is the contention any worse for it. The bad performance of the queuing lock at two processors is due to a specific interaction in constructing the software queue (Mellor-Crummey and Scott 1991). While experiments with larger-scale machines are warranted, the flattening of the curves indicates that, overall, the array-based lock and the ticket lock perform quite well and robustly for scalable cache-coherent machines, at least when implemented with LL-SC. The simple LL-SC lock with exponential backoff performs best when no delay occurs between an unlock and the next lock due to repeated unfair successful access by a processor in its own cache. The sophisticated queuing lock is unnecessary but also performs well with delays between unlock and lock.

More aggressive hardware support for locks has been proposed. The most prominent example is a hardware version of the queuing lock called QOLB (queue on lock bit). A distributed linked list of nodes waiting on a lock is maintained in hardware, and a releaser grants the lock to the first waiting node without affecting the others (Kägi, Burger, and Goodman 1997). Since the SCI protocol already has hardware support for a distributed list of waiting nodes (namely, the pending list), QOLB locks fit very well with SCI. This aggressive hardware support may reduce the lock transfer time as well as the interference of lock traffic with data access and coherence traffic; however, it is unlikely to change the scaling trends of the lock microbenchmarks, and, as with all system features, its true value to performance is best evaluated with real applications and workloads.

Algorithms and hardware support for barriers are discussed in Section 7.9. Since barriers reached simultaneously by multiple nodes cause contention for read-modify-write access to a shared counter, a number of interesting questions arise: Should this counter variable be a cacheable location or an uncached location accessed at main memory? Or can mechanisms be developed to allow processors to spin in their caches and either be updated at the release or read the release value from main memory rather than from the releaser's cache? Or is the hardware support for at-memory fetch&op operations particularly valuable as provided by machines like the Origin2000?

8.8.2

Implementing Atomic Primitives

Consider implementing atomic exchange (read-modify-write) primitives like test&set performed on a memory location. What matters for atomicity is that a conflicting write to that location by another processor occur either before the read component of the read-modify-write operation or after its write component. As we discussed for bus-based machines in Section 5.5.3, the read component may be allowed to complete as soon as the write component is serialized with respect to other writes and as long as we ensure that no incoming invalidations are applied to the block until the read has completed. If the read-modify-write is implemented at the processor (using cacheable primitives), this means that the read can complete

once the write has obtained ownership and even before invalidation acknowledgments have returned. Atomic operations can also be implemented at the memory, but it is easier to do this if we disallow the block from being cached in dirty state by any processor. Then all writes go to memory, and the read-modify-write can be serialized with respect to other writes as soon as it gets to memory. Memory can send a response to the read component in parallel with sending out invalidations corresponding to the write component.

Implementing LL-SC requires all the same consideration to avoid livelock as it did for bus-based machines, with one further complication. Recall that a store-conditional should not send out invalidations or updates if it fails since, otherwise, two processors may keep invalidating or updating each other and failing, causing livelock. To detect failure of a store-conditional, the requesting processor needs to determine if some other processor's write to the block has been serialized before the store-conditional. In a bus-based system, the cache controller can do this by checking upon a store-conditional whether the cache no longer has a valid copy of the block or whether there are incoming invalidations or updates for the block that have already appeared on the bus. The latter detection of serialization order cannot be done locally by the cache controller with a distributed interconnect, so a different mechanism is necessary. In an invalidation-based protocol, if the block is still in valid state in the cache, then the read-exclusive request corresponding to the store-conditional goes to the directory at the home. There it checks to see if the requestor is still on the sharing list. If it isn't, then the directory knows that another conflicting write has been serialized before the store-conditional, so it does not send out invalidations corresponding to the store-conditional and the store-conditional fails. Otherwise, it succeeds. In an update protocol, this is more difficult since, even if another write has been serialized before the store-conditional, the store-conditional requestor will still be on the sharing list. One solution (Gharachorloo 1995) is to again use a two-phase protocol as was used to provide write atomicity for updates. When the store-conditional reaches the directory, it locks down the entry for that block so that no other requests can access it. Then, the directory sends a message back to the store-conditional requestor, which upon receipt checks to see if the lock flag for the LL-SC has been cleared (by an update that arrived between the current time and the time the store-conditional request was sent out). If so, the store-conditional has failed and a message is sent back to the directory to this effect (and to unlock the directory entry). If not, then as long as point-to-point order is guaranteed in the network, we can conclude that no conflicting write beat the store-conditional to the directory, so the store-conditional should succeed. The requestor sends an acknowledgment back to the directory, which unlocks the directory entry and sends out the updates corresponding to the store-conditional, and the store-conditional succeeds.

8.9

IMPLICATIONS FOR PARALLEL SOFTWARE

Let us now consider the implications for parallel software more generally than for synchronization. What distinguishes the coherent shared address space systems

described in this chapter from those described in Chapters 5 and 6 is that they have physically distributed rather than centralized main memory. Distributed memory is at once an opportunity to improve performance and scalability through data locality and a burden on software to exploit this locality. As we saw in Chapter 3, on cache-coherent architectures with physically distributed memory (or CC-NUMA machines), such as those discussed in this chapter, parallel programs may need to be aware of physically distributed memory, particularly when their important working sets don't fit in the cache. Artifactual communication occurs when data is not allocated in the memory of a node that incurs capacity, conflict, or cold misses on that data. This situation can lead to some artifactual communication even when data does fit in the cache since looking up the directory on write misses (including upgrades) will generate network traffic and contention. Finally, consider a multiprogrammed workload in which application processes are migrated among processing nodes for load balancing. Migrating a process will turn what should be local misses into remote misses unless the system moves all the migrated process's data to the new node's main memory as well. For all these reasons, it may be important that data be allocated appropriately across the distributed memories.

In the CC-NUMA machines discussed in this chapter, the management of main memory is typically done at the fairly large granularity of pages. The large granularity can make it difficult to distribute shared data structures appropriately since data that should be allocated on two different nodes may fall on the same unit of allocation. The operating system may transparently migrate pages to the nodes that incur cache misses on them most often, using information obtained from hardware counters; or the run-time system of a programming language may migrate pages based on user-supplied hints or compiler analysis. (We saw that the Origin2000 provides protocol support for efficient migration.) More commonly today, the programmer may direct the operating system to place pages in the memories closest to particular processes. This may be as simple as providing these directives to the system—such as, “Place the pages in this range of virtual addresses in this process X's local memory”—or it may additionally involve padding and aligning data structures to page boundaries so they can be placed properly, or it may even require that data structures be organized differently to allow such placement at page granularity. We saw examples of the need for all three in using four-dimensional instead of two-dimensional arrays in the equation solver kernel and in Ocean. Simple, regular cases like these may also be handled by sophisticated compilers. In Barnes-Hut, on the other hand, proper placement would require a significant reorganization of data structures as well as code. Instead of having a single linear array for all particles (or cells), each process would have an array or list of its own assigned particles that it could allocate in its local memory; between time-steps, particles that were reassigned would be moved from one array or list to another. However, as we have seen, data placement is not very useful for this application due to the small working sets and low capacity miss rate and may even hurt performance due to its high costs. It is important that we understand the costs and potential benefits of data migration before using it. Similar issues hold for software-controlled replication of data instead

of migration, and the next chapter discusses alternative approaches to coherent replication and migration in main memory.

One of the most difficult problems for a programmer to deal with in a coherent shared address space is contention. Contention can be caused not only by data traffic that is implicit and often unpredictable but also by “invisible” protocol transactions, such as ownership requests, invalidations, and acknowledgments that a programmer is not inclined to think about at all and that are now point-to-point rather than amortized by a broadcast medium. All of these types of transactions occupy the protocol processing portion of the communication assist, reinforcing the importance of keeping the occupancy of the assist per transaction very low to contain endpoint contention. Invisible protocol messages and contention make performance problems like false sharing all the more important for a programmer to avoid, particularly when they cause a lot of protocol transactions to be directed toward the same node. Thus, while the software techniques for inherent communication and for spatial locality and false sharing at cache block granularity are the same as on bus-based machines, the potential impact on performance is different. For example, we are often tempted to structure some kinds of data as an array with one entry per process. If the entries are smaller than a page, several of them will fall on the same page. If these array entries are not padded to avoid false sharing or if they incur conflict misses in the cache, all the misses and traffic will be directed at the home of that page, causing considerable contention. In a distributed-memory machine it is advantageous not only to structure such data as an array of records rather than multiple arrays of scalars (as we do in Chapter 5 to avoid false sharing) but also to pad and align the records to a page and place the pages in the appropriate local memories.

An interesting example of how contention can cause different orchestration strategies to be used in message-passing and shared address space systems is illustrated by a high-performance parallel FFT. Conceptually, the computation is structured in phases. Phases of local computation are separated by phases of communication, which involve the transposition of a matrix. A process reads columns from a source matrix and writes them into its assigned rows of a destination matrix and then performs local computation on its assigned rows of the destination matrix. In a message-passing system, it is important to coalesce data into large messages, so it is necessary for performance to structure the communication this way (as a phase separate from computation). However, in a cache-coherent shared address space there are two differences. First, transfers are always done at cache block granularity. Second, each fine-grained transfer involves invalidations and acknowledgments (each local block that a process writes is likely to be in shared state in the cache of another processor from a previous phase and so must be invalidated), which cause contention at the coherence controllers. It may therefore be preferable to perform the communication on demand at fine grain while the computation is in progress, rather than all at once in a separate transpose phase, thus staggering the communication and easing the contention on the controller: a process that otherwise computes using a row of the destination matrix after the transpose can read the words of the corresponding source matrix column from a remote node on demand while it is

computing, performing the transpose in the process. Which method is better may depend on the architecture.

Finally, synchronization can be expensive in scalable systems, so programs should make a special effort to reduce the frequency of high-contention locks or global barrier synchronization.

8.10 ADVANCED TOPICS

Before concluding the chapter, we cover two additional topics. The first deals with the actual techniques used to reduce directory storage overhead in flat, memory-based schemes. The second addresses techniques for hierarchical coherence, both snooping and directory based.

8.10.1 Reducing Directory Storage Overhead

The discussion of flat, memory-based directories in Section 8.2.3 stated that the size or width of a directory entry can be reduced by using a limited number of pointers rather than a full bit vector and that doing so requires some overflow mechanism when the number of copies of the block exceeds the number of available pointers. Based on the empirical data about sharing patterns, the number of hardware pointers likely to be provided in limited pointer directories is very small, so it is important that the overflow mechanism be efficient. This section first discusses some possible overflow methods. It then examines techniques to reduce the number of directory entries, or directory “height,” by organizing the directory as a cache rather than having an entry for every memory block in the system. The limited pointer schemes with i pointers are named Dir_i followed by an abbreviation of their overflow methods, which include broadcast, no broadcast, coarse vector, software overflow, and dynamic pointers.

Overflow Methods for Reduced Directory Width

The overflow strategy in the *broadcast* or Dir_iB scheme (Agarwal et al. 1988) is to set a broadcast bit in the directory entry when the number of available pointers i is exceeded. When that block is written again, invalidation messages are sent to all nodes in the system, regardless of whether or not they were caching the block. It is not semantically incorrect to send an invalidation message to a processor not caching the block; however, network bandwidth may be wasted and latency stalls may be increased if the processor performing the write must wait for acknowledgments before proceeding. The advantage of the method is its simplicity.

The *no broadcast* or Dir_iNB scheme (Agarwal et al. 1988) avoids broadcast by never allowing the number of valid copies of a block to exceed i . Whenever the number of sharers is i and another node requests a shared copy of the block, the protocol invalidates the copy in one of the existing sharers and frees up that pointer in the directory entry for the new requestor. A major drawback of this scheme is that it

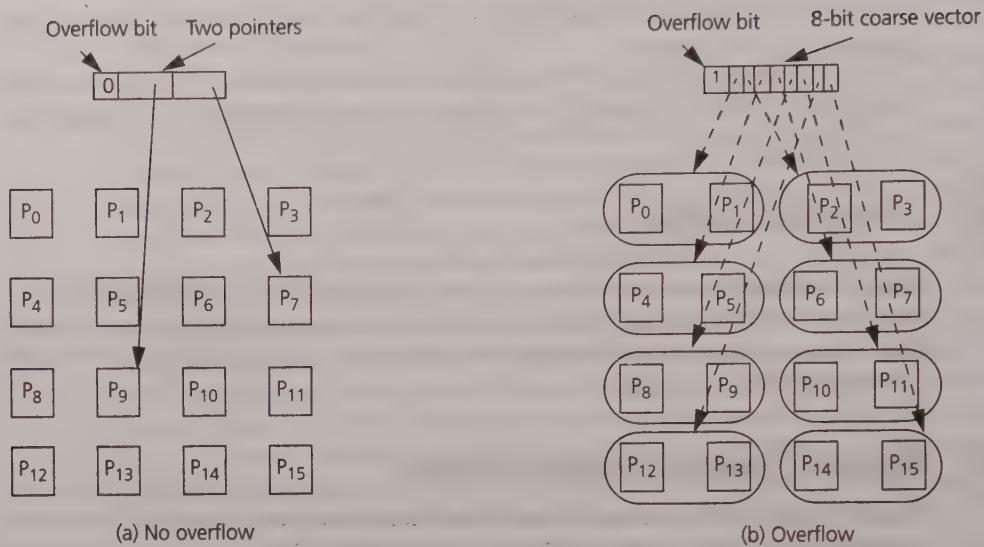


FIGURE 8.35 The change in representation in going from limited pointer representation to coarse vector representation on overflow. Upon overflow, the two 4-bit pointers (for a 16-node system) are viewed as an 8-bit coarse vector, each bit corresponding to a group of two nodes. The overflow bit is also set, so the nature of the representation can be easily determined. The dotted lines in (b) indicate the correspondence between bits and node groups.

does not deal well with data that is actively read by many processors during a period (e.g., tables of precomputed values or even program code), since copies will unnecessarily be invalidated and a continual stream of misses generated. Although special provisions can be made for blocks containing code (e.g., their consistency may be managed by software instead of hardware), it is not clear how to handle widely shared read-mostly data well in this scheme.

The *coarse vector* or Dir_iCV_r scheme (Gupta, Weber, and Mowry 1990) also uses i pointers in its initial representation, but on overflow the representation changes to a coarse bit vector like the one used by the Origin2000 for large machines. In this representation, each bit of the directory entry indicates not a node but a unique group of the nodes in the machine (the subscript r in Dir_iCV_r indicates the size of the group), and that bit is turned ON whenever any node in that partition is caching that block (see Figure 8.35). When a processor writes that block, all nodes in the groups whose bits are turned ON are sent an invalidation message, regardless of whether they have actually accessed or are caching the block. As an example, consider a 256-node machine for which we store eight pointers in the directory entry. Since each pointer needs to be 8 bits wide, 64 bits are available for the coarse vector on overflow. Thus, we can implement a Dir_8CV_4 scheme, with each coarse vector bit pointing to a group of 256/64 or four nodes. An additional single bit per entry keeps track of whether the current representation is that of the normal limited pointer or the coarse vector. As shown in Figure 8.36, an advantage of a scheme like Dir_iCV_r (and,

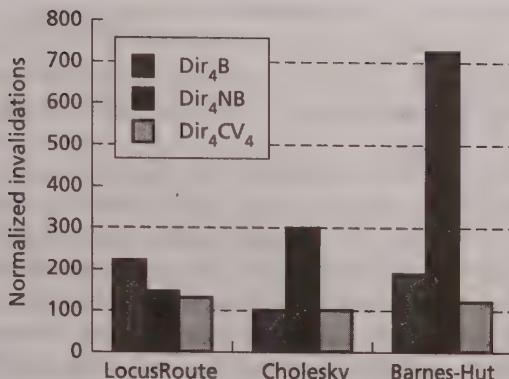


FIGURE 8.36 Robustness of the coarse vector overflow method relative to broadcast and no broadcast. The figure shows a comparison of invalidation traffic generated by Dir₄B, Dir₄NB, and Dir₄CV₄ schemes normalized to that generated by the full bit vector scheme (represented as 100 invalidations). The results are taken from (Weber 1993), so the simulation parameters are different from those used in this book. The number of processors (1 per node) is 64. The data for the LocusRoute wire-routing application, which has data that is written quite frequently and read by many nodes, shows the potential pitfalls of the Dir_iB scheme. Cholesky and Barnes-Hut, which have data that is read shared by large numbers of processors (e.g., nodes close to the root of the tree in Barnes-Hut) show the potential pitfalls of the Dir_iNB scheme. The Dir_iCV_i scheme is found to be reasonably robust.

even more so, of the following schemes) over Dir_iB and Dir_iNB is that its behavior is more robust to different sharing patterns.

The *software overflow* or *Dir_iSW* scheme is different from the previous ones in that it does not throw away the precise caching status of a block when overflow occurs. Rather, the current *i* pointers and a pointer to the new sharer are saved into a special portion of the node's local main memory by software. This frees up space for new pointers, so *i* new sharers can be handled by hardware before software must be invoked to store pointers away into memory again. The overflow also causes an overflow bit to be set in hardware. This bit ensures that when a subsequent write is encountered the pointers that were stored away in memory will be read out, and invalidation messages will be sent to those nodes as well. In the absence of a very sophisticated (programmable) communication assist, the overflow situations (both when pointers must be stored into memory and when they must be read out and invalidations sent) are handled by software running on the main processor, so the processor must be interrupted or a trap generated upon these events. The advantages of this scheme are that precise information is kept about sharers even upon overflow, so there is no extra invalidation traffic generated compared to a full bit vector (or unlimited pointer) representation, and that the complexity of overflow handling is managed by software. The major overhead is the cost of the interrupts and software processing. This disadvantage takes three forms: (1) the processor at the home of the block spends time handling the interrupt instead of performing the

user's computation; (2) the overhead of interrupts and of handling these requests is large, thus potentially becoming a bottleneck for contention and slowing down other requests; and (3) the requesting processor may stall longer because of the higher latency of the requests that can cause interrupts as well as increased contention.⁵

Software overflow for limited pointer directories was used in the MIT Alewife research prototype (Agarwal et al. 1995) and was called the LimitLESS scheme (Agarwal et al. 1991). The Alewife machine is designed to scale to 512 processors with one processor per node. Each directory entry is 64 bits wide. It contains five 9-bit pointers to record remote nodes caching the block and 1 dedicated bit to indicate whether the local node is also caching the block (thus saving 8 bits when this is true). Overflow pointers are stored in a hash table in the main memory. The main processor in Alewife has hardware support for multithreading (see Chapter 11), with support for fast handling of traps upon overflow. Nonetheless, although the latency of a request that causes five invalidations and can be handled in hardware is only 84 cycles on a 16-processor system, a request requiring six invalidations and, hence, software intervention takes 707 cycles.

The *dynamic pointers* or Dir_iDP scheme (Simoni and Horowitz 1991) is a variation of the Dir_iSW scheme. In addition to the i hardware pointers, each directory entry in this scheme contains a hardware pointer into a special portion of the local node's main memory. This special memory has a free list associated with it, from which pointer structures can be dynamically allocated to processors as needed. The key difference from Dir_iSW is that all linked-list manipulation is done in hardware by a special-purpose protocol processor rather than by the general-purpose processor of the local node. As a result, interrupts are not needed and the overhead of manipulating the linked lists is small. Because it also contains a hardware pointer to memory, the number of hardware pointers i used in this scheme is typically very small. The Dir_iDP scheme is the default directory organization for the Stanford FLASH multiprocessor (Kuskin et al. 1994). Because the pool of dynamic pointers is limited and because lists are traversed on invalidations, the use of replacement hints usually accompanies this approach.

Among these many alternative schemes for maintaining directory information in a memory-based protocol, it is quite clear that the Dir_iB and Dir_iNB schemes are not very robust to different sharing patterns. However, the actual performance (and cost-performance) trade-offs among the schemes are not very well understood for real applications on large-scale machines. The general consensus seems to be that full bit vectors are appropriate for machines that have a moderate number of processing nodes that are visible to the directory protocol. The most likely candidates for hardware overflow schemes are coarse vector and dynamic pointer: the former may suffer from lack of accuracy on overflow, while the latter has greater processing cost due to hardware list manipulation and free list management.

5. It is actually possible to respond to a requestor before the trap is handled and thus not affect the latency seen by it. However, that simply means that the next processor's request to that node is delayed and that processor may experience a stall.

Reducing Directory Height

In addition to reducing directory entry width, an orthogonal way to reduce directory memory overhead is to reduce the total number of directory entries used by not using one per memory block (Gupta, Weber, and Mowry 1990; O’Kafka and Newton 1990); that is, to go after the M term in the $P*M$ expression for directory memory overhead. Since the two methods of reducing overhead are orthogonal, they can be traded off against each other: reducing the number of entries allows us to make entries wider (use more hardware pointers) without increasing cost and vice versa.

The observation that motivates the use of fewer directory entries is that the total amount of cache memory is much less than the total main memory in the machine. This means that only a very small fraction of the memory blocks will be cached at a given time. For example, each processing node may have a 1-MB cache and 64 MB of main memory associated with it. If there were one directory entry per memory block, then across the whole machine 63/64 or 98.5% of the directory entries will correspond to memory blocks that are not cached anywhere in the machine. That is a tremendous number of directory entries lying idle with no bits turned ON (especially when replacement hints are used). This waste of memory can be avoided by organizing the directory as a cache and dynamically allocating the entries in it to directory entries, just as cache lines are allocated to memory blocks containing program data. In fact, if the number of entries in this directory cache is small enough, it may enable us to use fast SRAMs instead of slower DRAMs for directories, thus reducing the access time to directory information. As we know, this access time is in the critical path that determines the latency seen by the processor for many types of memory references. Such a directory organization is called a *sparse directory*, for obvious reasons. (The HAL S1 system, described in Section 8.6.8, uses this approach.)

While a sparse directory operates quite like a regular processor cache, there are some significant differences. First, this cache has no need for a backing store: when an entry is replaced from it, if any node’s bits (or pointers) in it are turned on then we can simply send invalidations or flush messages to those nodes. Second, there is only one directory entry per block in this cache, so spatial locality is not an issue. Third, a sparse directory handles references from potentially all processors, whereas a processor cache is only accessed by the processor(s) attached to it. And finally, the references stream that the sparse directory sees is heavily filtered, consisting of only those references that were not satisfied in the processor caches. For a sparse directory not to become a bottleneck, it is essential that it be large enough and have enough associativity that it does not incur too many replacements of actively accessed blocks. Some experiments and analysis studying the sizing of the sparse directory can be found in (Weber 1993).

8.10.2 Hierarchical Coherence

The introduction to this chapter mentions that one way to build scalable coherent machines is to hierarchically extend the snoopy coherence protocols based on the

buses and rings that are discussed in Chapters 5 and 6. We have also been introduced to hierarchical directory schemes in this chapter. This section describes these hierarchical approaches to coherence further. Although hierarchical ring-based snooping has been used in commercial systems (e.g., in the Kendall Square Research KSR1 [Frank, Burkhardt, and Rothnie 1993]) as well as research prototypes (e.g., in the University of Toronto's Hector system [Vranesic et al. 1991; Farkas, Vranesic, and Stumm 1992]), and hierarchical directories have been studied in academic research, these approaches have not gained much favor. Nonetheless, building large systems hierarchically out of smaller ones is an attractive abstraction, and it is useful to understand the basic techniques.

Hierarchical Snooping

The issues in hierarchical snooping are similar for buses and rings, so we study them mainly through the former. A bus hierarchy is a tree of buses. The leaves are bus-based multiprocessors that contain the processors. The buses that constitute the internal nodes of the tree don't contain processors but are used for interconnection and coherence control: they allow transactions to be snooped and propagated up and down the hierarchy as necessary. Hierarchical machines can be built with main memory either centralized at the root or distributed among the leaf multiprocessors (see Figure 8.37). While a centralized main memory may simplify programming, distributed memory has advantages in bandwidth and performance if locality is exploited. (Note, however, that if data is not distributed such that most cache misses are satisfied locally, remote data may actually be further away than the root of the hierarchy in the worst case, potentially leading to worse performance.) In addition, with distributed memory, a leaf in the hierarchy is a complete bus-based multiprocessor, which is already a commodity product with cost advantages. Let us focus on hierarchies with distributed memory, leaving centralized memory hierarchies to be explored in the exercises.

The processor caches within a leaf node (multiprocessor) are kept coherent by any of the snooping protocols discussed in Chapter 5. In a simple, two-level hierarchy, we connect several of these bus-based systems together using another bus (B_2). (The extension to multilevel hierarchies is straightforward.) What we need is a coherence monitor associated with each B_1 bus that monitors (snoops) the transactions on both buses and decides which transactions on its B_1 bus should be forwarded to the B_2 bus and which ones that appear on the B_2 bus should be forwarded to its B_1 bus. This device acts as a filter, forwarding only the necessary transactions in both directions, and thus reduces the bandwidth demands on the buses.

In a system with distributed memory, the coherence monitor for a node has to worry about two types of data for which transactions may appear on either the B_1 or B_2 bus: data that is allocated remotely but cached by some processor in the local node and data that is allocated locally but cached remotely. To watch for the former data, a *remote access cache* or *remote cache* per node can be used as in the Sequent NUMA-Q. This cache maintains inclusion (see Section 6.3.1) with regard to remote data cached in any of the processor caches on that node, including a dirty-but-stale

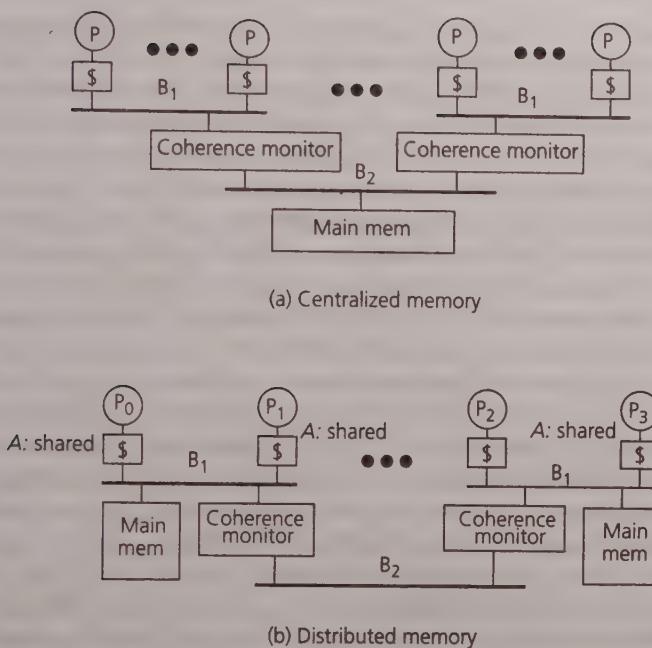


FIGURE 8.37 Hierarchical bus-based multiprocessors, shown with a two-level hierarchy. Main memory may be centralized at the root or physically distributed, and coherence monitors connect parent and child buses.

bit per block indicating when a processor cache in the node has the block dirty (data allocated in local memory does not enter the remote cache). This gives it enough information to determine which transactions are relevant in each direction and pass them along.

For locally allocated data, bus transactions can be handled entirely by the local memory or caches, except when the data is cached by processors in other (remote) nodes. For the latter data, there is no need to keep the data itself in the coherence monitor since the valid data is either already available locally or is in modified state remotely; in fact, we would not want to keep it there since the amount of data may be as large as the local memory. However, the monitor keeps state information for this data and snoops the local B_1 bus so that relevant transactions for this data can be forwarded to the B_2 bus if necessary. Let's call this part of the coherence monitor the *local state monitor*. Finally, the coherence monitor also watches the B_2 bus for transactions to its local addresses and passes them onto the local B_1 bus unless the local state monitor says they are cached remotely in a modified state. Both the remote cache and the local state monitor are looked up on B_1 and B_2 bus transactions.

Consider the three coherence protocol functions outlined in Section 8.1: (1) enough information about the state in other nodes of the hierarchy is implicitly available in the local node's coherence monitor (remote cache and local state monitor) to determine what action to take; (2) if this information indicates a need to find

other copies beyond the local node, the request or search is broadcast on the next bus (and so on hierarchically in deeper hierarchies), and other relevant monitors will respond; and (3) communication with the other copies is performed simultaneously as part of finding them through the hierarchical broadcasts on buses.

Let us examine the path of a read miss more closely, assuming a shared physical address space. A BusRd request appears on the local B_1 bus. If the remote access cache, the local memory, or another local processor cache has a valid copy of the block, they will supply the data. Otherwise, either the remote cache or the local state monitor will know to pass the request onto the B_2 bus. When the request appears on B_2 , the coherence monitors of other nodes will snoop it. If a node's local state monitor determines that a valid copy of the data exists in that node, it will pass the request onto its B_1 bus, wait for the response, and put it back on the B_2 bus. If a node's remote cache contains the data and has it in shared state, it may simply place a reply on the B_2 bus; if in dirty state, it will reply and broadcast a read request on its B_1 bus to have the dirty processor cache downgrade the block to shared; and if dirty-but-stale, it will simply broadcast the read request on its B_1 bus and reply with the result obtained. In the last case, the processor cache that has the data dirty will change its state from dirty to shared and put the data on the B_1 bus. The remote cache will accept the data reply from the B_1 bus, change its state from dirty-but-stale to shared, and pass the reply onto the B_2 bus. When the data reply appears on B_2 , the requestor's coherence monitor picks it up, installs it and changes state in its remote cache if appropriate, and places it on its local B_1 bus. (If the block has to be installed in the remote cache, it may replace some other block, which will trigger a flush/invalidation request on that B_1 bus to ensure the inclusion property.) Finally, the requesting cache picks up the response to its BusRd request from the B_1 bus and stores it in shared state.

For writes, consider the specific situation shown in Figure 8.37(b), with P_0 in the left node issuing a write to location A , which is allocated in the memory of a third node (not shown). Since P_0 's own cache has the data only in shared state, an ownership request (BusUpgr) is issued on the local B_1 bus. As a result, the copy of A in P_1 's cache is invalidated. Since the block is not available in the remote cache in dirty-but-stale state (which would have been incorrect since P_1 had it in shared state), the monitor passes the BusUpgr request to bus B_2 , to invalidate any other copies in the system, and at the same time updates the state for the block in the remote cache to dirty-but-stale. In another node, P_2 and P_3 have the block in their caches in shared state. Because of the inclusion property, their associated remote cache is also guaranteed to have the block in shared state. This remote cache therefore passes the BusUpgr request from B_2 onto its local B_1 bus and invalidates its own copy. When the request appears on the B_1 bus, the copies of A in P_2 and P_3 's caches are invalidated. If there is a node on the B_2 bus whose processors are not caching the block containing A , the upgrade request will not pass onto its B_1 bus. Now suppose another processor P_4 in the left node issues a store to location B . This request will be satisfied within the local node, with P_0 's cache supplying the data and the remote cache retaining the data in dirty-but-stale state, and no transaction will be passed onto the B_2 bus.

The implementation requirements on the processor caches and cache controllers remain unchanged from those discussed in Chapter 6. However, some constraints do apply to the remote access cache. It should be larger than the sum of the processor caches and quite associative to maintain inclusion without excessive replacements. It should also be *lockup-free*; that is, able to handle multiple requests at a time from processors in the local node while some requests are still outstanding (more on this in Chapter 11). Finally, whenever a block is replaced from the remote cache, an invalidation or flush request must be issued on the B_1 bus, depending on the state of the replaced block (shared or dirty-but-stale, respectively). Minimizing the access time for the remote cache is less critical than increasing its hit rate since it is not in the critical path that affects the clock rate of the processor. Remote caches are therefore more likely to be built out of DRAM than SRAM. The remote cache controller must also deal with the nonatomicity issues in requesting and acquiring the buses that were discussed in Chapter 6.

Finally, consider write serialization and determining store completion. From our earlier discussion of how these work on a single bus in Chapter 6, it should be clear that serialization between two requests will be determined by the order in which those requests appear on the closest bus to the root on which they both appear. For writes that are satisfied entirely within the same leaf node, the order in which they may be seen by other processors—within or without that leaf—is their serialization order provided by the local B_1 bus. Likewise, for writes that are satisfied entirely within the same subtree, the order in which they are seen by other processors—within or without that subtree—is the serialization order determined by the root bus of that subtree. It is easy to see this if we view each bus hanging off a common bus as a processor and recursively use the same reasoning applied to a single bus in Chapters 5 and 6. Similarly, for the store completion detection needed for sequential consistency, a processor cannot assume its store has committed until it appears on the closest bus to the root on which it will appear. An acknowledgment (which now may have to be an explicit bus transaction) cannot be generated until that time, and even then the appropriate orders must be preserved between this acknowledgment and other transactions on the way back to the requesting processor (see Exercise 8.26). Once this acknowledgment is sent back from a bus, the invalidations themselves no longer need to be acknowledged as they make their way down toward the processor caches, as long as the appropriate orders are maintained along this path (just as with multilevel cache hierarchies in Chapter 6).

One of the earliest machines that used the approach of hierarchical snooping buses with distributed memory was the Gigamax (Wilson 1987; Woodbury et al. 1989) from Encore Corporation. The system consisted of up to eight Encore Multimax machines (each a regular snooping bus-based multiprocessor) connected together by fiber-optic links to a ninth global bus, forming a two-level hierarchy. Figure 8.38 shows a block diagram. Each node is augmented with a uniform interconnection card (UIC) and a uniform cluster (node) cache (UCC) card. The UCC is the remote access cache, and the UIC is the local state monitor. The monitoring of the global bus is done differently in the Gigamax due to its particular organization. Nodes are connected to the global bus through a fiber-optic link, so while a node's

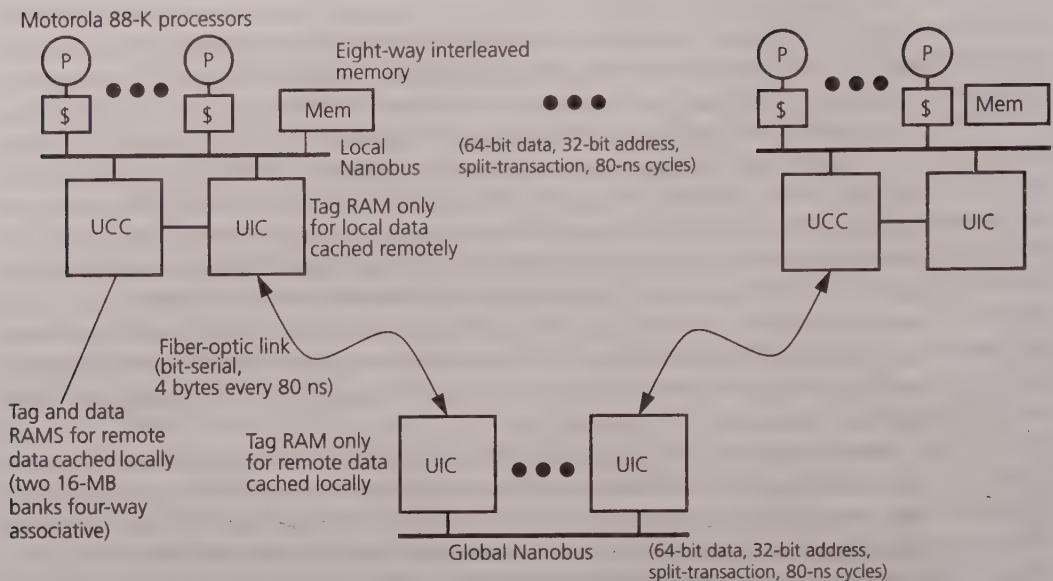


FIGURE 8.38 Block diagram for the Encore Gigamax multiprocessor. A two-level hierarchy of buses is used with memory distributed among the leaf nodes.

remote access cache (the UCC) caches remote data, it does not snoop the global bus directly. Rather, every node also has a second UIC on the global bus, which monitors global bus transactions for remote memory blocks that are cached in this local node. It then passes on the relevant requests to the local bus. If the UCC indeed sat directly on the global bus as well, the UIC on the global bus would not be necessary. The reason the Gigamax uses fiber-optic links and not a single UIC per node that sits on both buses is that high-speed buses are usually short: the Nanobus used in the Encore Multimax and Gigamax is 1 foot long (light travels 1 foot in a nanosecond, hence the name Nanobus). Since each node is at least 1 foot wide and the global bus is also 1 foot wide, flexible cabling is needed to hook these together. With fiber, links can be made quite long without affecting their transmission capabilities.

The extension of snooping cache coherence to hierarchies of rings is much like the extension to hierarchies of buses with distributed memory. Figure 8.39 shows a block diagram. The local rings and the associated processors constitute nodes, and these are connected by one or more global rings. The coherence monitor takes the form of an inter-ring interface, serving the same roles as the coherence monitor in a bus hierarchy.

Hierarchical Directory Schemes

Hierarchical directory schemes use point-to-point network transactions rather than snooping. However, as discussed earlier, unlike in flat directory schemes, the source

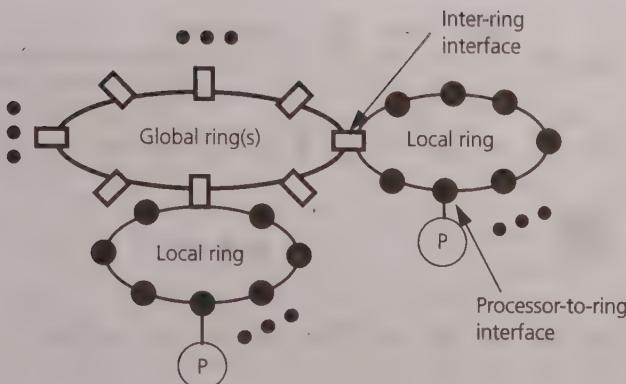


FIGURE 8.39 Block diagram for a hierarchical ring-based multiprocessor. In the two-level hierarchy shown, each local ring is a node as viewed by the global ring, and an inter-ring interface propagates relevant transactions between the two.

of the directory information in hierarchical directories is not found by going to a fixed node. The locations of copies are found neither at a fixed home node nor by traversing a distributed list pointed to by that home. Invalidations messages are not sent directly to the nodes with copies. Rather, all these activities are performed by sending messages up and down a hierarchy (tree) built upon the nodes, with the only direct communication being between parents and children in the tree.

At first blush, the organization of hierarchical directories is much like hierarchical snooping. Consider the example shown in Figure 8.40. The processing nodes are at the leaves of the tree and main memory is distributed along with the processing nodes. Every block has a home memory (leaf) in which it is allocated, but this does not mean that the directory information is maintained or rooted there. The internal nodes of the tree are not processing nodes but only hold directory information. Each such directory node keeps track of all memory blocks that are being cached or recorded by its subtrees. It uses a presence vector per block to tell which of its subtrees have copies of the block and a bit to tell whether one of them has it dirty. It also records information about local memory blocks (i.e., blocks allocated in the local memory of one of its descendants) that are being cached by processing nodes outside its subtree. As with hierarchical snooping, this information is used to decide when requests originating within the subtree should be propagated further up the hierarchy. Since the amount of directory information to be maintained by a directory node that is close to the root can become very large, the directory information is usually organized as a cache to reduce its size and maintains the inclusion property with respect to its children's caches or directories. This requires that on a replacement from a directory cache at a certain level of the tree, the replaced block must be flushed out of all of its descendent directories in the tree as well. Similarly, replacement of the information about a block allocated within that subtree requires that copies of the block in nodes outside the subtree be invalidated or flushed. These operations can be quite expensive.

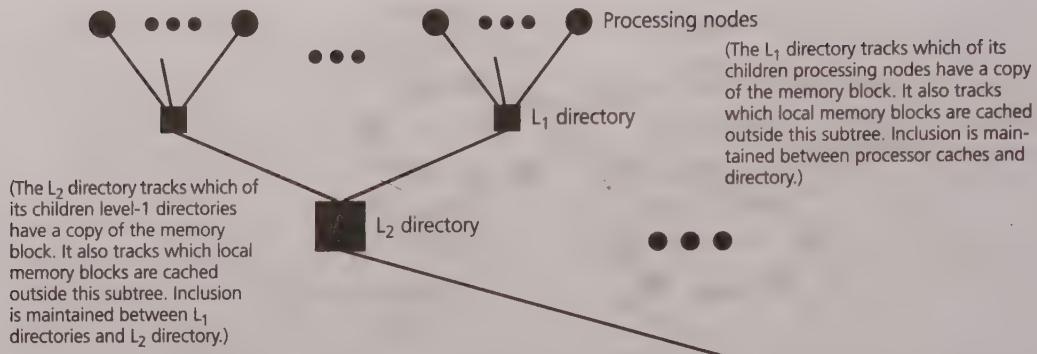


FIGURE 8.40 Organization of hierarchical directories. The processing nodes are at the leaves of the logical tree, and the internal nodes contain only directory information. There is one logical tree for each cached memory block. Logical trees may be embedded in any physical hierarchy.

A read miss from a node flows up the hierarchy either until a directory indicates that its subtree has a copy (clean or dirty) of the memory block being requested or until the request reaches the directory that is the first common ancestor of the requesting node and the home node for that block, and that directory indicates the block is not dirty outside that subtree. The request then flows down the hierarchy to the appropriate processing node to pick up the data. The data reply follows the same path back, updating the directories on its way. If the block was dirty, a copy of the block also finds its way to the home node.

A write miss in the cache flows up the hierarchy until it reaches a directory whose subtree contains the current owner of the requested memory block. The owner is either the home node, if the block is clean, or a dirty cache. The request travels down to the owner to pick up the data, and the requesting node becomes the new owner. If the block was previously in clean state, invalidations are also propagated through the hierarchy to all nodes caching that memory block. Finally, all directories involved in the preceding memory operation are updated to reflect the new owner and the invalidated copies.

In hierarchical snoopy schemes, the interconnection network is physically hierarchical to permit the snooping. With point-to-point communication, hierarchical directories do not need to rely on physically hierarchical interconnects. The hierarchy discussed here is a logical hierarchy, or a hierarchical data structure. It can be implemented either on a network that is physically hierarchical (that is, an actual tree network with directory caches at the internal nodes and processing nodes at the leaves) or on a general, nonhierarchical network such as a mesh with the hierarchical directory embedded in this general network. In fact, there is a separate hierarchical directory structure for every block that is cached. Thus, the same physical node in a general network can be a leaf (processing) node for some blocks and an internal (directory) node for others (see Figure 8.41).

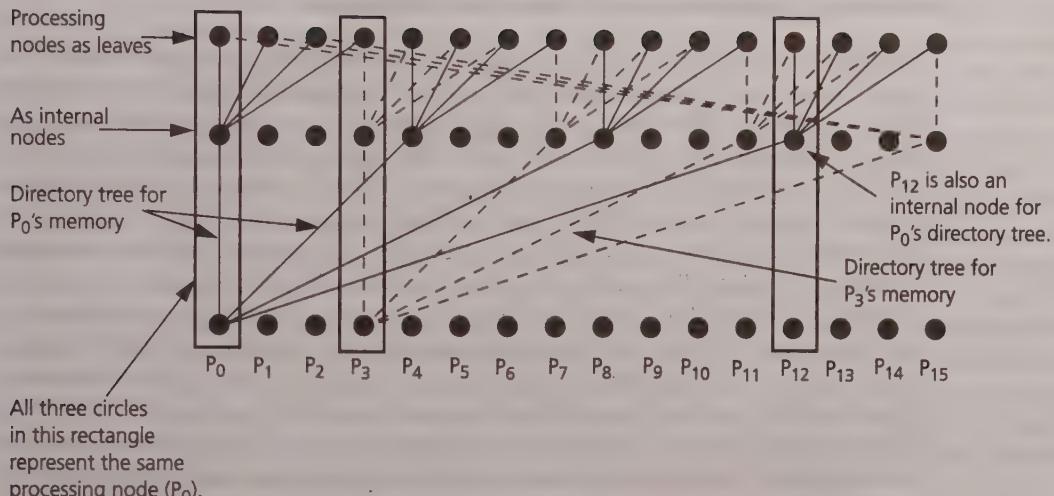


FIGURE 8.41 A multirooted hierarchical directory embedded in an arbitrary network. A 16-node hierarchy is shown. For the blocks in the portion of main memory that is located at a processing node, that node itself is the root of the (logical) directory tree. Thus, for P processing nodes, there are P directory trees. The figure shows only two of these. In addition to being the root for its local memory's directory tree, a processing node is also an internal node in the directory trees for the other processing nodes. The address of a memory block implicitly specifies a particular directory tree and guides the physical traversals to get from parents to children and vice versa in this directory tree.

Finally, the storage overhead of the hierarchical directory has attractive scaling properties. It is the cost of the directory caches at each level. The number of entries in the directory goes up as we go further up the hierarchy toward the root (to maintain inclusion without excessive replacements), but the number of directories becomes smaller. As a result, the total directory memory needed for all directories at any given level of the hierarchy is typically about the same. The directory storage needed is not proportional to the size of main memory but rather to that of the caches in the processing nodes, which is attractive. The overall directory memory overhead relative to main memory is proportional to

$$\frac{C \times \log_b P}{M \times B}$$

where C is the cache size per processing node at the leaf, M is the main memory per node, B is the memory block size in bits, b is the branching factor of the hierarchy, and P is the number of processing nodes at the leaves (so $\log_b P$ is the number of levels in the tree). More information about hierarchical directory schemes can be found in the literature (Scott 1991; Wallach 1992; Hagersten 1992; Joe 1995).

Performance Implications of Hierarchical Coherence

Hierarchical protocols, whether snoopy or directory, have some potential performance advantages that are extensions of the advantages of the two-level protocols discussed earlier. One is the combining of requests for a block as they go up and down the hierarchy. If a processing node is waiting for a memory block to arrive, another processing node that requests the same block can observe at their common ancestor directory that the block has already been requested. It can then wait at the intermediate directory and accept the response when it comes back rather than send a duplicate request. This combining of transactions can reduce traffic and, hence, contention. The sending of invalidations and gathering of invalidation acknowledgments can also be done hierarchically through the tree structure. Another advantage is that upon a miss, if a nearby node in the hierarchy has a cached copy of the block, then the block can be obtained from that nearby node (cache-to-cache sharing) rather than having to go to the home, which may be much further away in the network topology. This can reduce transit latency as well as contention at the home. Of course, this second advantage depends on how well locality in the hierarchy maps to locality in the underlying physical network as well as how well the sharing patterns of the application match the hierarchy.

While locality in the tree network can reduce transit delay on links, particularly for very large machines, the overall latency and bandwidth characteristics are usually not advantageous for hierarchical schemes. Consider hierarchical snooping schemes first. With buses, there is a bus transaction and snooping latency at every bus along the way. With rings, traversing rings at every level of the hierarchy further increases latency to potentially very high levels. For example, the uncontended latency to access a location on a remote ring in a fully populated Kendall Square Research KSR1 machine (Frank, Burkhardt, and Rothnie 1993) was higher than 25 microseconds (Saavedra, Gaines, and Carlton 1993), so other architectural techniques (discussed in Chapter 9) were used to reduce ring remote capacity misses. The commercial systems that have used hierarchical snooping have tended to use quite shallow hierarchies (the largest KSR machine was a two-level ring hierarchy with up to 32 nodes per ring). The fact that there are several processors per node also implies that the bandwidth between a node and its parent or child must be large enough to sustain their combined demands. The processors within a node will compete not only for bus or link bandwidth but also for snoop bandwidth and for the occupancy, buffers, and request tracking mechanisms of the node-to-network interface. To alleviate link bandwidth limitations near the root of the hierarchy, multiple buses or rings can be used closer to the root; however, bandwidth scalability in practical hierarchical systems remains quite limited.

For hierarchical directories, the latency problem is that the number of network transactions sent up and down the hierarchy to satisfy a request tends to be larger than in a flat, memory-based scheme. Even though these transactions may be more localized in the network, each one is a full-fledged network transaction that also requires either looking up or modifying the directory at its (intermediate) destination node. This increased endpoint overhead at the nodes along the critical path

tends to far outweigh any reduction in the total number of network hops traversed and hence network delay, especially given the characteristics of modern networks. Although some pipelining can be used—for example, the data reply can be forwarded toward the requesting node while a directory node is being updated—in practice, the latencies can still become quite large compared to machines with no hierarchy (Hagersten 1992; Joe 1995). Hierarchies with large branching factors can alleviate the latency problem but they increase contention. As with hierarchical snooping, the root of the directory hierarchy can become a bandwidth bottleneck, for both link bandwidth and directory lookup bandwidth. Multiple links may be used closer to the root (particularly appropriate for physically hierarchical networks [Leiserson et al. 1996]), and the directory cache may be interleaved among them. Alternatively, since each block has a separate logical hierarchy, a multirooted directory hierarchy may be embedded in a nonhierarchical, scalable point-to-point interconnect (Scott 1991; Wallach 1992; Scott and Goodman 1993). Figure 8.41 shows a possible organization. Like hierarchical directory schemes themselves, however, these techniques have only been in the realm of research so far.

8.11 CONCLUDING REMARKS

Scalable systems that support a coherent shared address space are an increasingly important part of the multiprocessor landscape since they combine the ease of programming of a coherent shared address space programming model with the scaling advantages of a distributed memory and interconnect. Hardware support for cache coherence is becoming increasingly popular in commercial multiprocessors designed for both technical and commercial workloads. Most of these systems use directory-based protocols, whether memory based or cache based. They are found to perform well, at least at the moderate scales at which they have been built so far, and to afford significant ease of programming compared to explicit message passing for many applications.

Directory-based cache coherence protocols are quite complex, with many transient states and “corner cases” to deal with. Figure 8.42 conveys a sense of the complexity by showing the almost complete state transition diagrams of the Origin2000 and NUMA-Q protocols.

While supporting cache coherence in hardware has a significant design cost, it is alleviated by increased experience, the appearance of standards, and the fact that microprocessors themselves provide support for cache coherence. Once the microprocessor coherence protocol is available designers can develop the multiprocessor protocol and communication architecture even before the microprocessor is ready so that not so much of a lag occurs between the two. Commercial multiprocessors today typically use the latest microprocessors available at the time they ship, alleviating the fear that multiprogrammers would have to play catch-up with the processor technology curve.

Some interesting open questions for hardware-coherent shared address space systems include whether their performance on real applications will indeed scale to

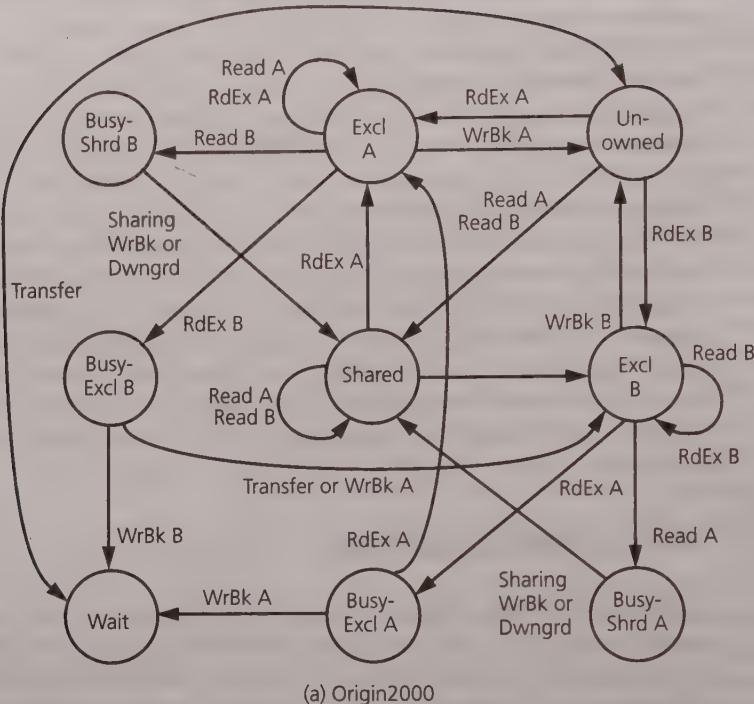


FIGURE 8.42 Expanded directory state diagrams for the case study multiprocessors of this chapter. The state diagram for the SGI Origin2000 in (a) is quite simplified: it shows the busy states at the directory but leaves out I/O operations, the poisoned state, and several race conditions. To show the use of busy states, accesses from two nodes A and B are shown. For example, a state labeled “Excl A” means that the directory thinks the block is in exclusive state in node A, and an arc labeled “RdEx B” indicates a read-exclusive operation from node B. The transfer operation and the wait state are used to handle write backs, as described in the text. The state diagram for the Sequent NUMA-Q in (b) is much more complete, though it also excludes a few corner cases. The arcs are not labeled in this diagram and several of the state labels are not explained; the purpose of this diagram is not to convey the complete protocol but simply to show that full-blown state transition diagrams can become quite complex in real systems.

large processor counts (and whether significant changes to current protocols will be needed for this), whether the appropriate node for a scalable system will be a small-scale multiprocessor or a uniprocessor, the extent to which commodity communication architectures will be successful in supporting this abstraction efficiently, and the success with which a communication assist can be designed that supports the most appropriate mechanisms for both cache coherence and explicit message passing. Some critical hardware/software trade-offs for coherent shared address space systems are discussed in the next chapter.

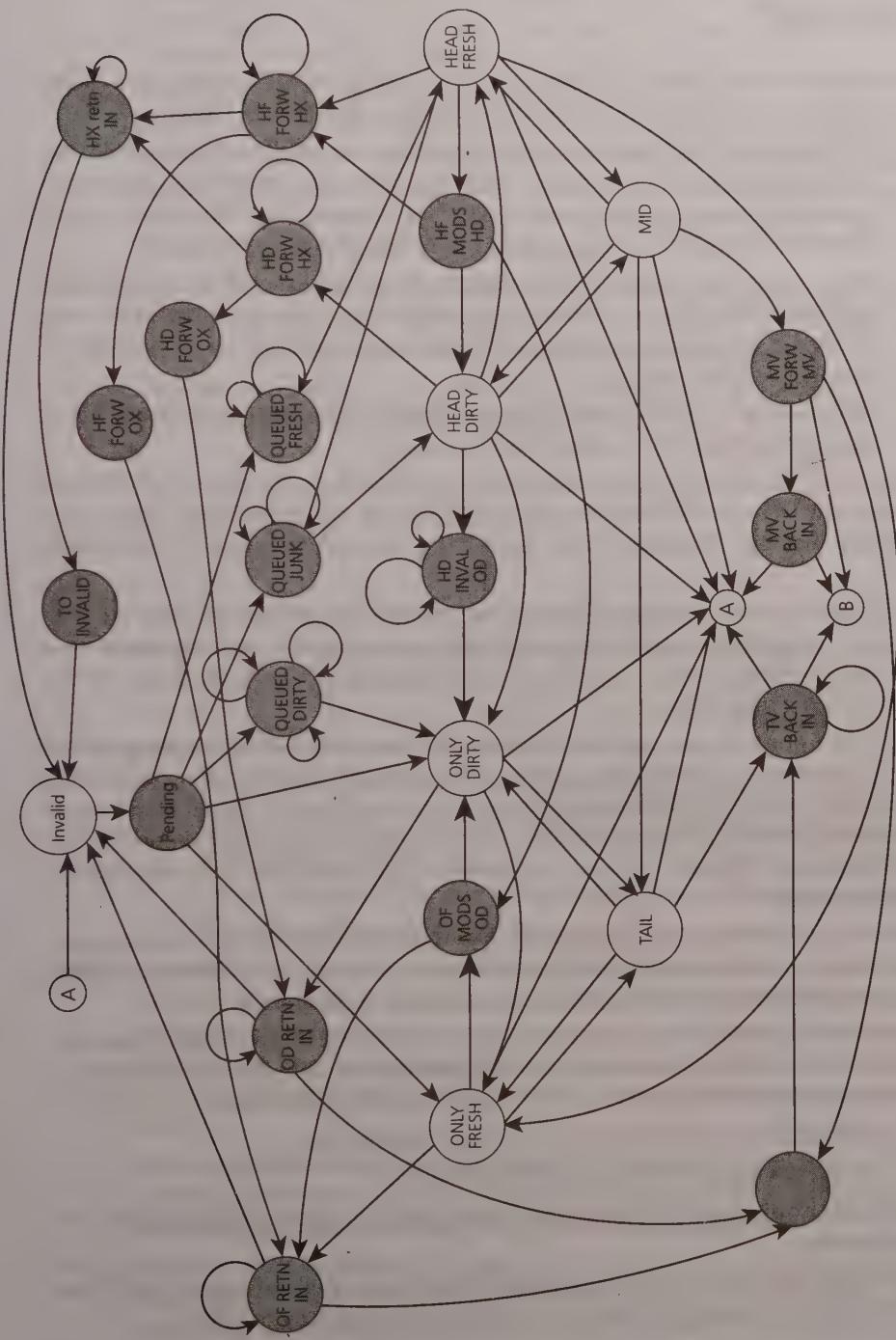


FIGURE 8.42 Expanded directory state diagrams for the case study multiprocessors of this chapter

(b) NUMA-Q

8.12 EXERCISES

- 8.1 What are the inefficiencies and efficiencies in emulating message passing on a cache-coherent machine compared to the kinds of machines discussed in Chapter 7?
- 8.2
 - a. For which of the case study parallel applications used in this book do you expect a substantial advantage in using multiprocessor rather than uniprocessor nodes (assuming the same total number of processors)? For which do you think there might be disadvantages, and under what circumstances?
 - b. How might your answer to the previous question differ with increasing scale of the machine? That is, how do you expect the performance benefits of using fixed-size multiprocessor nodes to change as the machine size is increased to hundreds of processors?
 - c. Are there any special benefits that the Illinois MESI coherence scheme offers for organizations with multiprocessor nodes?
- 8.3 Given a 512-processor system in which each node visible to the directory has 8 processors and 1 GB of main memory and a cache block size of 64 bytes, what is the directory memory overhead for (a) a full bit vector scheme, and (b) Dir_iB with $i = 3$?
- 8.4 The chapter provided diagrams showing the network transactions for strict request-response, intervention forwarding, and reply forwarding for read operations in a flat, memory-based protocol like that of the SGI Origin (see Figure 8.12). Do the same for write operations.
- 8.5 The Origin protocol assumed that acknowledgments for invalidations are gathered at the requestor. An alternative is to have the acknowledgments sent back to the home (from where the invalidation requests come) and have the home send a single acknowledgment back to the requestor. This solution is used in the Stanford FLASH multiprocessor. What are the main performance and complexity trade-offs between these two choices?
- 8.6 Draw the network transaction diagrams (like those in Figure 8.16) for an uncached read-shared request, an uncached read-exclusive request, and a write-invalidate request in the Origin protocol. State one example of a use of each.
- 8.7 Instead of the doubly linked list used in the SCI protocol, it is possible to use a singly linked list. What is the advantage? Describe what modifications would need to be made to the following operations if a singly linked list were used:
 - a. Replacement of a cache block that is in a sharing list.
 - b. Write to a cache block that is in a sharing list.Qualitatively discuss the effects this might have on large-scale multiprocessor performance.
- 8.8 How might you reduce the latency of writes that cause invalidations in the SCI protocol? Draw the network transactions. What are the major trade-offs?

- 8.9 When a variable exhibits migratory sharing, a processor that reads the variable will be the next one to write it. What kinds of protocol optimizations could you use to reduce traffic and latency in this case, and how would you detect the situation dynamically? Describe a scheme or two in some detail.
- 8.10 Another pattern that might be detected dynamically is a producer-consumer pattern, in which one processor repeatedly writes (produces) a variable and another processor repeatedly reads (consumes) it. Is the standard MESI invalidation-based protocol well suited to this? Why or why not? What enhancements or protocol might be better, and what are the savings in latency or traffic? How would you dynamically detect and employ the changes?
- 8.11 Why is write atomicity more difficult to provide with update protocols than with invalidation-based protocols in directory-based systems? How would you solve the problem? Does the same difficulty exist in a bus-based system?
- 8.12 Consider the following program fragment running on a cache-coherent multiprocessor, assuming all values to be 0 initially.

P_1	P_2	P_3	P_4
$A = 1$	$u = A$	$w = A$	$A = 2$
	$v = A$	$x = A$	

There is only one shared variable (A). Suppose that a writer magically knows where the cached copies are and sends updates to them directly without consulting a directory node. Construct a situation in which write atomicity may be violated, assuming an update-based protocol.

- a. Show the violation of sequential consistency that occurs in the results.
 - b. Can you produce a case where coherence is violated as well? How would you solve these problems?
 - c. Can you construct the same problems for an invalidation-based protocol?
 - d. Can you construct them for update protocols on a bus?
- 8.13 In handling write backs in the Origin protocol, we said that when the node doing the write back receives an intervention, it ignores it. Given a network that does not preserve point-to-point order, of what situations do we have to be careful in deciding to ignore the intervention? How do we detect that this intervention should be dropped? Would there be a problem with a network that preserved point-to-point order?
- 8.14 Can the serialization problems discussed for Origin in Section 8.5.2 arise even with a strict request-response protocol, and do the same guidelines apply? Show example situations, including the examples discussed in that section.
- 8.15 Consider the serialization of writes in NUMA-Q, given the two-level hierarchical coherence protocol. If a node has the block dirty in its remote cache, how might writes from other nodes that come to it get serialized with respect to writes from

- processors in this node? What transactions would have to be generated to ensure the serialization?
- 8.16 In the Origin implementation, incoming request messages to the memory/directory interface are given priority over incoming responses unless there is a danger of responses being starved. Why do you think this choice of giving priorities to requests was made? Describe some methods for how you might detect when to invert the priority. What would be the danger with responses being starved?
- 8.17
- a. Why is it necessary to flush TLBs when doing migration or replication of pages?
 - b. For a CC-NUMA multiprocessor with software-reloaded TLBs, suppose a page needs to be migrated. Which one of the following TLB flushing schemes would you pick and why: (i) only TLBs that currently have an entry for a page, (ii) only TLBs that have loaded an entry for a page since the last flush, or (iii) all TLBs in the system. [Hint: the selection should be based on the following two criteria: the cost of doing the actual TLB flush and the difficulty of tracking necessary information to implement the scheme.]
- 8.18 For a simple two-processor CC-NUMA system, the traces of cache misses for three virtual pages X, Y, Z from the two processors P_0 and P_1 are shown. Time goes from left to right. “R” is a read miss and “W” is a write miss. There are two memories M_0 and M_1 , local to P_0 and P_1 respectively. A local miss costs 1 time unit and a remote miss costs 4 units. Assume that read misses and write misses cost the same.
- Page X:
- | | | |
|---------|----------------|---------|
| P_0 : | RRRR R R RRRRR | RRR |
| P_1 : | R R R R R | RRRR RR |
- Page Y:
- | | |
|---------|------------------------|
| P_0 : | no accesses |
| P_1 : | RR WW RRRR RWRWRW WWWR |
- Page Z:
- | | |
|---------|----------------------|
| P_0 : | R W RW R R RRWRWRWRW |
| P_1 : | WR RW RW W W R |
- a. In which local memories would you place pages X, Y, and Z, assuming complete knowledge of the entire trace?
- b. Assume that all three pages were initially placed in M_0 . You have prior knowledge of the entire trace. You can do one migration, or one replication, or nothing for each page at the beginning of the trace at zero cost. What action would be appropriate for each of the pages?
- c. Answer part (b) where a page migration or replication costs 10 units. In addition, give the final memory access cost for each page.
- d. Answer part (c) where a migration or replication costs 60 units.
- e. Answer part (d) where the cache miss trace for each page is the shown trace repeated 10 times. (You still can only do one migration or replication at the beginning of the entire trace.)

- 8.19 Full-empty bits, introduced in Section 5.5, provide hardware support for fine-grained synchronization and have been proposed for CC-NUMA machines. What are the advantages and disadvantages of full-empty bits, and why do you think they are not used in modern systems?
- 8.20 With an invalidation-based protocol, lock transfers take more network transactions than necessary. An alternative to cached locks is to use uncached locks, where the lock variable stays in main memory and is always accessed at the memory itself.
- Write pseudocode for a simple lock and a ticket lock using uncached operations.
 - What are the advantages and disadvantages relative to using cached locks? Which would you deploy in a production system?
 - Can you describe a scheme that uses both cached and uncached read and write operations to improve the performance of locks? What specific operations would your scheme require?
- 8.21 Since high-contention and low-contention situations are best served by different lock algorithms, one strategy that has been proposed is to have a library of synchronization algorithms and provide hardware support to switch between them “reactively” at run time based on observed access patterns to the synchronization variable.
- Which locks would you provide in your library?
 - Assuming a memory-based directory protocol, design simple hardware support and a policy for switching between locks at run time.
 - Describe an example where this support might be particularly useful.
 - What are the potential disadvantages?
- 8.22 You are performing an architectural study using four applications: Ocean, blocked LU factorization, an FFT that performs local calculations on rows separated by a matrix transposition, and Barnes-Hut. For each application, answer the following questions, assuming a CC-NUMA system:
- What modifications or enhancements in data structuring or layout would you use to ensure good interactions with the extended memory hierarchy?
 - What are the interactions with cache size and granularities of allocation, coherence, and communication that you would be particularly careful to represent or not represent?
- 8.23 Consider the example of transposing a matrix of data in parallel, as is used in computations such as high-performance FFTs. Figure 8.43 shows the transpose pictorially. Every process transposes one “patch” of its assigned rows to every other processor, including one to itself. Before the transpose, a process has read and written its assigned rows of the source matrix of the transpose, and after the transpose it reads and writes its assigned rows of the destination matrix. The rows assigned to a process in both the source and destination matrix are allocated in its local memory. There are two ways to perform the transpose: a process can read the local elements from its rows of the source matrix and write them to the appropriate elements of the

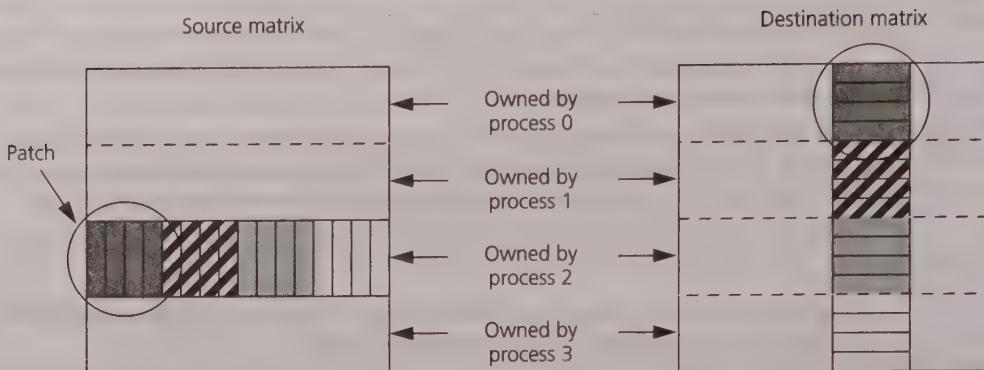


FIGURE 8.43 Sender-initiated matrix transposition. The source and destination matrices are partitioning among processes in groups of contiguous rows. Each process divides its set of n/p rows into p patches of size $(n/p) \times (n/p)$. Consider process 2 as a representative example: one patch assigned to it ends up in the assigned set of rows of every other process, and it transposes one patch (third-from-left, in this case) locally.

destination matrix, whether they are local or remote, as shown in the figure (called a sender-initiated transpose); or a process can write the local rows of the destination matrix and read the appropriate elements of the source matrix, whether they are local or remote (called a receiver-initiated transpose).

- Given an invalidation-based directory protocol, which method do you think will perform better and why?
- How do you expect the answer to (a) to change if you assume an update-based directory protocol?
- Consider the following implementation of a matrix transpose, which you plan to run on eight processors. Each processor has one level of cache, which is fully associative, 8 KB, with 128 byte lines. (Note: AT and A are not the same matrix.)

```
Transpose(double **A, double **AT)
{
    int i,j,mynum;
    GETPID(mynum);
    for (i=mynum*nrows/p; i<((mynum+1)*(nrows/p)); i++) {
        for (j=0; j<1024; j++) {
            AT[i][j] = A[j][i];
        }
    }
}
```

The input data set is a $1,024 \times 1,024$ matrix of double-precision floating-point numbers (i.e., `nrows` in 1,024), decomposed so that each processor is responsible for generating a contiguous block of rows in the transposed matrix AT (i.e., a receiver-initiated transpose). Ignoring the contention problem caused by all processors first going to processor 0, what is the major performance problem with this code? What technique would you use to solve it? Restructure the code to alleviate all performance problems as much as possible. Write the entire restructured loop.

- 8.24 Consider a hierarchical bus-based system with a centralized memory at the root of the hierarchy rather than distributed memory as discussed in the chapter. What would be the main differences in how reads and writes are satisfied? Briefly describe the path taken by reads and writes.
- 8.25 Could you construct a hierarchical bus-based system with centralized memory (say) without pursuing the inclusion property between the remote access cache and the L_1 caches in a node? If so, what complications would it cause?
- 8.26 To ensure sequential consistency in a two-level hierarchical bus design, is it okay to return an acknowledgment when the invalidation request reaches the B_2 bus? If so, what constraints are imposed on the design and implementation of the caches and the orders preserved among transactions? If not, why not? Would it be okay if the hierarchy had more than two levels?
- 8.27 Suppose two processors in two different nodes of a hierarchical bus-based machine issue an upgrade for a block at the same time. Trace their paths through the system, discussing all state changes and when they must happen as well as what precautions prevent deadlock and prevent both processors from gaining ownership.
- 8.28 An optimization in distributed-memory bus-based hierarchies is cache-to-cache sharing: if another processor's cache on the local bus can supply the data, we do not have to go to the global bus and remote node. What are the trade-offs of supporting this optimization in ring-based hierarchies?
- 8.29 What branching factor would you choose in a machine with a hierarchical directory? Highlight the major trade-offs. What techniques might you use to alleviate the performance trade-offs? Be as specific in your description as possible.
- 8.30 Is it possible to implement hierarchical directories without maintaining inclusion in the directory caches? Design a protocol that does that and discuss the advantages and disadvantages.

Hardware/Software Trade-Offs

This chapter addresses the potential limitations of the directory-based, cache-coherent systems discussed in Chapter 8 and the hardware/software trade-offs that arise in overcoming these limitations. The primary limitations of those systems are the following:

- *High waiting time at memory operations.* Sequential consistency (SC) is the memory consistency model of choice for the programmer and, so far, has been assumed for both snooping and directory-based systems. To satisfy the sufficient conditions for SC, a processor would have to wait for its previous memory operation to complete before issuing the next one. This has an even greater impact on performance in scalable systems than in bus-based systems since communication latencies are longer and more network transactions are in the critical path. Worse still, it is very limiting for compilers, which potentially cannot reorder memory operations to shared data at all if the programmer assumes sequential consistency.
- *Limited capacity for replication.* Communicated data is automatically replicated only in the processor cache, not in local main memory. This can lead to capacity misses and artifactual communication when working sets are large and include nonlocal data or when conflict misses are numerous.
- *High design and implementation cost.* The communication assist contains hardware that is specialized for supporting cache coherence and is tightly integrated into the processing node. Protocols are complex, and getting them right in hardware takes substantial design time. (By cost, here we mean the cost of hardware and of system design time. However, recall from Chapter 3 that a programming cost is also associated with achieving good performance, and approaches that reduce system cost can often increase this cost dramatically.)

This chapter focuses on these three limitations. The approaches that have been developed to address them are still controversial to varying degrees, but aspects of them are being adopted by designers of commercial parallel machines. Other limitations are often encountered as well, including the addressability limitations of a shared physical address space—as discussed for the CRAY T3D in Chapter 7—and the fact that a single protocol is hardwired into the machine. However, solutions to these problems are often incorporated in solutions to the primary problems, and they are discussed as advanced topics.

The problem of waiting too long at memory operations can be addressed in two ways in hardware. First, the implementation can be designed not to satisfy the sufficient conditions for SC, which modern nonblocking processors are not inclined to do anyway, but to satisfy the SC model itself. That is, a processor need not wait for the previous operation to complete before issuing the next one; however, the system ensures that operations do not complete or become visible out of program order. This method is used in the SGI Origin2000 system discussed in Chapter 8. Second, the memory consistency model can itself be relaxed so program order does not have to be maintained so strictly. Relaxing the consistency model changes the semantics of the shared address space and has implications for both hardware and software. It requires more care from the programmer in writing correct programs but enables the hardware to overlap and reorder operations to a greater extent. Importantly, it also allows the compiler to reorder memory operations within a process before they are even presented to hardware, as optimizing compilers are wont to do. Relaxed memory consistency models are discussed in Section 9.1.

The problem of limited capacity for replication can be addressed by automatically caching data in main memory, not just in the processor caches, and keeping this data coherent. Unlike in hardware caches, replication and coherence in main memory can be performed at a variety of granularities—for example, a cache block, a page, or a user-defined object—and can be managed either directly by hardware or through software. This provides a very rich space of protocols, hardware/software implementations, and cost-performance trade-offs. An approach directed primarily at improving performance is to manage the local main memory as a hardware cache, providing replication and coherence at cache block granularity there as well. This approach is called cache-only memory architecture, or COMA, and is discussed in Section 9.2. It relieves software from worrying about capacity misses and the initial distribution of data across main memories while still providing coherence at fine granularity and hence avoiding false sharing. However, it is hardware intensive and requires per-block tags and state to be maintained in main memory as well.

Finally, there are many approaches to addressing the problem of hardware cost. One approach is to integrate the communication assist and network less tightly into the processing node, at the cost of increasing communication latency and assist occupancy. Another is to provide automatic replication and coherence in software rather than hardware, leading to a range of possible system implementations, as illustrated in Figure 9.1. The software approaches provide replication and coherence in main memory and can operate at a variety of granularities. They enable the use of off-the-shelf commodity parts for the nodes and interconnect, reducing hardware cost but pushing much more of the (now much greater) burden of achieving good performance onto the programmer. These approaches to reduce hardware cost are discussed in Section 9.3.

The three issues are closely related. For example, cost is strongly related to the manner in which replication and coherence are managed in main memory: at what granularities and whether directly in hardware or through a run-time or operating

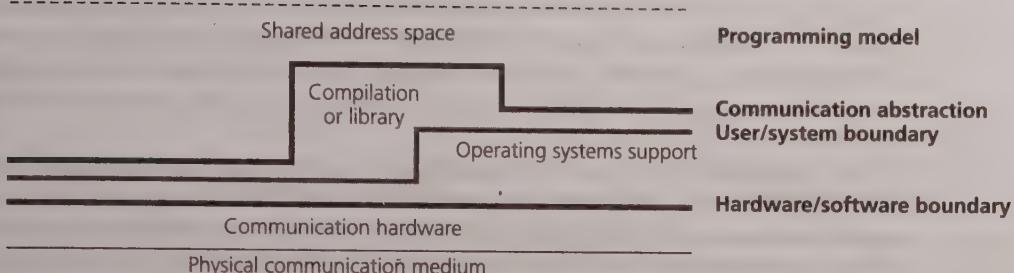


FIGURE 9.1 Layers of the communication architecture for systems discussed in this chapter.
The diagram represents the degrees to which software intervention is used to support a coherent shared address space.

system. Cost and granularity are also related to the memory consistency model: lower-cost, lower-performance solutions and larger granularities benefit more from relaxing the memory consistency model, and implementing the protocol in software makes it easier to fully exploit the relaxation of the semantics. A useful framework to understand the space of alternatives is based on the granularities at which data is allocated in the local replication store, kept coherent, and communicated. Section 9.4 constructs such a framework to summarize and relate the alternatives. This framework leads naturally to an approach that strives to achieve a good compromise between the high-cost COMA approach and the low-cost all-software approach. This approach, called Simple COMA, is discussed in Section 9.4 as well.

The implications for parallel software of the systems discussed in this chapter are explored in Section 9.5. Finally, Section 9.6 covers some advanced topics, including the techniques to address the potential limitations of a shared physical address space and a fixed coherence protocol.

9.1

RELAXED MEMORY CONSISTENCY MODELS

Recall from Chapter 5 that the memory consistency model for a shared address space specifies the constraints on the order in which memory operations (to the same or different locations) can appear to execute with respect to one another, enabling programmers to reason about the behavior and correctness of their programs. In fact, any system layer that supports a shared address space naming model has a memory consistency model: the programming model or programmer's interface, the user/system interface, and the hardware/software interface. Software that interacts with a layer must be aware of its memory consistency model. We focus mainly on the consistency model as seen by the programmer—that is, at the interface between the programmer and the rest of the system composed of the compiler,

operating system, and hardware—since that is the one with which programmers reason.¹ For example, a processor may preserve all program orders presented to it among memory operations, but if the compiler has already reordered operations then programmers can no longer reason with the simple model exported by the hardware.

The consistency model at the programmer's interface has implications for programming languages, compilers, and hardware as well. To the compiler and hardware, it indicates the constraints within which they can reorder accesses from a process and the orders that they cannot appear to violate, thus telling them what performance optimizations they can use. Programming languages must provide mechanisms to introduce such constraints if necessary, as we shall see. In general, the fewer the reorderings of memory accesses from a process that we allow the system to perform, the more intuitive the programming model we provide to the programmer but the more we constrain performance optimizations. The goal of a memory consistency model is to impose ordering constraints that strike a good balance between programming complexity and performance. The model should also be portable; that is, the specification should be implementable on many platforms so that the same program can run on all these platforms and preserve the same semantics.

The sequential consistency model that we have assumed so far provides an intuitive semantics to the programmer—program order within each process and a consistent interleaving across processes—and can be quite easily implemented by satisfying its sufficient conditions. However, its drawback is that, by preserving a strict order among accesses, it restricts many of the performance optimizations that modern uniprocessor compilers and microprocessors employ. With the high cost of memory access, computer systems achieve higher performance by reordering or overlapping the servicing of multiple memory or communication operations from a processor. Preserving the sufficient conditions for SC clearly does not allow for much reordering or overlap in hardware, and approaches that preserve SC without preserving the sufficient conditions also have limitations. With SC at the programmer's interface, the compiler cannot reorder memory accesses even if they are to different locations, thus disallowing critical performance optimizations such as code motion, common-subexpression elimination, software pipelining, and even register allocation as illustrated in Example 9.1.

EXAMPLE 9.1 Show how register allocation can lead to a violation of SC even if the hardware satisfies SC.

Answer Consider the code fragment shown in Figure 9.2(a). After register allocation, the code produced by the compiler and seen by hardware might look like that in Figure 9.2(b). The result $(u, v) = (0, 0)$ is disallowed under SC hardware in (a) but not

1. The term “programmer” here refers to the entity that is responsible for generating the parallel program. For example, if a human programmer writes a sequential program that is automatically parallelized by system software, then it is the system software that has to deal with the memory consistency model; the programmer simply assumes sequential semantics as on a uniprocessor.

P_1	P_2	P_1	P_2
$B = 0$	$A = 0$	$r1 = 0$	$r2 = 0$
$A = 1$	$B = 1$	$A = 1$	$B = 1$
$u = B$	$v = A$	$u = r1$	$v = r2$
		$B = r1$	$A = r2$

(a) Before register allocation

(b) After register allocation

FIGURE 9.2 Example showing how register allocation by the compiler can violate SC. The code in (a) is the original code with which the programmer reasons. $r1$, $r2$ are registers, and the code in (b) is as it might appear after register allocation performed by the compiler.

only can but will be produced by SC hardware in (b). In effect, register allocation reorders the write of A and the read of B on P_1 and reorders the write of B and the read of A on P_2 . A uniprocessor compiler might easily perform these optimizations in each process: they are valid for sequential programs since the reordered accesses are to different locations. ■

Providing SC at the programmer's interface implies supporting SC at lower-level interfaces, including the hardware/software interface. If the sufficient conditions for SC are met, a processor waits for an access to complete or at least commit before issuing the next one, so most of the latency suffered by memory references is directly seen by processors as stall time. Although a processor may continue executing non-memory instructions while a single outstanding memory reference is being serviced, the expected benefit from such overlap is tiny, since even without instruction-level parallelism, on average every third instruction is a memory reference (Hennessy and Patterson 1996). We need to do something about this performance problem.

One approach we can take is to preserve sequential consistency at the programmer's interface but find ways to hide the long latency stalls from the processor. This can be done in several ways, which fall into two categories (Gharachorloo, Gupta, and Hennessy 1991). The techniques and their performance implications are discussed further in Chapter 11; here, we simply provide an intuition about them. In the first category, the system still preserves the sufficient conditions for SC and the compiler does not reorder memory operations. Latency tolerance techniques such as prefetching of data or multithreading are used to overlap data transfers with one another or with computation—thus hiding much of their latency from the processor—but the actual read and write operations are not issued before previous ones complete in program order.

In the second category, the system preserves SC but not the sufficient conditions at the programmer's interface. The compiler can reorder operations as long as it can guarantee that sequential consistency will not be violated in the results. Compiler algorithms have been developed for this (Shasha and Snir 1988; Krishnamurthy and Yelick 1994, 1995), but they are expensive and their analysis is currently quite conservative. At the hardware level, memory operations are issued and executed out of

program order but are guaranteed to become visible to other processors in program order. This approach is well suited to dynamically scheduled processors that use an instruction lookahead buffer to find independent instructions to issue; for example, the R10000 processor in the SGI Origin2000. The instructions are inserted in the lookahead buffer in program order; they are chosen from the instruction lookahead buffer and executed out of order, but they are guaranteed to retire from the lookahead buffer in program order. Operations may even issue and execute out of order past an unresolved branch in the lookahead buffer based on branch prediction—called *speculative execution*—but since the branch will be resolved and retire before them, they will not become visible to the register file or external memory system before the branch is resolved. If the branch was mispredicted, the effects of those operations will never become visible. The technique called *speculative reads* goes a little further. Here, the values returned by reads are used even before they are known to be correct; later checks determine if they were incorrect, and if so, the computation is rolled back to reissue the read. Note that it is not possible to speculate with stores in this manner because once a store is made visible to other processors, it is extremely difficult to roll back and recover: a store's value should not be made visible to other processors or the external memory system environment until all previous references have correctly completed.

Some or all of these techniques are supported by many modern microprocessors, such as the MIPS R10000, the HP PA-8000, and the Intel Pentium Pro. However, while they are increasingly popular, they require substantial hardware resources and complexity, their success at hiding multiprocessor latencies is not yet clear (see Chapter 11), and not all processors support them. Perhaps most critically, these techniques work for processors, but they do not help compilers perform the reorderings of memory operations that are critical for their optimizations.

A completely different way to overcome the performance limitations imposed by SC is to change the memory consistency model itself; that is, not to guarantee such strong ordering constraints to the programmer but still retain semantics that are intuitive enough to be useful. By relaxing the ordering constraints, these *relaxed consistency models* allow the compiler to reorder accesses before presenting them to the hardware, at least to some extent. At the hardware level, they allow multiple memory accesses from the same process not only to be outstanding at a time but even to complete or become visible out of order, thus allowing much of the latency to be overlapped and hidden from the processor. The intuition behind relaxed models is that SC is usually too conservative; many of the orders it preserves are not really needed to satisfy a programmer's intuition in most situations. Detailed treatments of relaxed memory consistency models can be found in (Adve 1993; Gharachorloo 1995; Adve and Gharachorloo 1996).

Consider the simple example shown in Figure 9.3. On the left are the orderings that will be maintained by an SC implementation. On the right are the orderings that are necessary for intuitively correct program semantics. The latter are far fewer. For example, writes to variables A and B by P_1 can be reordered in this case without affecting the results observed by the program; all we must ensure is that both of them complete before the variable flag is set to 1. Similarly, reads to variables A and

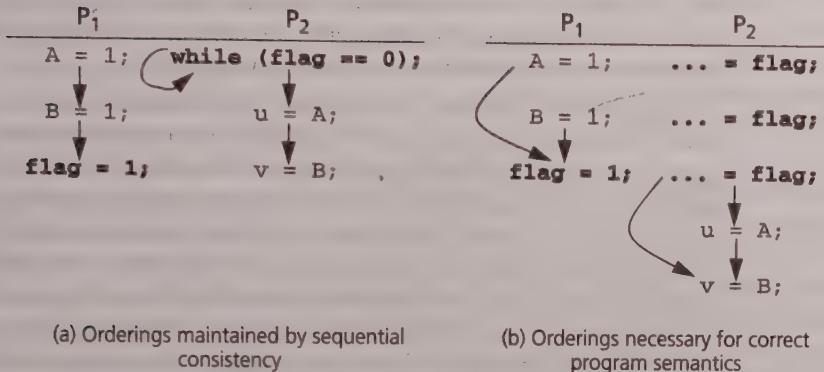


FIGURE 9.3 Intuition behind relaxed memory consistency models. The arrows in the figure indicate the orderings maintained. Part (a) shows the orderings maintained by the sequential consistency model. Part (b) shows the orderings that are necessary for “correct” or “intuitive” semantics. Bold font indicates that the accesses to the `flag` variable are the important ones for ordering and are in fact being used to orchestrate event synchronization.

`B` can be reordered at P_2 once `flag` has been observed to change to value 1.² Even with these reorderings, the results look just like those of an SC execution. On the other hand, although the accesses to `flag` are also simple variable accesses, a model that allowed them to be reordered with respect to `A` and `B` at either process would compromise the intuitive semantics and SC results. It would be wonderful if system software or hardware could automatically detect which program orders are critical to maintaining SC semantics and allow the others to be violated for higher performance (Shasha and Snir 1998). However, the problem is intractable (in fact, undecidable) for general programs, and inexact solutions are often too conservative to be very useful.

A complete solution for a relaxed consistency model consists of three parts:

1. *The system specification.* This is a clear specification of two things: first, what program orders among memory operations are guaranteed to be preserved, in an observable sense, by the system, including whether write atomicity will be maintained; and second, if not all program orders are guaranteed to be preserved by default, then what mechanisms the system provides for a programmer to enforce order explicitly when desired. As should be clear by now, the compiler and the hardware have their own system specifications, but we focus on the specification that the two together or the system as a whole presents to
2. Actually, it is possible to further weaken the requirements for correct execution. For example, it is not necessary for writes to `A` and `B` to complete before the write to `flag` is done; it is only necessary that they be complete by the time processor P_2 observes that the value of `flag` has changed to 1. It turns out that such relaxed models are difficult to implement in hardware. In software, some of these more relaxed models make sense, and we discuss them in Section 9.2.

the programmer. For a processor architecture, the specification it exports governs the reorderings that it allows and the order-preserving primitives it provides and is often called the *processor's memory model*.

2. *The programmer's interface.* The system specification is itself a consistency model. A programmer may use it to reason about correctness and insert the appropriate order-preserving mechanisms. However, this is a very low-level interface for a programmer: parallel programming is challenging enough without having to think about reorderings and write atomicity! The specific reorderings and order-enforcing mechanisms supported are different across system specifications, compromising portability. What a programmer therefore wants is a methodology for writing "safe" programs. This is a contract such that if the program follows certain high-level rules or provides enough program annotations—such as telling the system that `flag` in Figure 9.3 is in fact used as a synchronization variable—then any system on which the program runs will always guarantee a sequentially consistent execution, regardless of the default reorderings permitted by the system specifications it supports. The programmer's responsibility is to follow the rules and provide the annotations, which hopefully does not involve reasoning at the level of potential reorderings. The system's responsibility is to use the rules and annotations as constraints to maintain the illusion of sequential consistency. The implication for programming languages is that they should support the necessary annotations and provide an intuitive programming interface.
3. *The translation mechanism.* This translates the programmer's annotations to the interface (specifically, the order-preserving mechanisms) exported by the system specification, so that the system may do its job.

In the following discussion of relaxed consistency models, we first examine different low-level specifications exported by systems and particularly by microprocessors. Then Section 9.1.2 discusses the programmer's interface or contract and how the programmer might provide the necessary annotations. Section 9.1.3 briefly discusses translation mechanisms. Section 9.1.4 discusses current practice with regard to memory consistency models. A detailed treatment of implementation complexity and performance benefits is postponed until we discuss latency tolerance mechanisms in Chapter 11.

9.1.1 The System Specification

Several different reordering specifications have been proposed by microprocessor vendors and by researchers, each with its own mechanisms for enforcing orders. These include *total store ordering* (TSO) (Sindhu, Frailong, and Cekleov 1991; Sun Microsystems 1991), *partial store ordering* (PSO) (Sindhu, Frailong, and Cekleov 1991; Sun Microsystems 1991), and *relaxed memory ordering* (RMO) (Weaver and Germond 1994) from the Sun Sparc V8 and V9 specifications; *processor consistency* (PC) described in (Goodman 1989; Gharachorloo 1990) and used in the Intel Pentium processors; *weak ordering* (WO) (Dubois, Scheurich, and Briggs 1986; Dubois

and Scheurich 1990); *release consistency* (RC) (Gharachorloo 1990); and the Digital Alpha (Sites 1992) and IBM/Motorola PowerPC (May et al. 1994) models. Of course, a particular implementation of a processor may not support all the reorderings that its system specification allows. The system specification defines the semantic interface for that architecture, that is, what reorderings the programmer must assume might happen; the implementation determines what reorderings actually happen and how much performance can actually be gained.

Let us discuss some of the specifications or consistency models, using the relaxations in program order that they allow as our primary axis for grouping models together (Gharachorloo 1995). The first set of models, which includes TSO and PC, only allows a read to bypass (complete before) an earlier incomplete write in program order (i.e., allows the write → read order to be reordered). The next set, which includes PSO, also allows writes to bypass previous writes (i.e., write → write reordering). The final set, which includes WO, RC, RMO, Alpha, and PowerPC, allows reads or writes to bypass previous reads as well (i.e., allows all reorderings among read and write accesses). A read-modify-write operation is treated as being both a read and a write, so it is reordered with respect to another operation only if both a read and a write can be reordered with respect to that operation. In all cases, we assume basic cache coherence—write propagation and write serialization—and that uniprocessor data and control dependences are maintained within each process. The specifications discussed have in most cases been motivated by and defined for the processor architectures themselves, that is, the hardware interface. All are applicable to compilers as well; however, since sophisticated compiler optimizations require the ability to reorder all types of accesses, most compilers have not supported as wide a variety of ordering models. In fact, at the programmer's interface, all but the last set of models have limited utility because they do not allow many important compiler optimizations.

Relaxing the Write-to-Read Program Order

The main motivation for this class of models is to allow the hardware to hide the latency of write operations. While the write miss is still in the write buffer and not yet visible to other processors, the processor can issue and complete reads that hit in its cache or even a single read that misses in its cache. The benefits of hiding write latency can be substantial, as we see in Chapter 11, and most processors can take advantage of this relaxation.

The models in this class (like TSO and PC) preserve the programmer's intuition quite well, for the most part, even without any special operations. For example, the common idiom of spinning on a flag for event synchronization works without modification (Figure 9.4[a]). This is because TSO and PC models preserve the ordering of writes so that the write of the flag is not visible until all previous writes in program order have completed in the system. For this reason, most early multiprocessors supported one of these two models, including the Sequent Balance, Encore Multimax, Vax-8800, SparcCenter 1000/2000, SGI 4D/240, SGI Challenge, and even

P_1	P_2		P_1	P_2
$A = 1;$	while ($\text{Flag} == 0$);		$A = 1;$	print B ;
$\text{Flag} = 1;$	print A ;		$B = 1;$	print A ;
	(a)			(b)
P_1	P_2	P_3	P_1	P_2
$A = 1;$	while ($A == 0$);	while ($B == 0$);	$A = 1; \text{(i)}$	$B = 1; \text{(iii)}$
$B = 1;$		print A ;	print $B; \text{(ii)}$	print $A; \text{(iv)}$
	(c)			(d)

FIGURE 9.4 Example code sequences repeated to compare TSO, PC, and SC. Both TSO and PC provide the same results as SC for code segments (a) and (b), PC can violate SC semantics for segment (c) (TSO still provides SC semantics), and both TSO and PC violate SC semantics for segment (d).

the Pentium Pro quad, and it has been relatively easy to port even complex programs, such as the operating systems, to these machines.

Of course, the semantics of these models is not SC, so there are situations in which the differences show through. Figure 9.4 shows four code examples, three of which we have seen earlier, in which we assume that all variables start out having the value 0. Code fragment (a) is the example of spinning on a flag. In fragment (b), SC guarantees that if B is printed as 1 then A too will be printed as 1, since the writes of A and B by P_1 cannot be reordered. For the same reason, TSO and PC also have the same semantics in this fragment as well. For fragment (c), only TSO offers SC semantics and prevents A from being printed as 0, not PC. The reason is that PC does not guarantee write atomicity. Finally, for fragment (d), no interleaving of the operations under SC can result in 0 being printed for both A and B . To see why, consider that program order implies the precedence relationships (i) → (ii) and (iii) → (iv) in the interleaved total order. If $B = 0$ is observed, it implies (ii) → (iii), which therefore implies (i) → (iv). But (i) → (iv) implies A will be printed as 1. Similarly, a result of $A = 0$ implies $B = 1$. A popular software-only mutual exclusion algorithm called Dekker's algorithm—used in the absence of hardware support for atomic read-modify-write operations (Tanenbaum and Woodhull 1997)—relies on the property that both A and B will not be read as 0 in this case. SC provides this property, further contributing to its view as an intuitive consistency model. Neither TSO nor PC guarantees it since they both allow the read operation corresponding to the print to complete before previous writes are visible.

To ensure SC semantics when desired (e.g., to port a program written under SC assumptions to a TSO or PC system), we need mechanisms to enforce two types of extra orderings: (1) to ensure that a read does not complete before an earlier write in program order (applies to both TSO and PC) and (2) to ensure write atomicity for a read operation (applies only to PC). For the former, different processor architectures provide somewhat different solutions. For example, the Sun Sparc V9 specification (Weaver and Germond 1994) provides *memory barrier* (MEMBAR) or *fence* instruc-

tions of different flavors that can ensure any desired ordering. Here, we would insert a write-to-read ordering flavored MEMBAR before the read. This MEMBAR prevents any read that follows it in program order from issuing before all writes that precede it have completed. On architectures that do not provide memory barrier instructions, it is possible to achieve this effect by substituting an atomic read-modify-write operation or sequence for the original read. A read-modify-write is treated as being both a read and a write, so it cannot be reordered with respect to previous writes in these models. Of course, the value written in the read-modify-write must be the same as the value read to preserve correctness. Replacing a read with a read-modify-write also guarantees write atomicity at that read on machines supporting the PC model. The details of why this works are subtle, and the interested reader can find them in the literature (Adve et al. 1993).

Relaxing the Write-to-Read and Write-to-Write Program Orders

Allowing writes as well to bypass earlier writes (to different locations) allows the write buffer to merge and even retire writes before previous writes in program order complete. Thus, it enables multiple write misses to be fully overlapped and to become visible out of program order. The motivation is to further reduce the impact of write latency on processor stall time and to improve communication efficiency between processors by making new data values visible to other processors sooner. Sun Sparc's PSO model (Sindhu, Frailong, and Cekleov 1991; Sun Microsystems 1991) is the only model in this category. Like TSO, it guarantees write atomicity.

Unfortunately, reordering of writes can violate our intuitive SC semantics quite a bit. Even the use of ordinary variables as flags for event synchronization (Figure 9.4[a]) is no longer guaranteed to work since the write of `flag` may become visible to other processors before the write of `A`. This model must therefore demonstrate a substantial performance benefit to be attractive.

The only additional instruction we need over TSO is one that enforces write-to-write ordering in a process's program order. In Sun Sparc V9, this can be achieved by using a MEMBAR instruction with the write-to-write flavor turned on (the earlier Sparc V8 specification provided a special instruction called store barrier or STBAR to achieve this effect). For example, to achieve the intuitive semantics, we would insert such an instruction between the writes of `A` and `flag`.

Relaxing All Program Orders: Weak Ordering and Release Consistency

In this final class of specifications, no program orders are guaranteed by default (other than data and control dependences within a process, of course). The benefit is that multiple read requests can also be outstanding at the same time, can be bypassed by later writes in program order, and can themselves complete out of order, thus allowing us to hide read latency. These models are particularly well matched to dynamically scheduled processors whose implementation indeed allows them to proceed past read misses to other memory references. They are also the only models

that allow many of the key reorderings and elimination of accesses as done by compiler optimizations. Given the importance of these compiler optimizations for node performance, as well as their transparency to the programmer, these may in fact be the only reasonable high-performance memory models for multiprocessors (unless compiler analysis of potential violations of consistency makes dramatic advances). Prominent models in this group are weak ordering (WO) (Dubois, Scheurich, and Briggs 1986; Dubois and Scheurich 1990), release consistency (RC) (Gharachorloo 1990), Digital Alpha (Sites 1992), Sparc V9 relaxed memory ordering (RMO) (Weaver and Germond 1994), and IBM PowerPC (May et al. 1994; Corella, Stone, and Barton 1993). WO is the seminal model, RC is an extension of WO supported by the Stanford DASH prototype (Lenoski et al. 1993), and the last three are supported in commercial architectures. Let us discuss these models individually and see how they deal with the problem of providing intuitive semantics despite all the reordering; for instance, how they deal with the flag synchronization example.

Weak Ordering The motivation behind the weak ordering model (also known as the *weak consistency model*) is quite simple. Most parallel programs use synchronization operations to coordinate accesses to data when this is necessary. Between synchronization operations, they do not rely on the order of accesses being preserved. Two examples are shown in Figure 9.5. The left fragment (a) uses a lock-unlock pair to delineate a critical section inside which the head of a linked list is updated (Adve and Gharachorloo 1996). The right fragment (b) uses flags to control access to variables participating in a producer-consumer interaction (e.g., A and D are produced by P₁ and consumed by P₂). The key in the flag example is to think of the accesses to the flag variables as synchronization operations since that is indeed the purpose they are serving. If we do this, then in both situations the intuitive semantics are not violated by any program reorderings that happen between synchronization operations or accesses (i.e., in the critical section in segment [a] and in the four statements after the while loop in segment [b]) as long as synchronization operations are not reordered with respect to data accesses or one another. Based on these observations, weak ordering relaxes all program orders for nonsynchronization memory operations by default and guarantees that orderings will be maintained only at synchronization operations that can be identified by the system as such. Further orderings can be enforced by adding synchronization operations or labeling some memory operations as synchronization. How appropriate operations are identified as synchronization operations is discussed in Section 9.1.2.

The left side of Figure 9.6 illustrates the reorderings of memory operations allowed by weak ordering. Each block with a set of reads/writes represents a contiguous run of nonsynchronization memory operations from a processor. Synchronization operations are shown separately. Sufficient conditions to ensure a WO system are as follows. Before a synchronization operation is issued, the processor waits for all previous operations in program order (both reads and writes) to have completed. Similarly, memory accesses that follow the synchronization operation are not issued until the synchronization operation completes. Read, write, and read-modify-write operations that are not labeled as synchronization can be arbitrarily reordered

P_1, P_2, \dots, P_n	P_1	P_2
...		
Lock(TaskQ)		
newTask→next = Head;	A = 1;	x = A;
if (Head != NULL)	u = B;	y = D;
Head→prev = newTask;	v = C;	B = 3;
Head = newTask;	D = B * C;	C = D / B;
UnLock(TaskQ)	flag2 = 0;	flag1 = 0;
...	flag1 = 1;	flag2 = 1;
	goto TOP;	goto TOP;
(a)		(b)

FIGURE 9.5 Use of synchronization operations to coordinate access to ordinary shared data variables. The synchronization may be through the use of explicit lock, unlock, and barrier operations or through the use of flag variables for point-to-point events.

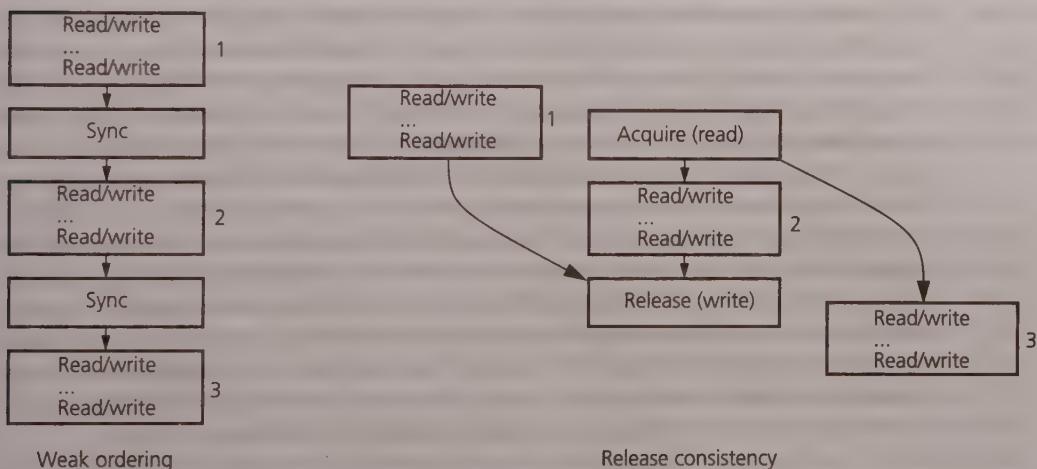


FIGURE 9.6 Comparison of the weak ordering and release consistency models. The operations in block 1 precede the first synchronization operation, which is an acquire, in program order. Block 2 occurs between the two synchronization operations, and block 3 follows the second synchronization operation, which is a release.

between synchronization operations. Especially when synchronization operations are infrequent, as in many parallel programs, WO typically provides considerable reordering freedom to the hardware and compiler.

Release Consistency Release consistency observes that weak ordering does not go far enough. It extends the weak ordering model by distinguishing among types of synchronization operations and exploiting their semantics. In particular, it divides

synchronization operations into acquires and releases. An *acquire* is a read operation (it can also be a read-modify-write) that is performed to gain access to a set of operations or variables. Examples include the Lock (TaskQ) operation in part (a) of Figure 9.5, and the accesses to flag variables within the while conditions in part (b). A *release* is a write operation (or a read-modify-write) that grants permission to another processor to gain access to some operations or variables. Examples include the UnLock (TaskQ) operation in part (a) of Figure 9.5, and the statements setting the flag variables to 1 in part (b).

The separation into acquire and release operations can be used to further relax ordering constraints, as shown in Figure 9.6. The purpose of an acquire is to delay memory accesses that follow the acquire operation until the acquire completes. It has nothing to do with accesses that precede it in program order (accesses in block 1), so there is no reason to wait for those accesses to complete before the acquire can be issued or completed. That is, the acquire itself can be reordered with respect to previous accesses. Similarly, the purpose of a release operation is to grant access to the new values of data that are modified before it in program order. It has nothing to do with accesses that follow it in program order (accesses in block 3), so these need not be delayed until the release has completed. However, we must wait for accesses in block 1 to complete as well before the release is visible to other processors (since they precede the release too, and we do not know exactly which variables are associated with the release or are “protected” by the release³), and similarly we must wait for the acquire to complete before the operations in block 3 can be performed. Besides these constraints, the memory operations in blocks 1, 2, and 3 can be overlapped and reordered. Thus, the sufficient conditions for providing an RC interface are as follows: before an operation labeled as a release is issued, the processor waits until all previous operations in program order have completed; operations that follow an acquire operation in program order are not issued until that acquire operation completes. These are sufficient conditions, and we examine more aggressive implementations when we discuss alternative approaches to a shared address space that rely on relaxed consistency models for good performance. Note that the write propagation clause of coherence, as defined in Chapter 5, is not guaranteed unless enough synchronization is present, nor is write serialization, a point we return to in Section 9.3.3.

Digital Alpha, Sparc V9 RMO, and IBM PowerPC memory models While the WO and RC models are specified in terms of using labeled synchronization operations to enforce orders, they do not take a position on the exact operations (instructions) that must be used. The memory models of some commercial microprocessors provide no ordering guarantees by default (for memory or synchronization operations) but provide specific hardware instructions called *memory barriers* or *fences* that can

3. It is possible that the release is intended to also grant access to the results of operations outside of (before) the operations controlled by the preceding acquire. The exact association of variables with synchronization accesses is very difficult to exploit at the hardware level. Software implementations of relaxed models, however, do exploit such optimizations, as we shall see in Section 9.3.

be used to enforce orderings. To implement WO or RC with these microprocessors, operations that the WO or RC program labels as synchronizations (or acquires or releases) cause the compiler to insert the appropriate special instructions, or the programmer can insert these instructions directly.

The Alpha architecture (Sites 1992), for example, supports two kinds of fence instructions: the memory barrier (MB) and the write memory barrier (WMB). The MB fence is like a synchronization operation in WO: it waits for all previously issued memory accesses to complete before issuing any new accesses. It does not have flavors like the Sparc MEMBAR instructions. The WMB fence imposes program order only between writes (it is like the STBAR in PSO). Thus, a read issued after a WMB can still bypass (complete before) a write access issued before the WMB, but a write access issued after the WMB cannot. The Sparc V9 relaxed memory order (RMO) (Weaver and Germond 1994) provides a fence or MEMBAR instruction with four flavor bits associated with it, as discussed earlier. Each bit indicates a particular type of ordering to be enforced between previous and following load-store operations (the four possibilities are read-to-read, read-to-write, write-to-read, and write-to-write orderings). Any combinations of these bits can be set, offering a variety of ordering choices. Finally, the IBM PowerPC model (May et al. 1994; Corella, Stone, and Barton 1993) provides only a single fence instruction, called SYNC, that is equivalent to Alpha's MB fence. It differs from the Alpha and RMO models in that the writes are not atomic, as in the processor consistency (PC) model. The model envisioned by PowerPC is WO, to be synthesized by putting SYNC instructions before and after every synchronization operation. We see how different models can be synthesized with these primitives in Exercise 9.13.

The prominent specifications just discussed are summarized in Table 9.1 (Adve and Gharachorloo 1996).⁴ They have different performance implications and require different kinds of annotations to ensure orderings. It is worth noting again that if program order is defined as seen by the programmer, then only the models that allow both read and write operations to be reordered within sections of code (WO, RC, Alpha, RMO, and PowerPC) allow the flexibility needed by many important compiler optimizations. This may change if substantial improvements are made in compiler analysis to determine what reorderings are possible given a consistency model. The difficulty of reasoning with allowable reorderings and inserting order-enforcing instructions should be clear, as should the portability problem of the specifications. For example, a program with enough memory barriers to work "correctly" (produce intuitive or sequentially consistent executions) on a TSO system will not necessarily work "correctly" when run on an RMO system: it will need more special

4. The relaxation "read own write early" in the table is relevant to both program order and write atomicity. The processor is allowed to read its own previous write before the write is serialized with respect to other writes to the same location (i.e., before the write completes). A common hardware optimization that relies on this relaxation is the processor reading the value of a variable from its own write buffer. This relaxation can be used with almost all models without violating their semantics. It can even be used with SC as long as other program order and atomicity requirements are maintained.

Table 9.1 Characteristics of Various System Specifications

Model	Write-to-Read Reorder	Write-to-Write Reorder	Read-to-Read / Write Reorder	Read Other's Write Early	Read Own Write Early	Operations for Ordering
SC					yes	
TSO	yes				yes	MEMBAR, RMW
PC	yes			yes	yes	MEMBAR, RMW
PSO	yes	yes			yes	STBAR, RMW
WO	yes	yes	yes		yes	SYNC
RC	yes	yes	yes	yes	yes	REL, ACQ, RMW
RMO	yes	yes	yes		yes	various MEMBARS
Alpha	yes	yes	yes		yes	MB, WMB
PowerPC	yes	yes	yes	yes	yes	SYNC

A "yes" in the appropriate column indicates that those orders can be violated by that system-centric model. "Read other's write early" means that a processor is allowed to see the result of a write operation before that write operation has completed globally.

operations. Let us therefore examine higher-level interfaces that are more convenient for programmers and portable to the different systems, safely exploiting the performance benefits and reorderings that each system affords.

9.1.2 The Programmer's Interface

The programming interfaces are inspired by the WO and RC models, in that they assume that program orders do not have to be maintained at all between synchronization operations. The idea is for the program to ensure that all synchronization operations, including point-to-point event synchronization using flags, are explicitly labeled or identified as such. This is the programmer's part of the contract. The compiler or run-time library translates these synchronization operations into the appropriate order-preserving operations (memory barriers or fences) called for by the system specification. Then the system (compiler plus hardware) guarantees sequentially consistent executions even though it may reorder operations between synchronization operations in any way it desires (without violating dependences to a location within a process). This is the system's part of the contract. This contract allows the compiler sufficient flexibility between synchronization points for the reorderings it desires. It also allows the processor to perform as many reorderings as permitted by its memory model or implementation and is therefore portable: if SC executions are guaranteed even with the weaker models that allow all reorderings, they surely will be guaranteed on systems that allow fewer reorderings. The consistency model presented at the programmer's interface should be at least as weak (relaxed) as that at the hardware interface but need not be the same.

Programs that label all synchronization events are called *synchronized programs*. Formal models for specifying synchronized programs have been developed, namely, the *data-race-free* models influenced by weak ordering (Adve and Hill 1990a) and the *properly labeled* model (Gharachorloo et al. 1992) influenced by release consistency (Gharachorloo et al. 1990). Interested readers can obtain more details from these references (the differences between the models are minor). The basic question the programmer must address is which operations to label as synchronization operations. This is, of course, already done in the majority of cases when explicit, system-specified programming primitives such as locks and barriers are used. These are usually also easy to distinguish as acquire or release, for memory models such as RC that can take advantage of this distinction; for example, a lock is an acquire and an unlock is a release, and a barrier contains both since arrival at a barrier is a release (indicating completion of previous accesses) whereas leaving it is an acquire (obtaining permission for the new set of accesses). The real question is how to determine which memory operations on ordinary variables (such as our flag variables) should be labeled as synchronization operations. Often, programmers can identify these easily since they know when they are using this event synchronization idiom. The following definitions describe a more general method for identifying synchronization events when all else fails.

- *Conflicting operations*: Two memory operations from different processes are said to *conflict* if they access the same memory location and at least one of them is a write.
- *Competing operations*: These are a subset of the conflicting operations. Two conflicting memory operations (from different processes) are said to be *competing* if it is possible for them to appear next to each other in a sequentially consistent total order (execution), that is, to appear one immediately following the other in such an order with no intervening memory operations on shared data between them.
- *Synchronized program*: A parallel program is *synchronized* if all competing memory operations have been labeled as synchronization operations (perhaps differentiated into acquire and release by labeling the read operations as acquires and the write operations as releases).

The fact that “competing” means competing under any possible SC interleaving is an important aspect of the programming interface. Even though a system uses a relaxed consistency model, the reasoning about where annotations are needed can itself be done while assuming an intuitive, SC execution model, shielding the programmer from reasoning directly in terms of reorderings. Of course, the programmer’s task would be a lot simpler if the compiler could automatically determine what operations are conflicting or competing. However, this problem is similar to that of determining what reorderings are possible under a consistency model, and since the known analysis techniques are expensive and/or conservative (Shasha and Snir 1988; Krishnamurthy and Yelik 1994, 1995), the job is almost always left to the programmer.

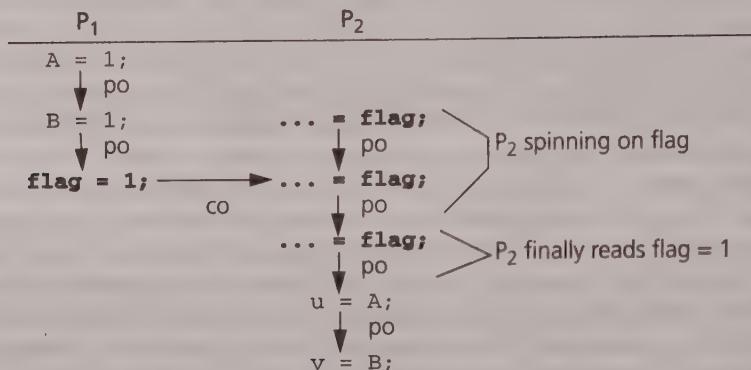


FIGURE 9.7 An example code sequence illustrating program and conflict orders. The arcs labeled “po” show program order, and the one labeled “co” shows conflict order. Notice that between the write to A by P_1 and the read to A by P_2 is a chain of accesses formed by program order and conflict order arcs. This will be true in all SC executions of the program. Such chains that have at least one program order arc imply that the accesses are noncompeting; they will not be present for accesses to the variable $flag$ between which there is only a conflict order arc. Boldface indicates that the accesses to the $flag$ variable are the important ones for ordering and are in fact being used to orchestrate event synchronization.

Consider the example in Figure 9.7, repeated from Figure 9.3. The accesses to the variable $flag$ are competing operations by the preceding definition. What this really means is that on a multiprocessor they may execute simultaneously on their respective processors, unordered with respect to each other, so we have no guarantee about which executes or appears to complete first. Thus, they are also said to constitute *data races*. In contrast, the accesses to variable A (and to B) by P_1 and P_2 are conflicting operations, but they are necessarily separated in any SC interleaving and hence ordered by an intervening write to the variable $flag$ by P_1 and a corresponding read of $flag$ by P_2 . Thus, they are not competing accesses and do not have to be labeled as synchronization operations.

To be a little more formal, given a particular SC execution order, the *conflict order* is the order in which the conflicting operations to a location occur (from any process). In addition, we have the *program order* for each process. Figure 9.7 shows arcs corresponding to the program orders and conflict order for a sample execution of our code fragment. Two accesses are noncompeting if under all possible SC executions (interleavings) a chain of other references always exists between them, such that at least one link in the chain is formed by a program order rather than conflict order arc. Otherwise, they are competing. The complete formalism can be found in (Gharachorloo 1995).

Of course, the definition of synchronized programs allows a programmer to conservatively label more operations than necessary as synchronization operations without compromising correctness. In the extreme, labeling all memory operations as

synchronization operations always yields a synchronized program. This extreme case will, of course, deny us the performance benefits that the system might otherwise provide by reordering nonsynchronization operations and will, on most systems, yield much worse performance than straightforward SC implementations due to the overhead of the order-preserving instructions that will be inserted. The goal is to label only competing operations as synchronization operations.

In specific circumstances, we may want to allow data races in the program and may, therefore, decide not to label some competing accesses as synchronization operations. Now we are no longer guaranteed SC semantics, but we may know through application knowledge that the competing operations are not being used as synchronization operations and that we do not need such strong ordering guarantees in certain sections of code. An example in Chapter 2 is the use of the asynchronous equation solver rather than red-black ordering. There is no synchronization between barriers or grid sweeps, so within a sweep the read and write accesses to the border elements of a partition are competing accesses. If they are not labeled, the program will not satisfy SC semantics on a system that allows access reorderings, but this is okay since the solver repeats the sweeps until convergence: even if the processes sometimes read old values in a sweep and sometimes new in an unpredictable manner, they will read updated values in the next sweep (after the barrier) and make progress toward convergence. If we had labeled the competing accesses, we would have compromised access reordering and performance. The number of sweeps to convergence might have been a little smaller, but the cost of each sweep would have been larger.

The last issue related to the programming interface is how the labels for competing accesses are to be specified by the programmer. In many cases, this is quite stylized and already present in the programming language. Some parallel programming languages, for example, High Performance Fortran (High Performance Fortran Forum 1993), allow parallelism to be expressed in only stylized ways from which it is trivial to extract the relevant information. For example, in FORALL loops (loops in which all iterations are independent) only the implicit barrier at the end of the loop needs to be labeled as synchronization: the FORALL specifies that there are no data races within the loop body that the system should worry about. In more general programming models, if programmers use a library of synchronization primitives such as LOCK, UNLOCK, and BARRIER, then, even if these primitives are implemented using ordinary memory operations, the code that implements them can be labeled by the designer of the library; the programmer needn't do anything special. Finally, if the application programmer wants to add further labels at memory operations—for example, at flag variable accesses or to preserve some other orders, as in the examples of Figure 9.4—we need support from a programming language or library. A programming language could provide an attribute for variable declarations that indicates that all references to a variable are synchronization accesses; or there could be annotations at the statement level, indicating that a particular access is to be labeled as a synchronization operation. This tells the compiler to constrain its reordering across those points, and the compiler in turn translates these references to the appropriate order-preserving mechanisms for the processor.

9.1.3 The Translation Mechanism

For most microprocessors, translating labels to order-preserving mechanisms amounts to inserting a suitable memory barrier or fence instruction before and/or after each operation that is labeled as a synchronization (or acquire or release). It would save instructions if we could have flavor bits associated with individual loads and stores themselves, indicating what orderings to enforce and thus avoiding extra instructions; but since the operations are usually infrequent, making such core changes to the instruction set is not the direction that most microprocessors have taken so far.

9.1.4 Consistency Models in Real Multiprocessor Systems

With the large growth in sales of multiprocessors, modern microprocessors are designed so that they can be seamlessly integrated into these machines. As a result, microprocessor vendors expend substantial effort defining and precisely specifying the memory model presented at the hardware/software interface. While sequential consistency remains the best model for programmers to reason with, many vendors allow orders to be relaxed for performance reasons. Some vendors like Silicon Graphics (in the MIPS R10000 processor) continue to support SC even in multiple-issue, dynamically scheduled processors by allowing out-of-order issue and execution of operations but not out-of-order completion or visibility. This allows substantial overlapping of memory operations by the dynamically scheduled processor and does not satisfy the sufficient conditions for SC, but it forces operations to complete in program order. The Intel Pentium family supports a processor consistency model, so reads can complete before previous writes in program order, and many microprocessors from Sun Microsystems support TSO, which allows the same reorderings. Many other vendors have moved to models that allow all orders to be relaxed (e.g., Digital Alpha and IBM PowerPC) and that provide memory barriers to enforce orderings where necessary.

At the hardware interface, multiprocessors usually follow the consistency model exported by the microprocessors they use since this is the easiest thing to do. For example, we saw that the NUMA-Q hardware exports the processor consistency model of its Pentium Pro processors. In particular, on a write, the ownership and perhaps data is obtained before the invalidations begin; the processor is allowed to complete its write and go on as soon as the ownership is received, and the SCLIC communication assist takes care of the sequence of invalidations and acknowledgments. It is also possible for the communication assist to alter the model, within limits. We have seen an example in the Origin2000, where preserving the processor's SC model requires that the assist (Hub) only reply to the processor on a write once the exclusive reply and all invalidation acknowledgments have been received (called *delayed-exclusive* replies). The dynamically scheduled processor can then retire its write from the instruction lookahead buffer and allow subsequent operations to retire and complete as well. If the Hub replies as soon as the exclusive reply is received and before invalidation acknowledgments are received (called *eager*-

exclusive replies), then the write will retire and subsequent operations (including writes) may become visible and complete before the write is actually completed, so the consistency model is more relaxed. Essentially, the Hub fools the processor about the completion of the write.

Having the assist fool the processor can enhance performance but increase design complexity, as in the case of eager-exclusive replies. The handling of invalidation acknowledgments must now be done asynchronously by the assist through tracking buffers in its processor interface even after the reply has been passed to the processor, in accordance with the desired relaxed consistency model. There are also questions about whether subsequent accesses to a block from other nodes, forwarded to this processor from the home, should be serviced while invalidations for a write on that block are still outstanding (see Exercise 9.3) and about what happens if the processor has to write that block back to memory due to replacement while invalidations are still outstanding in the assist. In the latter case, either the write back has to be buffered by the assist and delayed until all invalidations have been acknowledged, or the protocol must be extended so a later access to the written-back block is not satisfied by the home until the acknowledgments are received by the requestor. The extra complexity and the limited performance improvement perceived by the Origin2000 designers led them to persist with a sequential consistency model and delayed-exclusive replies.

On the compiler side, the picture for memory consistency models is currently not so well defined, complicating matters for programmers. It does not do a programmer much good for the processor to support sequential consistency or processor consistency if the compiler reorders accesses as it pleases before they even get to the processor (as uniprocessor compilers do). Microprocessor memory models are defined at the hardware interface; they tend to be concerned with program order as presented to the processor and assume that a separate arrangement will be made with the compiler. As we have discussed, exporting intermediate models such as TSO, PC, and PSO up to the programmer's interface does not allow the compiler enough flexibility for reordering. Programmers might assume that in practice the compiler will not reorder or eliminate operations in a manner that would violate the consistency model, for example, since most compiler reorderings of memory operations tend to focus on loops; but this is a very dangerous assumption, and sometimes the orders we rely upon indeed occur in loops. To really use these models at the programmer's interface, uniprocessor compilers would have to be modified to follow their restriction on reordering, compromising performance significantly. These intermediate models are supported at the hardware interface but are not very appropriate for the programmer's interface. (Of course, the point would be moot if the compiler could detect competing operations, in which case it could export the strongest SC model to the programmer and yet itself perform the reorderings of the most relaxed models we discuss.)

More relaxed models like Alpha, RMO, PowerPC, WO, RC, and synchronized programs can be used even at the programmer's interface because they allow the compiler the flexibility it needs. The mechanisms used to communicate ordering constraints at the programmer's interface must be heeded not only by the processor

but also by the compiler, and compilers for multiprocessors are beginning to do so (see Section 9.5). Beneath a relaxed model at the programmer's interface, the hardware interface can use the same or stronger ordering model, as we saw in the context of synchronized programs. However, significant motivation exists to use relaxed models even at the processor interface in this case to realize the performance potential. As we move now to discussing alternative approaches to supporting a shared address space with coherent replication of data, we see that relaxed consistency models can be critical to performance when we want to support coherence at larger granularities than cache blocks. We also see that the consistency model can be relaxed even beyond release consistency. (Can you think how?)

9.2 OVERCOMING CAPACITY LIMITATIONS

In a CC-NUMA system like the SGI Origin2000, a processor cache replicates remotely allocated data directly upon reference, without it being replicated in the local main memory first. On a cache miss, the assist determines from the physical address whether to look up local memory and directory state or to send the request directly to a remote home node. The granularity of communication, of coherence, and of allocation in the replication store (cache) is a cache block. As discussed earlier, a problem with these systems is that the capacity for local replication is limited to the hardware cache. If a remotely installed block is replaced from the cache, it must be fetched from remote memory if it is needed again, incurring artificial communication. The goal of the systems discussed in this section is to overcome the replication capacity problem while still providing coherence in hardware at the granularity of cache blocks.

9.2.1 Tertiary Caches

One way to achieve this goal is to use a large but slower remote access cache, as in the Sequent NUMA-Q and Convex Exemplar (Convex Computer Corporation 1993; Thekkath et al. 1997). This may be needed for functionality anyway if the nodes of the machine are themselves small-scale multiprocessors, in order to present a single per-node cache to the protocol across nodes. This remote access cache keeps track of remotely allocated blocks that are currently in the local processor caches and can simply be made larger for performance. Then it will also hold replicated remote blocks that have been replaced from local processor caches. In NUMA-Q, this DRAM remote cache is at least 32 MB whereas the sum of the four lowest-level processor caches in a node is only 2–4 MB. A similar method, which is sometimes called the *tertiary cache* approach, is to take a fixed portion of the local main memory and manage it like a remote cache, requiring additional hardware for per-block tags and state.

These approaches replicate data in main memory at fine grain, but they do not automatically migrate or change the home of a block to the node that incurs cache misses most often on that block. Space is always allocated for the block in main memory at the original home. Thus, if data were not distributed appropriately in

main memory by the application, then in the tertiary cache approach, even if only one processor ever accesses a given memory block (no need for multiple copies or replication), the system may end up wasting half of its available main memory: there are two copies of each block in main memory, one at the home and one in the tertiary cache, but only one is ever used. In addition, a statically established tertiary cache is wasteful if its replication capacity is not needed for performance. The cache-only memory architecture or COMA approach to increasing replication capacity is to treat all of local memory as a hardware-controlled cache. This approach, which achieves both replication and migration, does not have these problems and is discussed in more depth next. In all these cases, replication and coherence are managed at a fine granularity, though this does not necessarily have to be the same as the block size in the processor caches. Only data that is already in the local memory, remote cache, or tertiary cache is brought into the processor cache hierarchy; since the data is kept coherent across nodes at the outer level, processor caches themselves do not have to be kept coherent across nodes through a separate internode protocol but must only be kept coherent with the local memory or remote/tertiary cache.

9.2.2 Cache-Only Memory Architectures (COMA)

In COMA machines, every fine-grained memory block in the entire main memory has a hardware tag associated with it. There is no fixed node where space is always guaranteed to be allocated for a memory block. Rather, data dynamically moves to and is replicated in the main memories of the nodes that access and hence “attract” it; these main memories, organized as caches, are therefore called *attraction memories*. When a remote block is accessed, it is replicated in attraction memory as well as being brought into the processor cache and is kept coherent by hardware. Migration of a block is achieved through replacement or invalidation in the attraction memory: if block x originally resides in node A’s main (attraction) memory, then when node B reads it, B will obtain a copy (replication); if the copy in A’s memory is later invalidated or replaced by another block that A references, then the only copy of that block left is now in B’s attraction memory. Thus, we do not have the problem of wasted original copies that we potentially had with the tertiary cache approach, and both data migration and space management are demand driven. Since a data block may reside in any attraction memory and move transparently from one to the other, the location of data is decoupled from its physical address. Automatic data migration also has substantial advantages for multiprogrammed workloads in which the operating system may decide to migrate processes among nodes at any time, although in this case software migration of pages may be successful too.

Hardware/Software Trade-Offs

Like the other approaches, the COMA approach introduces clear hardware/software trade-offs. By overcoming the cache capacity limitations of the pure CC-NUMA approach, the goal is to free parallel software from worrying about data distribution

in main memory. The programmer can view the machine as if it had a centralized main memory and worry only about inherent communication and false sharing (of course, cold misses may still be satisfied remotely if data is not distributed well). Although this makes the task of software writers much easier, COMA machines require a lot more hardware support than pure CC-NUMA machines since they implement main memory as a hardware cache. This includes per-block tags and state in main memory as well as the necessary comparators. There is also the extra memory overhead needed for replication in the attraction memories, which we discuss later in this section. Finally, the coherence protocol for attraction memories is more complicated than what we saw for processor caches. There are two reasons for this, both having to do with the fact that data moves dynamically to where it is referenced and does not have a fixed “home” to back it up. First, the location of the data must be determined upon an attraction memory miss, since it is no longer bound to the physical address. Second, with no space necessarily reserved for the block at the home, it is important to ensure that the last or only copy of a block is not lost from the system by being replaced from its attraction memory. This extra complexity is not a problem in the tertiary cache approach.

Performance Trade-Offs

Performance has its own interesting set of trade-offs. Although the number of remote accesses due to artifactual communication is reduced, COMA machines tend to increase the latency of accesses that do need to be satisfied remotely, including cold, true sharing, and false sharing misses. The reason is that even a cache miss that will not be satisfied in the local attraction memory needs to first look up that memory to see if it has a local copy of the block. Also, the attraction memory access itself is a little more expensive than a standard DRAM access because the attraction memory is usually implemented to be set associative, so a tag selection may be in the critical path.

In terms of performance, then, COMA is most likely to be beneficial for applications that have high capacity miss rates in the processor cache (large working sets) to data that is not allocated locally to begin with and most harmful to applications where performance is dominated by coherence misses. The advantages are also greatest when access patterns are unpredictable or when accesses from different processes are spatially interleaved at fine grain, so data placement, replication, or migration at page granularity in software would be difficult on CC-NUMA machines. For example, COMA machines are likely to be more advantageous when a two-dimensional array representation is used for a near-neighbor grid computation than when a four-dimensional array representation is used because, in the latter case, appropriate data distribution at page granularity through the OS is not difficult in software; in fact, the higher cost of communication may make COMA machines perform worse than pure CC-NUMA when four-dimensional arrays are used with proper data distribution. Figure 9.8 summarizes the trade-offs in terms of application characteristics. Let us briefly look at some design options for COMA protocols

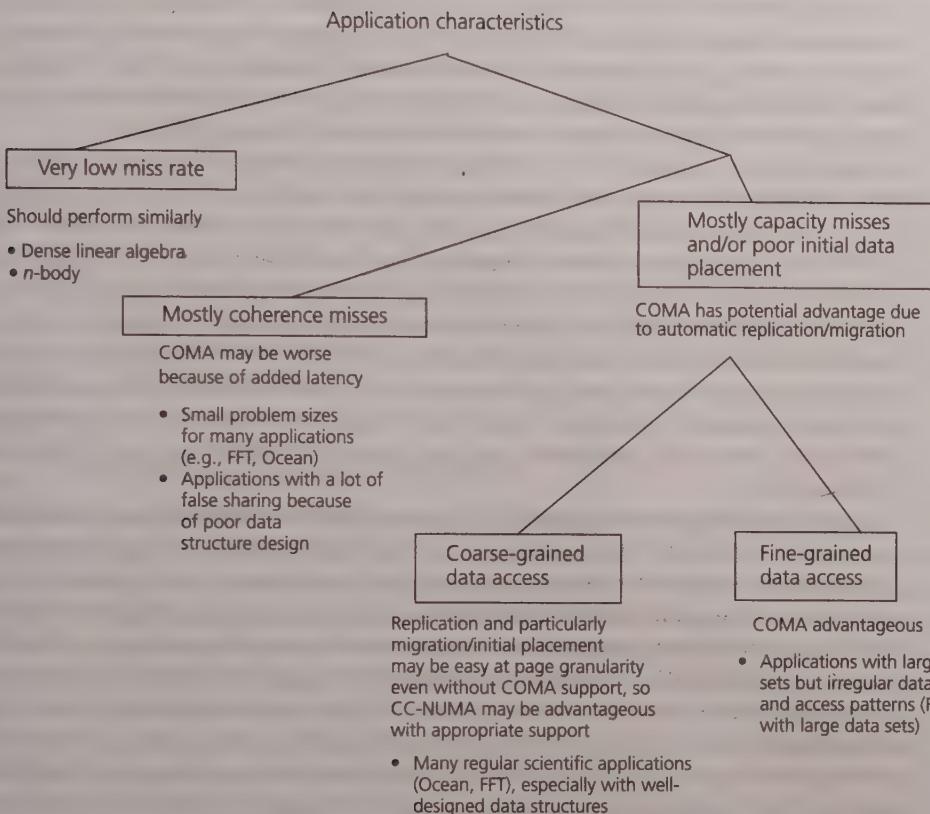


FIGURE 9.8 Performance trade-offs between COMA and CC-NUMA architectures. The application characteristics are in boxes. Below each box is the expected performance comparison of COMA and CC-NUMA systems built with similar technology for that set of characteristics followed by a list of example application areas.

and how they might solve the protocol problems of finding the data on a miss and not losing the last copy of a block.

Design Options: Flat versus Hierarchical Approaches

COMA machines can be built with hierarchical or with flat directory schemes or even with hierarchical snooping (see Section 8.10.2). Hierarchical directory-based COMA was used in the Data Diffusion Machine prototype (Hagersten, Landin, and Haridi 1992; Hagersten 1992), and hierarchical-snooping COMA was used in commercial systems from Kendall Square Research (Frank, Burkhardt, and Rothnie 1993). In these *hierarchical* COMA schemes, data is found on a miss by traversing the hierarchy, just as in non-COMA hierarchical protocols discussed in Chapter 8. The difference is that whereas in non-COMA machines there is a fixed home node

for a memory block in a processing node, here there is not. When a reference misses in the local attraction memory, it proceeds up the hierarchy until a node is found that indicates the presence of the block in its subtree in the appropriate state. The request then proceeds down the hierarchy to the appropriate processing node (which is at a leaf), guided by directory lookups or snooping at each node along the way.

In flat COMA schemes, there is still no home for a memory block in the sense of a reserved location for the data; however, there is a fixed home where just the directory information can be found (Stenstrom, Joe, and Gupta 1992; Joe 1995). This fixed home is determined from either the physical address (as in CC-NUMA) or a global identifier obtained from the physical address. The (static) location of the directory information is also decoupled from the (dynamically changing) location of the actual data. A miss in the local attraction memory goes to the home to look up the directory information, and the directory keeps track of where copies actually are in either a memory-based or cache-based way. The trade-offs for hierarchical directories versus flat directories are very similar to those without COMA (see Section 8.10.2).

Let us see how hierarchical and flat schemes can solve the last copy replacement problem. If the block being replaced from an attraction memory is in shared state, then if we are certain that there is another copy of the block in the system, we can safely discard the replaced block. But for a block that is in an exclusive state or is the last copy in the system, we must ensure that it finds a place in some other attraction memory and is not thrown away. In the hierarchical case, for a block in shared state we simply have to go up the hierarchy until we find a node that indicates that a copy of the block exists somewhere in its subtree. Then we can discard the replaced block as long as we have updated the state information on the path along the way. For a replaced block in an exclusive state, we go up the hierarchy until we find a node that has a block in invalid or shared state somewhere in its subtree, which this block can replace. If the replaceable block is in shared rather than invalid state, then its replacement will require the same procedure to be followed; an invalid block is the easier case.

In a flat COMA, more machinery is required for the last copy problem since there is no built-in mechanism to search for available space. One mechanism is to label one copy of a memory block as the *master* copy and to ensure that the master copy is not dropped upon replacement. A new cache state called *master* is added in the attraction memory. When data is initially allocated, every block is a master copy. Later, a master copy is either an exclusive copy or one of the shared copies. When a shared copy of a block that is not the master copy is replaced, it can be safely dropped (we may, if we like, send a replacement hint to the directory entry at the home). If a master copy is replaced, a replacement message must be sent to the home. The home then chooses another node to send this master copy to in the hope of finding room and sets that node to be the master. If all available blocks in that set of the attraction memory at this new destination are also masters, then the request is sent back to the home, which tries another node and so on. Otherwise, one of the replaceable blocks in the set is replaced and discarded (some optimizations are discussed in [Joe and Hennessy 1994]).

Regardless of whether a hierarchical or flat COMA protocol is used, the initial data set of the application should not fill the entire main or attraction memory, in order to ensure that enough space is available in the system for a replaced last (master) copy to find a new residence. To help find replaceable blocks, the attraction memories should be quite highly associative as well. Not having enough extra (initially unallocated) memory available for replication can cause performance problems for several reasons. First, it makes the COMA nature of the machine less effective in satisfying cache capacity misses locally. Second, it implies that useful replicated blocks are more likely to be replaced to make room for replaced master copies. And third, the traffic generated by replaced last copies can become substantial, which can cause a lot of contention in the system. How much memory should be set aside for replication and how much associativity is needed can be determined empirically (Joe and Hennessy 1994).

Summary: Path of a Read Operation

Consider a flat COMA scheme. A virtual address is first translated to a physical address by the memory management unit. This may cause a page fault and a new mapping to be established, as in a uniprocessor, though in the COMA case the actual data for the page is not loaded into memory. The physical address is used to look up the cache hierarchy. If it hits, the reference is satisfied. If not, then it must look up the local attraction memory. Some bits from the physical address are used to find the relevant set in the attraction memory, and the tag store maintained by the hardware is used to check for a tag match. If the block is found in an appropriate state, the reference is satisfied. If not, then a remote request must be generated, and the request is sent to the home determined from the physical address. The directory at the home determines where to forward the request and whether the data is in shared or exclusive state, and the owner node uses the physical address as an index into its own attraction memory to find and return the data. The directory protocol ensures that states are maintained correctly, as usual.

9.3

REDUCING HARDWARE COST

The last of the major issues discussed in this chapter is hardware cost. Reducing cost often implies moving some functionality from specialized hardware to software that runs on existing or commodity hardware. In this case, the functionality in question is managing replication and coherence. Since it is much easier for software to control these functions in main memory than in the hardware cache, the low-cost approaches tend to provide replication and coherence in main memory, like COMA or tertiary cache systems do. The differences from COMA or tertiary caches are the higher overhead or assist occupancy for communication and, often, the granularity at which replication and coherence are managed.

Consider the hardware cost of a pure CC-NUMA approach. The portion of the communication architecture that is on a node can be divided into four parts: the part

of the assist that checks for access control violations, the per-block tags and state that it uses for this purpose, the part that does the actual protocol processing (including intervening in the processor cache), and the network interface itself. To keep data coherent at cache block granularity in hardware, the access control part needs to see every load or store to shared data that misses in the cache so that it can take the necessary protocol action. Thus, the assist must be able to snoop on the local memory system (as well as issue requests to the local memory system, including the cache, in response to incoming requests from the network).

For coherence to be managed efficiently, each of the other functional components of the assist can benefit greatly from hardware specialization and integration. Determining access faults quickly requires that the tags per block of main memory be located close to the access control part of the assist. The speed with which protocol actions can be invoked and the assist can intervene in the processor cache increases as the assist is integrated closer to the cache. Performing protocol operations quickly demands that the assist be either hardwired or, if programmable (as in the Sequent NUMA-Q), specialized for the types of operations that protocols perform most often (e.g., bit-field extractions and manipulations). Finally, moving small pieces of data quickly between the assist and network interface asks that the network interface be tightly integrated with the assist. Thus, for highest performance, we would like the four parts of the communication assist to be tightly integrated, with as few bus crossings as possible to communicate among them, and the whole assist to be specialized and tightly integrated into the node's memory system.

Early cache-coherent machines accomplished this by integrating a hardwired assist into the cache controller and integrating the network interface tightly into the assist. However, modern processors tend to have even their second-level cache controllers on the processor chip, so it is difficult to integrate the assist into this controller once the processor is built. The SGI Origin therefore integrates its hardwired Hub into the memory controller, the Stanford FLASH integrates its specialized programmable protocol engine into the memory controller, and the Sequent NUMA-Q and Hal S1 attach specialized controllers to the memory bus. By using such specialized, tightly integrated hardware support for cache coherence, these approaches do not leverage inexpensive commodity parts for the communication architecture. They are therefore expensive, more so in design and implementation time than in the amount of actual hardware needed.

Research efforts are attempting to lower this cost with several different approaches. One approach is to perform access control in specialized hardware but delegate much of the other activity to software and commodity hardware. Other approaches perform access control in software as well, thus providing a coherent shared address space abstraction on commodity nodes and networks with no specialized hardware support. Access control is provided either at fine granularity by instrumenting the program code, at page granularity by leveraging the existing virtual memory support, or at the granularity of user-defined objects by using a run-time layer that exports an object-based programming interface.

Let us discuss each of these approaches, which are all currently at the research stage. We cover the page-based approach more thoroughly because it changes the

granularity at which data is allocated, communicated, and kept coherent while still preserving the same transparent programming interface as hardware-coherent systems, because it requires substantially different protocols than we have seen so far, and because it illustrates the mechanisms needed to fully exploit the relaxations afforded by relaxed memory consistency models.

9.3.1 Hardware Access Control with a Decoupled Assist

While specialized hardware support is used for fine-grained access control in this approach, some or all of the other aspects (protocol processing, tags, and network interface) can be decoupled from this specialized hardware and from one another. They can then either use commodity hardware attached to less intrusive parts of the node like the I/O bus or use no extra hardware beyond that on the uniprocessor node. For example, the per-block tags and state can be kept in special fast memory or in regular DRAM, and protocol processing can be done in software either on a separate, inexpensive general-purpose processor or even on the main processor itself. The network interface usually has some specialized support for fine-grained communication to reduce the endpoint overheads. Some possible combinations of how the various functions might be integrated are shown in Figure 9.9.

The problem with the decoupled hardware approach, of course, is that it increases the latency of protocol invocation, protocol processing, and communication since the interaction of the different components with each other and with the node is slower (e.g., it may involve several bus crossings). More critically, the effective occupancy of the decoupled communication assist is much larger than that of a specialized, integrated assist, which can hurt performance substantially for many applications as described in Section 8.7.

9.3.2 Access Control through Code Instrumentation

It is possible to use no additional hardware support over a standard uniprocessor node but perform all the functions needed for fine-grained replication and coherence in main memory in software. The trickiest part of this is fine-grained access control in main memory, for which a standard uniprocessor does not provide support. To accomplish this, individual read and write operations can be instrumented in software by adding instructions to look up per-block tag and state data structures maintained in main memory (Schoinas et al. 1994; Scales, Gharachorloo, and Thekkath 1996). To the extent that cache misses can be predicted, only the reads and writes that miss in the processor cache hierarchy need to be thus instrumented. The necessary protocol processing can be performed on the main processor or on whatever form of communication assist is provided. In fact, such software instrumentation allows us to provide access control and coherence at any granularity, even different granularities for different data structures.

Software instrumentation incurs a run-time cost since it inserts extra instructions into the code to perform the necessary checks. The approaches that have been developed use several tricks to reduce the number of checks and lookups needed (Scales,

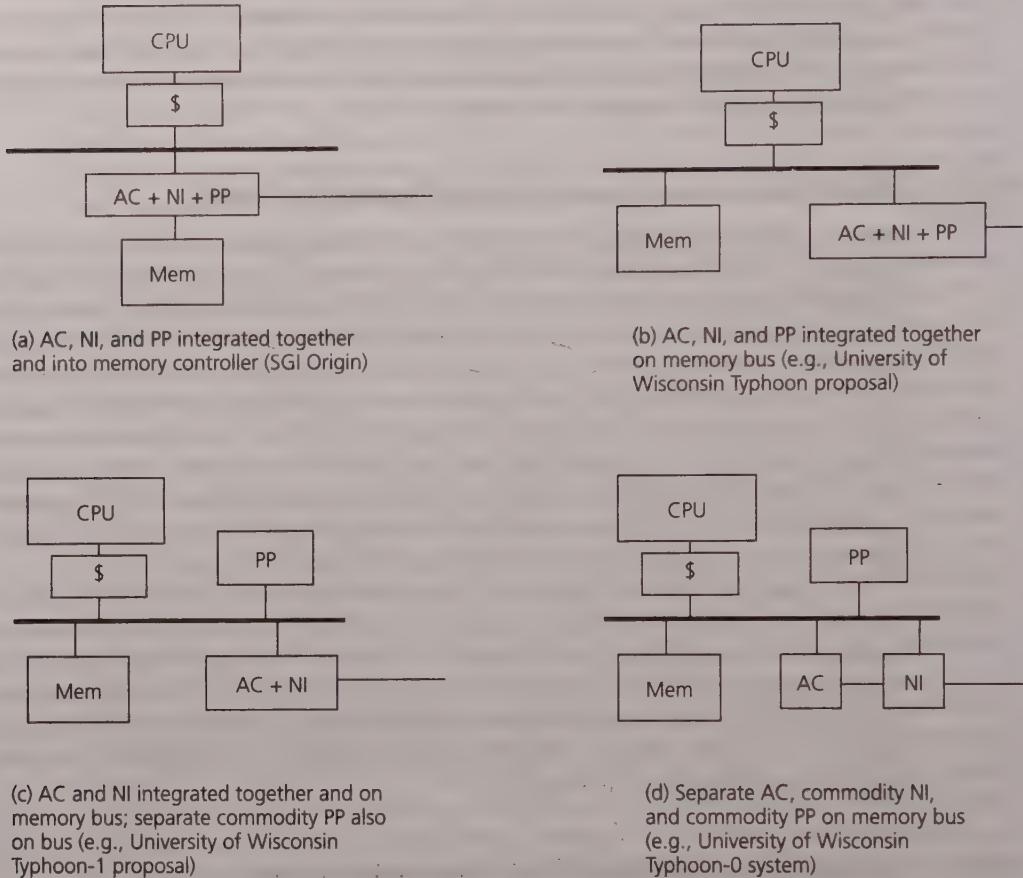


FIGURE 9.9 Some alternatives for reducing cost over a highly integrated and specialized assist. AC is the access control facility, NI is the network interface, and PP is the protocol processing facility (whether hardwired finite state machine or programmable). The highly integrated solution is shown in (a) with alternative, less integrated solutions shown in (b), (c), and (d). As the distance between parts increases, so does the number of expensive bus crossings required for the parts to communicate with one another to process a transaction. In these designs, the commodity PP in (c) and (d) is a complete processor like the main CPU, with its own cache system.

Gharachorloo, and Thekkath 1996), so the cost of access control and protocol invocation may well be competitive with that in the decoupled hardware approach. Protocol processing in software on the main processor also has a significant cost, and while the network interface and interconnects used by such systems usually provide support for fine-grained communication, they are usually commodity based and, hence, less efficient than in tightly coupled multiprocessors.

9.3.3 Page-Based Access Control: Shared Virtual Memory

Another approach to providing access control and coherence with no additional hardware support is to leverage the virtual memory support provided by the memory management units of microprocessors and by the operating system. Memory management units already perform access control in main memory at the granularity of pages (e.g., to detect page faults) and manage main memory as a fully associative cache on the virtual address space. By embedding a coherence protocol in the page fault handlers, we can provide replication and coherence at page granularity and manage the main memories of the nodes as coherent, fully associative caches on a shared virtual address space (Li and Hudak 1989). Access control now requires no special tags, and the assist needn't even see every cache miss. Data enters the local cache only when the corresponding page is already present in local memory. As in the previous two approaches, the processor caches themselves do not have to be kept coherent across nodes by hardware since when a page is invalidated the TLB will not let the processor access its blocks in the cache (care must, of course, be taken to keep processor caches coherent with the local memory and vice versa).

This approach is called *page-based shared virtual memory* or SVM for short. Since the costs can be amortized over a whole page of data, protocol processing is often done on the main processor itself, and we can more easily do without special hardware support for fine-grained communication in the network interface. Thus, there is less need for hardware assistance beyond that available on a standard uniprocessor system.

A very simple form of shared virtual memory coherence is illustrated in Figure 9.10, following an invalidation protocol very similar to those in pure CC-NUMA. A few aspects are worthy of note. First, since the memory management units of different processors manage their main memories independently, the physical address of the page in P_1 's local memory may be completely different from that of the copy in P_0 's local memory, even though the pages have the same (shared) virtual address. There is a shared virtual address space but private physical address spaces. Second, a page fault handler that implements the protocol must be able to perform the three protocol functions discussed in Chapter 8 (finding the source of state information, finding the appropriate copy or copies, and communicating with the copies) before it can set the page's access rights as appropriate and return control to the application process. A directory mechanism can be used for this—every page may have a home, determined by its virtual address, and a directory entry maintained at the home—though high-performance SVM protocols tend to be more complex, as we shall see.

The problems with page-based shared virtual memory are the high overheads of protocol invocation and processing and the large granularity of coherence and communication. The former is expensive because most of the work is done in software on a general-purpose uniprocessor. Page faults take time to cause an interrupt or trap and to switch into the operating system and invoke a handler; the protocol processing itself is done in software, and the messages sent to other processors use the underlying message-passing mechanisms that are expensive, especially with commodity nodes and interconnects. On a representative SVM system in 1998, the

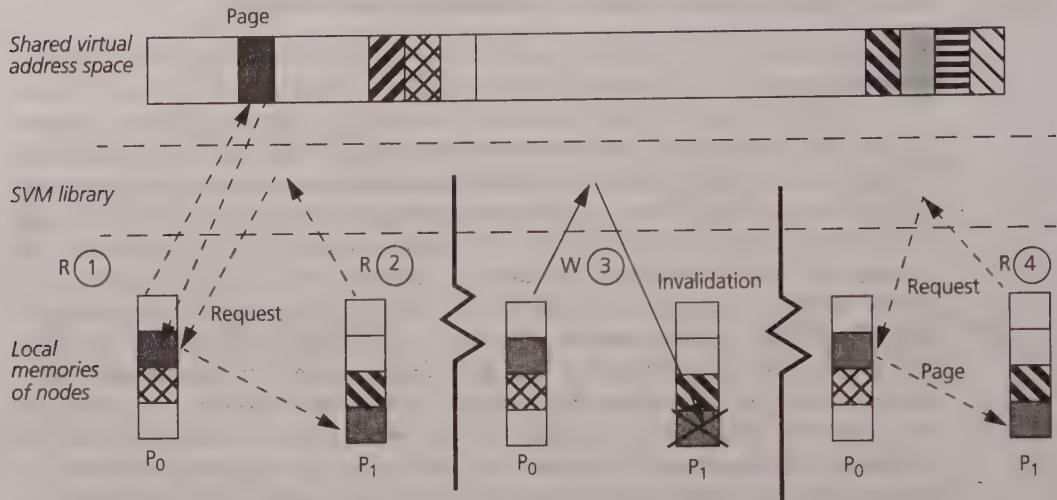


FIGURE 9.10 Illustration of simple shared virtual memory. At the beginning, no node has a copy of the stippled shared virtual page with which we are concerned. Events occur in the order 1, 2, 3, 4. Read 1 incurs a page fault during address translation and fetches a copy of the page to P₀ (presumably from disk). Read 2, shown in the same frame, incurs a page fault and fetches a read-only copy to P₁ (from P₀). This is the same virtual page but is at two different physical addresses in the two memories. Write 3 incurs a page fault (write to a read-only page), and the SVM library, implemented in the page fault handlers, determines that P₁ has a copy and causes it to be invalidated. P₀ now obtains read-write access to the page, which is like the modified or dirty state. When Read 4 by P₁ tries to read a location on the invalid page, it incurs a page fault and fetches a new copy from P₀ through the SVM library.

round-trip cost of satisfying a remote page fault ranges from a few hundred microseconds with aggressive system software support to over a millisecond. This should be compared with less than a microsecond needed for a read miss on aggressive hardware-coherent systems. In addition, since protocol processing is typically done on the main processor (to avoid additional hardware support), even incoming requests interrupt the processor, pollute the cache, and slow down the currently running application thread (which may have nothing to do with that request).

The large granularity of communication and coherence is problematic for two reasons. First, if spatial locality is not very good, it causes a lot of fragmentation in communication and hence useless data transfer (only a word is needed but a whole page is fetched). Second, it can easily lead to false sharing, which causes expensive protocol operations and communication to be invoked frequently. Under a sequential consistency model, invalidations are propagated and performed as soon as a write is detected, so pages may be frequently ping-ponged back and forth among processors due to either true or false sharing. (Figure 9.11 shows an example.) The high cost and high frequency of the operations are an unfortunate combination, so it is very important that the effects of false sharing and the frequency of communication in general be alleviated. This leads to very different protocols and approaches

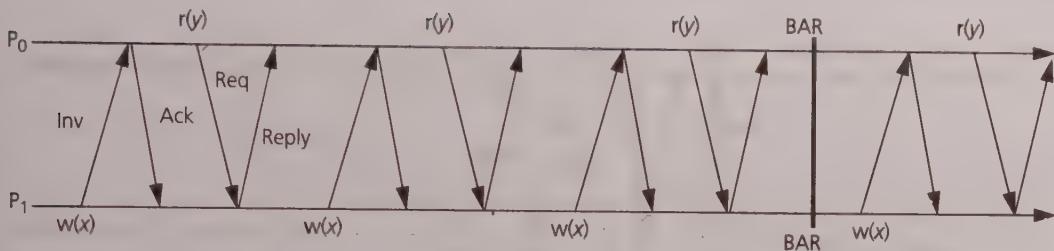


FIGURE 9.11 Problem with sequential consistency for SVM. Time proceeds from left to right in the figure. The operations that a process performs are shown above or below the horizontal timeline for that process. Process P_0 repeatedly reads variable y while process P_1 repeatedly writes variable x , which happens to fall on the same page as y . Since P_1 cannot proceed under SC until invalidations are propagated and acknowledged, the invalidations are propagated immediately, and substantial (and very expensive) communication ensues repeatedly due to this false sharing.

than those used for fine-grained coherence, where false sharing is much less significant, so let us examine them in some depth.

Using Relaxed Memory Consistency

The frequency of communication is reduced by exploiting a relaxed memory consistency model such as release consistency. This allows coherence actions such as invalidations or updates, collectively called *write notices* in SVM systems, to be postponed until the next synchronization point (writes do not have to become visible until then). Let us continue to assume an invalidation-based protocol. Figure 9.12 shows the same example as Figure 9.11: the writes to x by processor P_1 will not generate invalidations to the copy of the page at P_0 until the barrier is reached, so the effects of false sharing will be greatly mitigated and none of the reads of y by P_0 before the barrier will incur page faults. Of course, when P_0 accesses y after the barrier, it will incur a page fault due to false sharing since the page has now been invalidated. Similar communication reduction would be observed for true sharing as well, since the protocol does not distinguish between the two: even true sharing modifications would not be observed until the next synchronization point, which is okay according to the consistency model.

There is a significant difference here from how relaxed consistency is typically used in hardware-coherent machines or writes. There, it is used to avoid stalling the processor to wait for acknowledgments (completion), but the invalidations are usually propagated and applied as soon as possible since this is the natural thing for hardware to do. Although release consistency does not guarantee that the effects of the writes will be seen until the synchronization, in fact they usually will be. The amount of false sharing of cache blocks is therefore not reduced much, even within a period with no synchronization, nor is the number of network transactions or messages; the goal is mostly to hide latency from the processor. In the SVM case, the system takes the contract literally: invalidations are actually not propagated until the

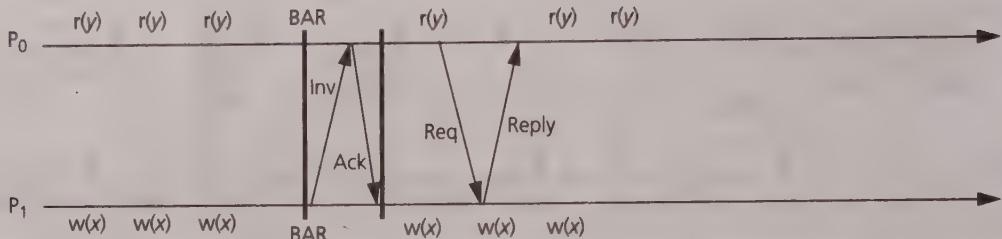


FIGURE 9.12 Reducing SVM communication due to false sharing by using a relaxed consistency model. No communication occurs at reads and writes until a synchronization event, at which point invalidations are propagated to make pages coherent. Only the first access to an invalidated page after a synchronization point generates a page fault and hence a request.

synchronization points. Of course, this makes it critical for correctness that all synchronization points be clearly labeled and communicated to the system.⁵

When exactly should invalidations (or write notices) be propagated from the writer to other copies, and when should they be applied? One possibility is to propagate them when the writer issues a release operation. At a release, invalidations for each page that the process wrote since its previous release are propagated to all processes that have copies of the page. If we wait for the invalidations to complete before proceeding past the release, this satisfies the sufficient conditions for RC that were presented earlier. However, even this propagation is sooner than necessary: under release consistency, a given process does not really need to see the write notice until it does an acquire. Propagating and applying write notices to all copies at release points, called *eager release consistency* or ERC (Carter, Bennett, and Zwaenepoel 1991, 1995), is conservative because the system does not know when the next acquire by another processor will occur or whether a given process will even perform an acquire and need to see those write notices. As shown in Figure 9.13(a), it can send expensive invalidation messages to more processes than necessary (P_2 need not have been invalidated by P_0); it requires separate messages for invalidations and lock acquisitions; and it may invalidate processes earlier than necessary, thus causing false sharing (see the false sharing between variables x and y , as a result of which the page is fetched twice by P_1 and P_2 —once at the read of y and once at the write of x). The extra messages problem is even more significant when an update-based protocol is used and repeated writes to a page generate repeated updates. These issues and alternatives are discussed further in Exercises 9.23–9.25.

5. In fact, a similar approach can be used to reduce the effects of false sharing in hardware as well. Invalidations may be buffered in hardware at the requestor and sent out only at a release or a synchronization (depending on whether release consistency or weak ordering is being followed) or when the buffer becomes full. Or they may be buffered at the destination and only applied at the next acquire point by that destination. This approach has been called *delayed consistency* (Dubois et al. 1991) since it delays the propagation of invalidations. As long as processors do not see the invalidations, they continue to use their copies without any coherence actions, alleviating false sharing effects.

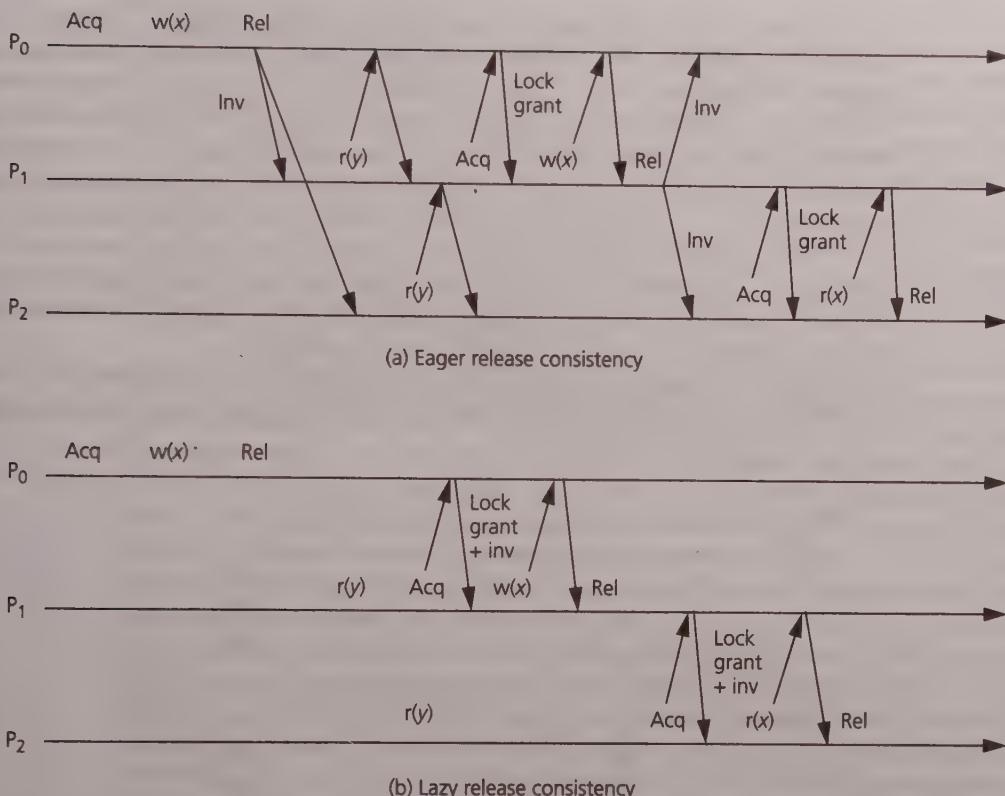


FIGURE 9.13 Eager versus lazy implementations of release consistency. Eager release consistency performs consistency actions (invalidation) at a release point, whereas lazy release consistency performs them at an acquire. Variables x and y are on the same page. The reduction in communication can be substantial, particularly for SVM systems with their large granularity of coherence.

The best-known SVM systems tend to use a form of release consistency, called *lazy release consistency* or LRC. As shown in Figure 9.13(b), LRC propagates and applies invalidations to a given process not at the release that follows the writes but only at the next acquire by that process (Keleher, Cox, and Zwaenepoel 1992). On an acquire, the process obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation and applies them to the relevant pages. Identifying which are the release operations that occurred before a given acquire is an interesting question. They can be defined as all releases that would have to appear before this acquire in any sequentially consistent ordering of the synchronization operations (that preserves dependences among them as well).⁶ Another way of putting this is that two

6. The ordering could even be processor consistent since RC allows acquires (reads) to bypass previous releases (writes) in program order.

types of partial orders are imposed on synchronization operations: program order within each process and a dynamically determined dependence order among acquires and releases to the same synchronization variable (by all processes). Accesses to a synchronization variable form a chain of successful acquires and releases in the dependence order. When an acquire request comes to a releasing process P , the synchronization operations that have occurred before that release are those that precede it in the intersection of these program orders and dependence orders. These synchronization operations are said to have occurred before the release in a *causal* sense. The operations before the acquire are the union of these operations with the operations that precede the acquire in program order. Figure 9.14 clarifies this concept of causal order among synchronization operations.

By further postponing coherence actions to acquires, LRC alleviates the three problems associated with ERC; for example, if memory operations that exhibit false

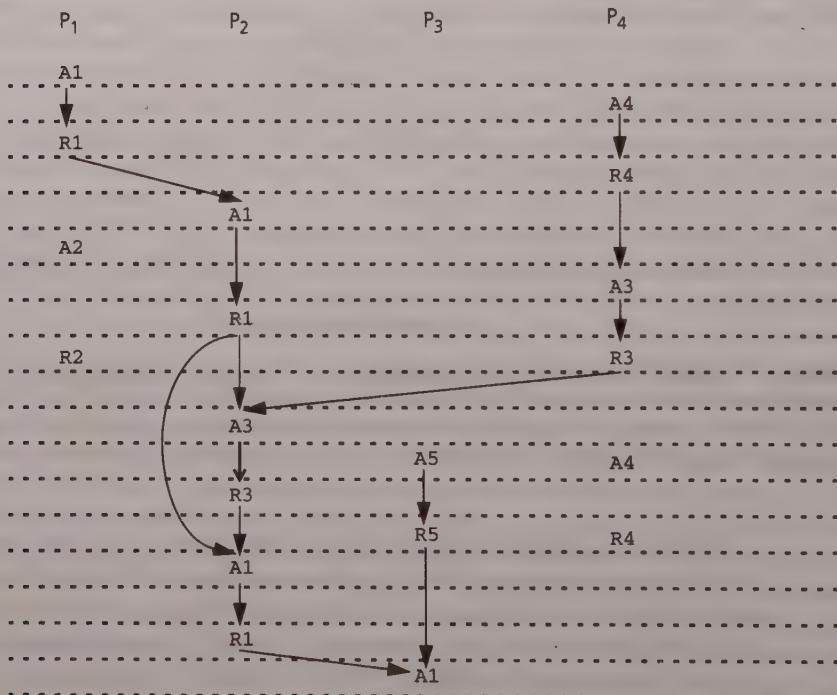


FIGURE 9.14 The causal order among synchronization operations and hence the groups of data accesses between them. The figure shows what the synchronization operations are before the acquire A_1 (by process P_3) and the release R_1 (by process P_2) that enables it. The dotted horizontal lines are time increments, increasing downward, indicating a possible interleaving in time. The bold arrows show dependence orders along an acquire-release chain, whereas the gray arrows show the program orders that are part of the causal order. The A_2 , R_2 and A_4 , R_4 pairs that are untouched by arrows do not happen before the acquire of interest in causal order, so the data accesses after the acquire are not guaranteed to see the accesses between those pairs.

sharing on a page occur before the acquire but not after it, their ill effects will not be seen (there is no page fault on the read of y in Figure 9.13[b]). On the other hand, some of the work and communication to be done is shifted from release point to acquire point, and LRC is significantly more complex to implement than ERC, as we shall see. Intermediate approaches are possible, such as propagating write notices at a release but applying them only at an acquire, thus saving not on write traffic but on page faults. However, LRC is currently the method of choice.

The relationship between these page-based software protocols and the consistency models developed for hardware-coherent systems is also interesting. The software protocols do not satisfy the requirements of coherence discussed in Chapter 5 since writes are not automatically guaranteed to be propagated unless the appropriate synchronization is present. Making writes visible only through synchronization operations also makes write serialization more difficult to guarantee. Different processes may see the same writes through different synchronization chains and hence in different orders, so most software systems do not guarantee write serialization. In fact, synchronization-based relaxed consistency specifications like release consistency do not guarantee coherence according to the definitions of Chapter 5. Finally, the only difference between hardware implementations of release consistency and software ERC lies in when writes are propagated. However, by propagating write notices only at acquires, LRC implementations may differ from release consistency even in whether writes are propagated and hence may allow results that are not permitted under release consistency. LRC is therefore a different consistency model than release consistency and requires greater programming care (see Example 9.2), whereas ERC is simply a different implementation of release consistency. However, if a program is properly labeled, in the sense of labeling all synchronization operations as discussed in Section 9.1.2, then it is guaranteed to run “correctly” under both RC and LRC, and both coherence and sequential consistency will appear to be satisfied.

EXAMPLE 9.2 Design an example in which LRC produces a different result than RC. How would you avoid the problem?

Answer Consider the code fragment below, assuming the pointer `ptr` is initialized to `NULL`.

<code>P₁</code>	<code>P₂</code>
----------------------------	----------------------------

```

lock L1;
ptr = non_null_ptr_val;
unlock L1; . . .
while (ptr == null) {};
lock L1;
a = ptr;
unlock L1;

```

Under RC and ERC, the new non-null pointer value is guaranteed to propagate to P_2 before the unlock (release) by P_1 is complete, so P_2 will see the new value and

jump out of the loop as expected. Under LRC, P_2 will not see the write by P_1 until it performs its lock (acquire operation); it will therefore enter the while loop, never see the write by P_1 , and hence never exit the while loop. The solution is to put the appropriate acquire synchronization before the reads in the while loop or to label the accesses to `ptr` appropriately as synchronizations to create a properly labeled program. ■

The fact that coherence information is propagated only at synchronization operations that are recognized by the software SVM layer has an interesting, related implication. It may be difficult to run existing application binaries “as is” on SVM systems that use relaxed consistency models, even if those binaries were compiled for systems that support a very relaxed consistency model and they are properly labeled. The reason is that the labels have already been compiled down to the specific fence instructions used by the commercial microprocessor, and those fence instructions may not be visible to the software SVM layer. Of course, if the source code or assembly code with the labels is available, then the labels can be translated to primitives recognized by the SVM layer; and if only the binary is available then it can be edited, using available tools, and instrumented to make the labels visible to the SVM runtime system.

Multiple Writer Protocols

Delaying write notices works very well in mitigating the effects of false sharing when only one of the sharers writes the page in the interval between two synchronization points, as in our previous examples (the others may read the page). However, it does not in itself solve the multiple writer problem. Consider the revised example in Figure 9.15. Now P_0 and P_1 both modify the same page between the same two barriers. If we follow a protocol in which only a single writer is allowed at a time, then each of the writers must obtain ownership of the page before writing it, leading to ping-ponging communication even between the synchronization points and compromising the potential benefits of the relaxed consistency model (which allows multiple writers to coexist). To truly exploit the benefits of relaxed consistency, we need a *multiple writer* protocol. This is a protocol that allows each processor writing a page between synchronization points to modify its own copy locally, letting the copies become inconsistent, and makes the copies consistent only at the next synchronization point as needed by the consistency model. Let us look briefly at some multiple writer mechanisms that can be used with either eager or lazy release consistency.

The first method is used in the TreadMarks SVM system from Rice University (Keleher et al. 1994). The idea is quite simple. To capture the modifications to a shared page, it is initially write protected. At the first write after a synchronization point, a protection violation occurs. At this point, the system makes a copy of the page (called a *twin*) in software and then unprotects the actual page so further writes can happen without protection violations. Later, at the next release or incoming acquire at that process (for ERC or LRC, respectively), the twin and the current copy are compared to create a “diff,” which is simply a compact encoded representation of

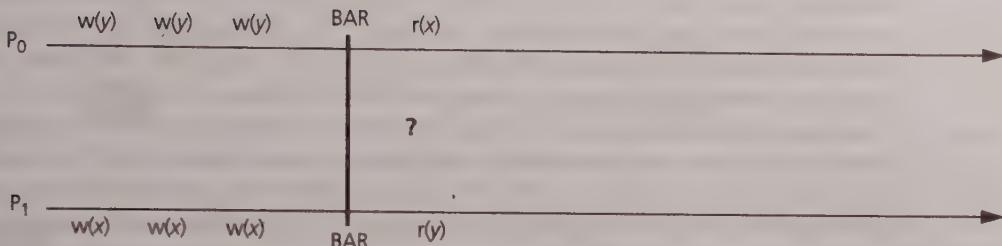


FIGURE 9.15 The multiple writer problem. At the barrier, two different processors have written the same page independently, and their modifications need to be merged.

the differences between the two. The diff therefore captures the modifications that processor has made to the page in that synchronization interval. When a processor incurs a page fault, it must obtain the diffs for that page from other processors that have created them and merge them into its copy of the page. As with write notices, several alternatives are available for when we might compute the diffs and when we might propagate them to other processors with copies of the page (see Exercise 9.23). If diffs are propagated eagerly at a release, they and the corresponding write notices can be freed immediately and the storage reused. In a lazy implementation, diffs and write notices may be kept at the creator until they are requested. In that case, they must be retained until it is clear that no other processor needs them. Since the amount of storage needed by these diffs and write notices can become very large, garbage collection becomes necessary (by forcibly propagating diffs and write notices, for example). This garbage collection algorithm is quite complex and expensive since when it is invoked each page may have uncollected diffs distributed among many nodes (Keleher et al. 1994).

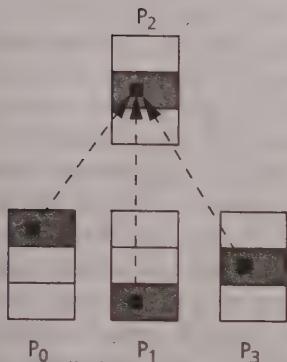
An alternative software multiple writer method gets around the garbage collection problem for diffs while still implementing LRC and makes a different set of performance trade-offs (Iftode, Singh, and Li 1996b; Zhou, Iftode, and Li 1996). The idea here is to not maintain the diffs at the writer until they are requested nor to propagate them to all the copies at a release, but rather to do something in between. Every page has a home node, just like in flat hardware cache coherence schemes, and the diffs are propagated to the home at a release. The releasing processor can then free the storage for the diffs as soon as it has sent them to the home. The arriving diffs are merged into the home copy of the page (and the diff storage freed there too), which is therefore kept up-to-date. A processor performing an acquire obtains write notices for pages from the previous releaser just as before. However, when it has a subsequent page fault on one of those pages, it does not obtain diffs from all previous writers but rather fetches the whole page from the home. This is called a *home-based protocol*. In addition to much lower storage overhead and better storage scalability, it has the performance advantage that on a page fault only one round-trip message is required to fetch the data whereas, in the previous scheme, diffs had to be

obtained from all the previous (“multiple”) writers. Also, a processor never incurs a page fault for a page for which it is the home. The disadvantages are that whole pages are fetched rather than diffs (though this can be traded off with storage and protocol processing overhead by storing the diffs at the home and not applying them there and then fetching diffs from the home) and that the distribution of pages among homes becomes important for performance despite replication in main memory. Which scheme performs better will depend on how the application sharing patterns manifest themselves at page granularity and on the performance characteristics of the communication architecture.

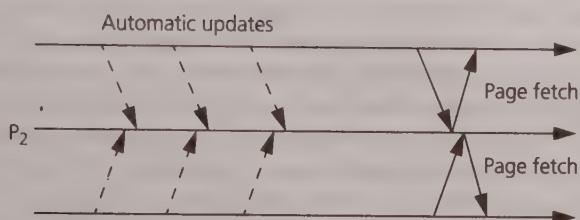
Alternative Methods for Propagating Writes

Diff processing—twin creation, diff computation, and diff application—incurs significant overhead, requires substantial additional storage, and can also pollute the first-level processor cache, replacing useful application data. Some recent systems provide hardware support for fine-grained communication in the network interface, particularly fine-grained propagation of writes to remote memories, that can be used to accelerate these home-based, multiple writer SVM protocols and avoid diffs altogether. The idea of hardware propagation of writes originated in the PRAM (Lipton and Sandberg 1988) and PLUS (Bisiani and Ravishankar 1990) systems; modern examples include the network interface of the SHRIMP multicomputer prototype at Princeton University (Blumrich et al. 1994) and the Memory Channel from Digital Equipment Corporation (Gillett, Collins, and Pimm 1996).

These network interfaces allow mappings to be established between a pair of pages on different nodes so that the writes performed to the source page are propagated in hardware to the destination page. The writes can be detected by snooping the memory bus (as in the SHRIMP case, called an *automatic update mechanism*) or by software instrumentation of write operations to generate special writes to a different address space (as in the Memory Channel, called a *write doubling mechanism*). The detected writes are then propagated according to the mappings by the network interface, which may even reside on the I/O bus. The snooping approach may require that caches be write through, while the latter approach experiences extra instruction overhead and requires instrumentation. By establishing such mappings from the copies of a page to the home copy (when those copies are first made), writes will be propagated to the home, which can be kept up-to-date according to the consistency model (see Figure 9.16). Consistency actions like propagating and applying write notices are managed at synchronization points exactly as before, and the entire page is fetched from the home on a page fault. Home-based protocols have been developed using these features (Iftode et al. 1996; Iftode, Singh, and Li 1996a; Kontothanassis and Scott 1996), and in fact they inspired the all-software home-based protocols. These fine-grained write propagation approaches avoid diffs entirely; however, they require hardware support and they increase data traffic by propagating all writes rather than only the final new values produced by the end of a synchronization interval.



(a) The automatic update mechanism



(b) Supporting multiple writers using automatic update

FIGURE 9.16 Using an automatic update mechanism to solve the multiple writer problem.

The variables x and y fall on the same page, which has node P_2 as its home. If P_0 (or P_1) were the home, it would not need to propagate automatic updates and would not incur page faults on that page (only the other node would).

An all-software alternative to computing diff's is to maintain dirty bits per word or per block in main memory in software (Zekauskas, Sawdon, and Bershad 1994). A *dirty bit* for a word keeps track of whether that word has been written by the local node since the dirty bit was last cleared. Dirty bits are cleared at synchronization points, and the dirty bits that are found to be set in a page upon reaching a synchronization point indicate the equivalent of a diff for that page. While determining diff's does not require pages to be compared with their twins, the setting and unsetting of dirty bits requires extra instructions and instrumentation, similar to that needed by the write propagation in the Memory Channel interface. Software analysis can be used to reduce the overhead, but it remains significant.

The discussion so far has focused on the functionality of different degrees of laziness but has not addressed implementation. How do we ensure that the necessary write notices to satisfy the partial orders of causality get to the right places at the right time, and how do we reduce the number of write notices transferred? A range of methods and mechanisms is available for implementing release consistency protocols of different forms (single versus multiple writer, acquire- versus release-based, degree of laziness actually implemented). The mechanisms, the forms of laziness each can support, and their trade-offs are interesting and are discussed in Section 9.6.2.

Summary: Path of a Read Operation

To summarize the behavior of an SVM system, let us look at the path of a read. We examine the behavior of a home-based system since it is simpler. A read reference first undergoes address translation from a virtual to a physical address in the processor's

memory management unit. If a local page mapping is found, the cache hierarchy is looked up and it behaves just like a regular uniprocessor operation. If a local page mapping is not found, a page fault occurs and then the page is mapped in, providing a physical address. If the operating system indicates that the page is not currently mapped on any other node, then it is mapped in from disk with read-write permission, otherwise it is obtained from another node with read-only permission. Now the cache is looked up. If inclusion is preserved between the local memory and the cache hierarchy, as it normally will be to keep the caches coherent with local memory, the reference will miss and the block is loaded into the cache from local main memory where the page is now mapped. Note that inclusion means that a page must be flushed from the cache (or the state of the cache blocks changed) when it is invalidated in the local main memory, downgraded from read-write to read-only mode, or replaced. If a write reference is made to a read-only page, then a page fault is also incurred and ownership of the page obtained before the reference can be satisfied by the cache hierarchy.

Performance Implications

Lazy release consistency and multiple writer protocols improve the performance of SVM systems dramatically compared to sequentially consistent implementations. However, there are still many performance problems compared with machines that manage coherence in hardware at cache block granularity. The problems of false sharing and either extra communication or protocol processing overhead do not disappear with relaxed models, and the page faults and fetches that remain are still expensive to satisfy. The high cost of communication and the contention induced by higher endpoint processing overhead often greatly magnifies imbalances in communication volume and, hence, execution time among processors. Another problem in SVM systems is that synchronization is performed in software through explicit software messages and is very expensive. This is exacerbated by the fact that expensive page misses often occur within critical sections, artificially dilating them and hence greatly increasing the serialization at critical sections. The result is that while applications with coarse-grained data access patterns (little false sharing or communication fragmentation) and synchronization perform quite well on SVM systems, applications with finer-grained access patterns (i.e., accesses from different processes being interleaved finely in the shared virtual address space) and especially synchronization do not migrate well from hardware cache-coherent systems to SVM systems unless they are manufactured (Jiang, Shan, and Singh 1997). The scalability of SVM systems is also undetermined, both in performance and in the ability to run large problems since the storage overhead of auxiliary data structures grows with the number of processors.

Overall, it is still unclear whether fine-grained applications that run well on hardware-coherent machines can be restructured to run efficiently on SVM systems as well and whether or not such systems are viable for a wide range of applications. Research is being done to understand the performance issues and bottlenecks

(Dwarkadas et al. 1993; Iftode, Singh, and Li 1996a; Kontothanassis et al. 1997; Jiang, Shan, and Singh 1997) as well as the value of adding some hardware support for fine-grained communication while still maintaining coherence in software at page granularity (Kontothanassis and Scott 1996; Iftode, Singh, and Li 1996a; Bilas, Iftode, and Singh 1998). With the dramatically increasing popularity of low-cost SMPs, there is also a lot of research in extending SVM protocols to build coherent shared address space machines as a two-level hierarchy: hardware coherence within the SMP nodes and software SVM coherence across SMP nodes (Erlichson et al. 1996; Stets et al. 1997; Samanta et al. 1998). The goal is for the outer SVM protocol to be invoked as infrequently as possible and only when cross-node coherence is needed, while still preserving laziness within the node as well. The extension to cache-coherent distributed-memory nodes rather than SMP nodes is natural to contemplate. In fact, instead of an inexpensive but low-performance substitute for hardware coherence across uniprocessor nodes, software shared memory approaches like SVM can be seen as a way of extending the coherent shared address space programming model from available hardware-coherent multiprocessor nodes to clusters of such nodes and thus constructing large-scale systems. Let us examine some other software approaches.

9.3.4

Access Control through Language and Compiler Support

Language and compiler support can also be enlisted to support coherent replication. One approach is to program in terms of data objects or “regions” of data and have the run-time system that manages these objects provide access control and coherent replication at the granularity of objects. By explicitly using objects, this approach does not view memory as a flat address space. This “shared object space” programming model motivates the use of even more relaxed memory consistency models, as we shall see next. We shall also briefly discuss compiler-based coherence and approaches that provide a shared address space in software but do not provide automatic replication and coherence.

Object-Based Coherence

The release consistency model takes advantage of the “when” dimension of memory consistency—it tells us when it is necessary for writes by one process to be performed with respect to another process. This allows successively lazy implementations to be developed, as we have discussed, which delay the performing of writes as long as possible. However, even with release consistency, if the synchronization in the program requires that process P_1 ’s writes be performed or become visible at process P_2 by a certain point, this means that all of P_1 ’s writes to all the data it wrote become visible even if P_2 does not need to see all the data (see Figure 9.17). More relaxed consistency models take into account the “what” dimension, by propagating invalidations or updates only for that data that the process acquiring the synchronization may

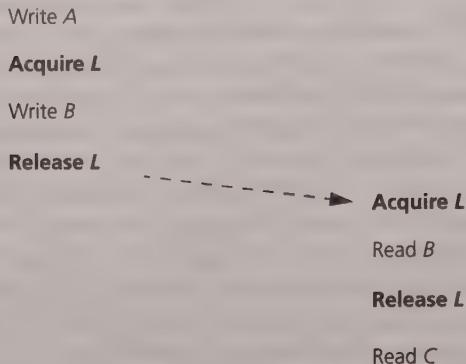


FIGURE 9.17 Why release consistency is conservative. Suppose A and B are on different pages. Since P_1 wrote A and B before releasing L (even though A was not written inside the critical section), invalidations for both A and B will be propagated to P_2 and applied to the copies of A and B there. However, P_2 does not need to see the write to A but only that to B . Suppose now P_2 reads another variable C that resides on the same page as A . Since the page containing A has been invalidated, P_2 will incur a page miss on its access to C due to false sharing. If we could somehow associate the page containing B with the lock L , then only the invalidation of B could be propagated on the acquire of L by P_2 , and the false sharing miss would be saved.

actually need to see according to the causal synchronization relationships. The when dimension in release consistency is specified by the programmer through the synchronization inserted in the program. The question is how to specify the what dimension. It is possible to associate with a synchronization event (or variable) the set of pages that must be made consistent with respect to that event. However, this is very awkward for a programmer to do.

Region- or object-based approaches provide a better solution. The programmer breaks up the data into logical objects or regions (regions are arbitrary, user-specified ranges of virtual addresses that are treated like objects but do not require object-oriented programming). A run-time library then maintains consistency at the granularity of these regions or objects rather than leaving it entirely to the operating system to do at the granularity of pages. The disadvantages of this approach are the additional programming burden of specifying and using regions or objects appropriately and the need for a sophisticated run-time system between the application and the OS. The major advantages are (1) the use of logical objects (rather than fixed machine granularities as coherence units) by itself can help reduce false sharing and fragmentation to begin with, and (2) they provide a handle on specifying data logically and can be used to relax the consistency using the what dimension.

For example, in the *entry consistency* model (Bershad, Zekauskas, and Sawdon 1993), the programmer associates a set of data (regions or objects) with every synchronization variable such as a lock or barrier or with every synchronization event in the program (these associations or bindings can be changed at run time, with

some cost). At synchronization events, only those objects or regions that are associated with that synchronization variable are guaranteed to be made consistent. Write notices for other modified data do not have to be propagated or applied. If no bindings are specified for a synchronization variable, the default release consistency model is used for it. However, the bindings are not hints: if they are specified, they must be complete and correct or the program will obtain the wrong answer. The need for explicit, correct bindings imposes a substantial burden on the programmer, and sufficient performance benefits have not yet been demonstrated to make this worthwhile. The Jade programming language achieves a similar effect, although by specifying data usage in a different way (Rinard, Scales, and Lam 1993). Finally, attempts have been made to exploit the association between synchronization and data implicitly even in a page-based shared virtual memory approach, using a model called *scope consistency* (Iftode, Singh, and Li 1996b).

Compiler-Based Coherence

Research has focused on having the compiler keep caches coherent in a shared address space, by using additional hardware support in the processor system. These approaches rely on the compiler (or programmer) to identify parallel loops. A simple approach to coherence is to insert a barrier at the end of every parallel loop and flush the caches between loops. However, this does not allow any data locality to be exploited in caches across loops. Even if only shared data is flushed, data that is declared in the shared address space but is not actively shared will also be unnecessarily flushed. More sophisticated approaches have been proposed that require support for selective invalidations and fairly sophisticated hardware support to keep track of which blocks to invalidate (Cheong and Viedenbaum 1990). Other than nonstandard hardware and compiler support for coherence, the major problem with these approaches is that they rely on the automatic parallelization of sequential programs by the compiler, which is not very successful yet for realistic programs.

Shared Address Space without Coherent Replication

Systems in this category support a shared address space abstraction through the language and compiler but without automatic replication and coherence, just like the CRAY T3D and T3E did in hardware. One type of example is a data parallel language like High Performance Fortran (see Chapter 2). The distributions of data specified by the user, together with the owner computes rule, are used by the compiler or runtime system to translate off-node memory references to explicit messages, to make messages larger, to align data for better spatial locality, and so on. Replication and coherence are usually left up to the user, which compromises ease of programming; alternatively, system software may try to manage coherent replication in main memory automatically. Efforts similar to HPF are being made with languages based on C and C++ as well (Bodin et al. 1993; Larus, Richards, and Viswanathan 1996).

A more flexible language- and compiler-based approach is taken by the Split-C language (Culler et al. 1993). Here, the user explicitly specifies arrays as being local or global (shared) and for global arrays specifies how they should be laid out among physical memories. Computation may be assigned independently of the data layout, and references to global arrays are converted into messages by the compiler or run-time system based on the layout. The decoupling of computation assignment from data distribution makes the language much more flexible than an owner computes rule for load-balancing irregular programs, but it still does not provide automatic support for replication and coherence, which can be difficult for the programmer to manage. Of course, all these software systems can be easily ported to hardware-coherent shared address space machines, in which case the shared address space, replication, and coherence are implicitly provided. In this case, the run-time system may be used to manage replication and coherence in main memory and to transfer data in larger chunks than cache blocks, but these capabilities may not be necessary.

9.4

PUTTING IT ALL TOGETHER: A TAXONOMY AND SIMPLE COMA

The approaches to managing replication and coherence in the extended memory hierarchy discussed in this chapter have a range of goals: improving performance by replicating in main memory in the case of COMA and reducing cost in the case of SVM and the other systems of the previous section. Examining the management of replication and coherence in a unified framework leads to the design of alternative systems that can pick and choose aspects of existing ones. A useful framework is one that distinguishes the approaches along two closely related axes:

1. the granularities at which they allocate data in the lowest-level replication store, keep data coherent, and communicate data between nodes
2. the degree to which they utilize additional hardware support in the communication assist beyond that available in uniprocessor systems

The two axes are related because some functions are either not possible at fine granularity without additional hardware support (e.g., allocation of data in main memory) or not possible with high performance. The framework applies whether replication is done only in the cache (as in CC-NUMA) or in main memory.

Figure 9.18 depicts the overall framework and places different types of systems in it. We divide granularities into “page” and “block” (cache block) since these are the most common in transparent shared address space systems that do not require a stylized programming model such as objects. Other fine granularities such as individual words and coarse granularities such as objects or regions of memory can also be included in this framework. The granularities of allocation, coherence (access control), and communication influence one another, as we shall see.

On the left side of the figure are COMA systems, with allocation in main memory at the granularity of cache blocks using additional hardware support. CC-NUMA

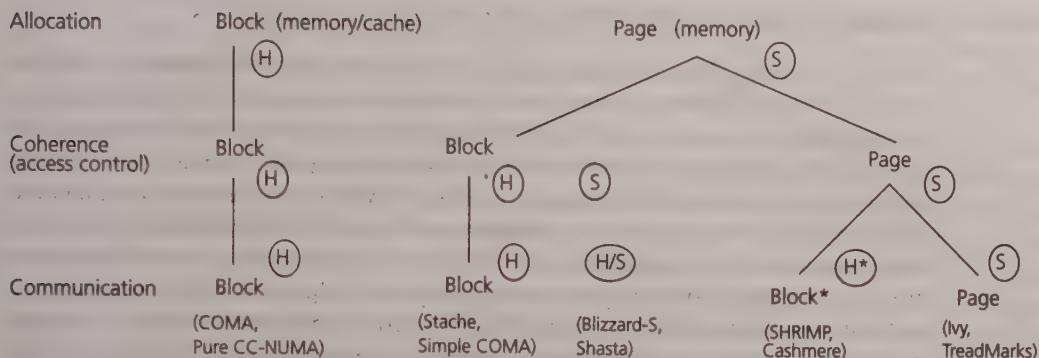


FIGURE 9.18 Granularities of allocation, coherence, and communication in coherent shared address space systems. The granularities specified are for the replication store in which data is first replicated automatically when it is brought into a node. This level is main memory in all the systems, except for pure CC-NUMA where it is the cache. Below each leaf in the taxonomy are listed some representative systems, protocols, or system families of that type, and next to each node is a letter indicating whether the support for that function is usually provided in hardware (H) or software (S). The asterisk next to “block” and “H” in one case means that not all communication is performed at fine granularity. Citations for these systems or system families include COMA (Hagersten, Landin, and Haridi 1992; Stenstrom, Joe, and Gupta 1992; Frank, Burkhardt, and Rothnie 1993), pure CC-NUMA (Laudon and Lenoski 1997), Stache (Reinhardt, Larus, and Wood 1994), Simple COMA (Saulsbury et al. 1995), Blizzard-S (Schoinas et al. 1994), Shasta (Scales, Gharachorloo, and Thekkath 1996), SHRIMP (Blumrich et al. 1994; Iftode et al. 1996), Cashmere (Kontothanassis and Scott 1996), Ivy (Li and Hudak 1989), and TreadMarks (Keleher et al. 1994).

systems that replicate data only in the cache, called pure CC-NUMA systems, also fall in this category. Given allocation at cache block granularity, it makes sense to keep data coherent at a granularity at least this fine as well and to communicate at fine granularity.⁷

On the right side of the figure are systems that allocate and manage space in main memory at page granularity, with no extra hardware needed for this function. These systems may provide access control and hence coherence at page granularity as well, as in SVM, in which case they may or may not provide support for fine-grained communication. Or they may provide access control and coherence at a finer, block granularity, using either software instrumentation or hardware support. Systems that support coherence at fine granularity typically provide some form of hardware support for efficient fine-grained communication as well.

7. This does not mean that fine-grained allocation necessarily implies fine-grained coherence or communication. For example, it is possible to exploit communication at coarse grain even when allocation and coherence are at fine grain to gain the benefits of large data transfers. However, the situations discussed are indeed the common case, and we shall focus on these.

9.4.1 Putting It All Together: Simple COMA and Stache

While COMA and SVM both replicate in main memory, and each addresses some but not all of the limitations of pure CC-NUMA, they are at two ends of the spectrum in the preceding taxonomy. COMA is a hardware-intensive solution and maintains fine granularities, but some issues that it raises in managing main memory (such as the last copy problem) are challenging for hardware. SVM leaves these complex memory management problems to system software—which simplifies hardware and enables main memory to be managed as a fully associative cache through the OS virtual-to-physical mappings—but its performance may suffer due to the large granularities and software overhead of coherence. The framework in Figure 9.18 leads to interesting ways to combine the low cost and hardware simplicity of SVM with the performance advantages and ease of programming of COMA; namely, the Simple COMA (Saulsbury et al. 1995) and Stache (Reinhardt, Larus, and Wood 1994) approaches shown in the middle of the figure. These approaches divide the task of coherent replication in main memory into two parts: memory management (address translation, allocation, and replacement) and coherence (including replication and communication). Like COMA, these approaches provide coherence at fine granularity with specialized hardware support for high performance, but like SVM (and unlike COMA), they leave memory management to the operating system at page granularity. Let us begin with Simple COMA.

The major appeal of Simple COMA relative to COMA is design simplicity. Performing memory management through the virtual memory system simplifies the hardware protocol and also allows fully associative management of the attraction memory with arbitrary replacement policies. To provide coherence or access control at fine grain in hardware, each page in a node's main memory is divided into coherence blocks of any chosen size (say, a cache block), and state information is maintained in hardware for each of these blocks. Unlike in COMA, there is no need for tags since the presence check is done at page level. The page permission is checked, as usual, before the block is accessed. The state of the block is checked in parallel with the memory access when a miss occurs in the processor cache. Thus, there are two levels of access control: for the page, under operating system control, and if that succeeds, then for the block, under hardware control.

Consider the performance trade-offs relative to COMA. Simple COMA reduces the latency for accesses satisfied in the local attraction memory (which is hopefully the frequent case among cache misses). There is no need for the hardware tag comparison and selection used in COMA or in fact for the local/remote address check that is needed in pure CC-NUMA machines to determine whether to look up the local memory. On the other hand, every cache miss looks up the local attraction memory, so like in COMA the path of a nonlocal access is longer. Since a shared page can reside in different physical addresses in different memories, controlled independently by the node operating systems, unlike in COMA we cannot simply send a block's physical address across the network on a miss and use it at the other end. This means that we must support a shared virtual, rather than physical, address space. However, the virtual address issued by the processor is no longer available by the time the attraction memory miss is detected. The physical address, which is

available, must be “reverse translated” to a virtual address or other globally consistent identifier; this identifier is sent across the network and is translated back to a (potentially different) physical address by the other node. This process incurs some added latency and is discussed further in Section 9.6.

Another drawback of Simple COMA compared to COMA is that, although communication and coherence are at fine granularity, allocation is at page granularity. This can lead to fragmentation in main memory when the access patterns of an application do not match page granularity well. If a processor accesses only one word of a remote page, only that coherence block will be communicated, but space will be allocated in the local main memory for the whole page. Similarly, if only one block is brought in for an unallocated page, it may have to replace an entire page of useful data (fortunately, the replacement is fully associative and under the control of software, which can make sophisticated choices). In contrast, COMA systems typically allocate space for only that coherence block in the attraction memory. Simple COMA is therefore more sensitive to spatial locality than COMA.

An approach similar to Simple COMA is taken in the Stache design proposed for the Typhoon system (Reinhardt, Larus, and Wood 1994) and implemented in the Typhoon-0 research prototype (Reinhardt, Pfile, and Wood 1996). Unlike Simple COMA, Stache does not manage all of memory as a cache but uses the tertiary cache approach discussed earlier for replication in main memory; however, like Simple COMA it manages allocation at page level in software and coherence at fine grain in hardware. The assist in the Typhoon systems is programmable, and physical addresses are reverse translated to virtual addresses rather than other global identifiers to enable protocol handlers to be written in user-level software (see Section 9.6). Designs have also been proposed to combine the benefits of CC-NUMA and Simple COMA (Falsafi and Wood 1997).

Summary: Path of a Read Reference

Consider the path of a read in Simple COMA. The virtual address is first translated to a physical address by the processor’s memory management unit. If a page fault occurs, space must be allocated for a new page, though data for the page is not loaded in. The virtual memory system decides which page to replace, if any, and establishes the new mapping. To preserve inclusion, data for the replaced page must be flushed or invalidated from the cache. All blocks on the newly mapped page are set to invalid. The physical address is then used to look up the cache hierarchy. If it hits (it will not if a page fault occurred), the reference is satisfied. If not, then it looks up the local attraction memory, where by now the locations are guaranteed to correspond to that page. If the block of interest is in a valid state, the reference completes. If not, the physical address is reverse translated to a global identifier that plays the role of a virtual address, which is sent across the network guided by the directory coherence protocol. The remote node translates this global identifier to a local physical address, uses this physical address to find the block in its memory hierarchy, and sends the block back to the requestor. The block is then loaded into the local attraction memory and cache and the data is delivered to the processor.

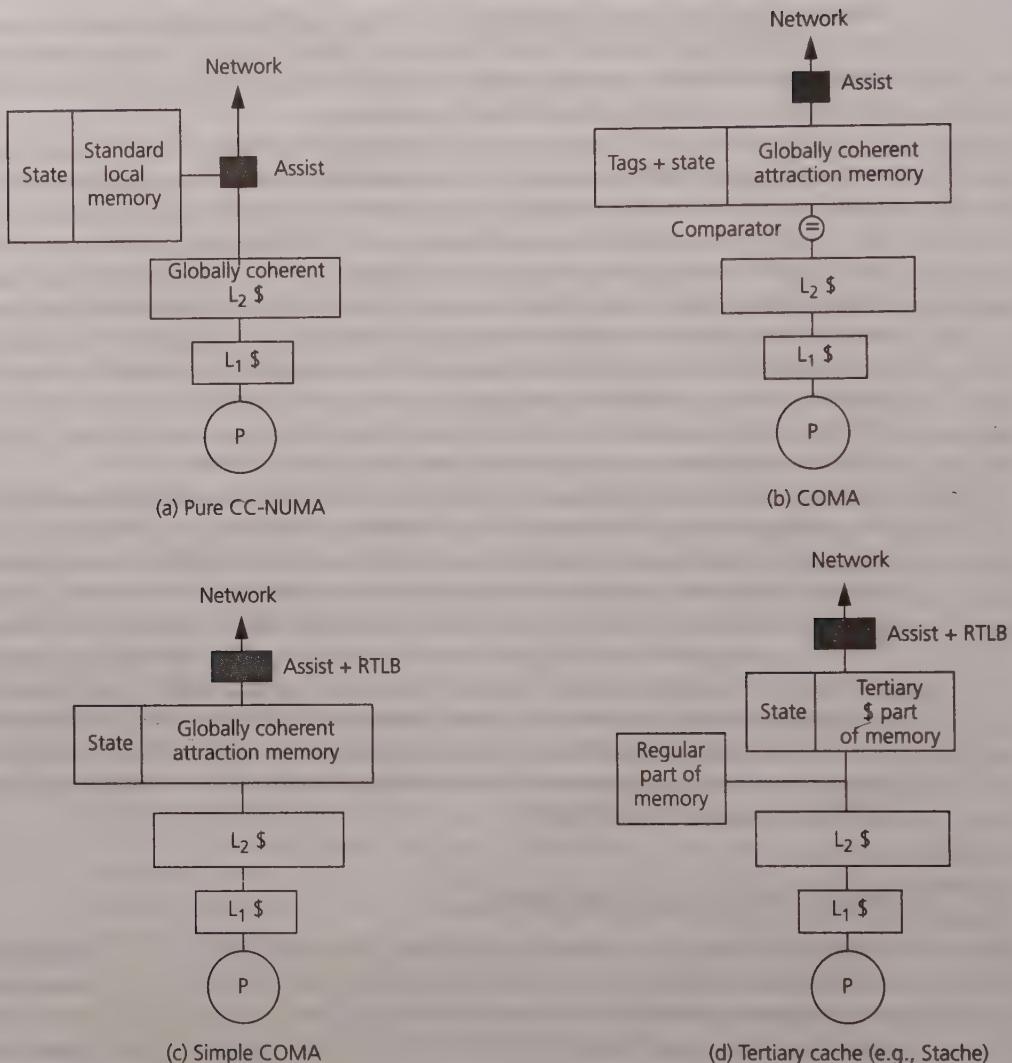


FIGURE 9.19 Logical structure of a typical node in four approaches to a coherent shared address space. The common property of the approaches is that they maintain coherence at fine granularity in hardware. A vertical path through the node in the figures traces the path of a read miss that must be satisfied remotely (going through main memory means that main memory must be looked up, though this may be done in parallel with issuing the remote request if speculative lookups are used). RTLB stands for reverse translation lookaside buffer, and is a structure that provides reverse translation of addresses from physical to virtual (or to a global identifier).

Figure 9.19 summarizes the node structure of the approaches that use hardware support to preserve coherence at fine granularity.

9.5**IMPLICATIONS FOR PARALLEL SOFTWARE**

Let us examine the implications for parallel software of all the approaches discussed in this chapter, beyond the parallel programming issues already discussed in earlier chapters.

Relaxed memory consistency models require that parallel programs label the desired conflicting accesses as synchronization operations. The insertion points are usually quite stylized, for example, looking for variables that a process spins on in a while loop before proceeding; but sometimes orders must be preserved even without spin-waiting, as in some of the examples in Figure 9.4. A programming language may provide support to label some variables or accesses as synchronization, which will then be translated by the compiler to the appropriate order-preserving instructions. Current programming languages do not provide integrated support for such labeling but rely on the programmer inserting special instructions or calls to a synchronization library. Labels can also be used by the compiler itself to restrict its own reorderings of accesses to shared memory. One mechanism is to declare some (or all) shared variables to be of a “volatile” type, which means that those variables will not be allocated in registers and will not be reordered with respect to the accesses around them. Recall that register allocation of shared variables can cause coherence to be violated as well, so key shared variables like flags are often declared to be volatile for coherence itself, even under sequential consistency. Some new compilers also recognize explicit synchronization calls and don’t reorder memory operations across them (perhaps distinguishing between acquire and release calls) or obey orders with respect to special order-preserving instructions that the program may insert.

The automatic, fine-grained replication and migration provided by COMA machines is designed to allow the programmer to ignore the distributed nature of main memory. It is very useful when capacity or conflict misses dominate and data accesses are fine-grained. Experience with parallel applications indicates that because of the nature of working sets and the sizes of caches in modern systems, the migration feature may be more broadly useful than replication and coherence; that is, the benefits arise less frequently from having copies of a block present in multiple main (attraction) memories and more frequently from bringing the single copy of the data to the right attraction memory for a phase of computation. Of course, systems with small caches or applications with large, unstructured working sets can benefit from replication in main memory as well. Fine-grained, automatic migration is particularly useful when the data structures in the program cannot easily be distributed appropriately at page granularity, so page migration techniques such as those provided by the Origin2000 or even explicit page migration or placement may not be so successful; for example, when data that should be allocated on two different nodes falls on the same page (see Section 8.9). Data migration is also very useful in conjunction with process migration, although this case is likely to be handled quite well by data migration at page granularity.

In general, although COMA systems suffer from higher communication latencies than CC-NUMA systems, they may allow a wider set of workloads to perform well

with less programming effort. Interestingly, in flat COMA systems, explicit migration and proper data (home) placement can still be moderately useful despite the COMA nature. This is because ownership requests on writes still go to the directory, which may be remote and does not migrate with the data, and thus can cause extra traffic and contention even if only a single processor ever writes a page.

Commodity-oriented systems put more pressure on software not only to reduce communication volume but also to orchestrate data access and communication carefully since the costs and/or granularities of communication are much larger. Consider shared virtual memory, which performs communication and coherence at page granularity. The actual data access and sharing patterns interact with this granularity to produce an induced sharing pattern at page granularity, which is the pattern relevant to the system (Iftode, Singh, and Li 1996a). Other than by a high communication-to-computation ratio, performance is adversely affected when the induced pattern involves write sharing (and hence multiple writers of the same page) or fragmentation in communication. This makes it important to try to structure programs so that accesses from different processes tend not to be interleaved at a fine granularity in the address space. The high cost of synchronization and the dilation of critical sections due to page faults within them makes it especially important to reduce the use of synchronization in programming for SVM systems. Finally, the high cost of communication and synchronization may make it more difficult to use task stealing successfully for dynamic load balancing in SVM systems. The remote accesses and synchronization needed for stealing may be so expensive that little work is left to steal by the time stealing is successful. It is therefore much more important to have a well-balanced initial assignment of tasks for a task-stealing-based computation in an SVM system than in a hardware-coherent system (Jiang, Shan, and Singh 1997). In general, the importance of different programming and algorithmic optimizations depend on the communication costs and granularities of the system at hand.

9.6

ADVANCED TOPICS

Before we conclude this chapter, let us discuss two other topics: other limitations of the traditional CC-NUMA approach and the mechanisms and techniques that enable software to take advantage of relaxed memory consistency models, for example, in shared virtual memory protocols.

9.6.1 Flexibility and Address Constraints in CC-NUMA Systems

Two other limitations of traditional, pure CC-NUMA systems are the fact that a single coherence protocol is hardwired into the machine and the potential limitations of addressability in a shared physical address space. Let us discuss each in turn.

Providing Flexibility

One size never really fits all. It is always possible to find workloads that would be better served by a different protocol than the one hardwired into a given machine.

For example, while we have seen that invalidation-based protocols overall have advantages over update-based protocols for cache-coherent systems, update-based protocols are advantageous for purely producer-consumer sharing patterns. For a one-word producer-consumer interaction, in an invalidation-based protocol the producer will generate an invalidation that will be acknowledged, and then the consumer will issue a read miss that the producer will satisfy, leading to four network transactions; an update-based protocol will need only one transaction in this case. As another example, if large amounts of predetermined data are to be communicated from one node to another, then it may be useful to transfer them in a single large explicit message instead of one cache block at a time through load and store misses. A single protocol may not even best fit all phases or all data structures of a single application, or even the same data structure in different phases of the application. If the performance advantages of using different protocols in different situations are substantial, it may be useful to support multiple protocols in the communication architecture. This is particularly likely when the performance penalty for mismatched protocols is very high, as in commodity-based systems with less efficient communication architectures.

Protocols can be altered—or mixed and matched—by making the protocol processing part of the communication assist programmable rather than hardwired, thus implementing the protocol in software rather than hardware. This is clearly quite natural in software implementations of coherence protocols where the protocol is in programmable software handlers. For hardware-supported, fine-grained coherence, it turns out that the requirements placed on a programmable assist by different protocols are usually very similar. On the control side, they all need quick dispatch to a protocol handler, based on a transaction type, and support for efficient bit-field manipulation in tags. On the data side, they need high-bandwidth, low-overhead pipelined movement of data through the controller and network interface. The Sequent NUMA-Q discussed in Chapter 8 provides a pipelined, specialized programmable coherence controller, as does the Stanford FLASH design (Kuskin et al. 1994; Heinrich et al. 1994). Note that the controller being programmable doesn't alter the need for specialized hardware support for coherence; those issues remain the same as with a fixed protocol.

The protocol code that runs on the controllers in these fine-grained coherent machines operates in privileged mode so that it can communicate and use the physical addresses that it sees on the bus directly. Some researchers also advocate that users be allowed to write their own protocol handlers in user-level software so they can customize protocols to match the needs of individual applications much better than a predetermined library of system protocols can (Falsafi et al. 1994). While this can be advantageous, particularly on machines with less efficient communication architectures, it introduces several complications. For example, since the address translation is already done by the processor's memory management unit by the time a cache miss is detected, the assist sees only physical addresses on the bus. However, to maintain protection, user-level protocol software cannot be allowed access to these physical addresses. So if protocol software is to run on the assist at the other end at the user level, the physical addresses must be reverse translated back to virtual addresses before this software can use them. Such reverse translation (which

was also needed for Simple COMA systems for a different reason) requires further hardware support, increases latency, and is complicated to implement. In addition, since protocols have many subtle complexities related to correctness and deadlock and are difficult to debug, it is not clear how desirable it is to allow users to write protocols that may deadlock a shared machine.

Overcoming Physical Address Space Limitations

In a shared physical address space, the assist making a request sends the physical address of the location (or cache block) across the network, to be interpreted by the assist at the other end. While this has the advantage that the assist at the other end does not have to reverse translate addresses before accessing physical memory, a problem arises when the physical addresses generated by the processor may not have enough bits to serve as global addresses for the entire shared physical address space, as we saw for the CRAY T3D in Chapter 7 (the Alpha 21064 processor emitted only 32 bits of physical address, insufficient to address the 128 GB or 37 bits of physical memory in a 2,048-processor machine). In the T3D, segmentation through the annex registers was used to extend the address space, potentially introducing delays into the critical paths of memory accesses. The alternative is not to have a shared physical address space but to send virtual addresses across the network that will be retranslated to (potentially different) physical addresses at the other end. As we have seen, SVM and Simple COMA systems use this approach of implementing a shared virtual rather than physical address space, as do user-level programmable protocols.

One advantage of this approach is that now only the virtual addresses need to be large enough to index the entire shared (virtual) address space; physical addresses need only be large enough to address a given processor's main memory. A second advantage, seen earlier, is that each node manages its own address space and virtual-to-physical translations, so more flexible allocation and replacement policies can be used in main memory. However, this approach does require address translation at both ends of each communication.

9.6.2 Implementing Relaxed Memory Consistency in Software

The discussion of shared virtual memory schemes that exploit release consistency showed that such schemes can be either single writer or multiple writer and can propagate coherence information at either release or acquire operations. A range of techniques can be used to implement these schemes, successively adding complexity, enabling new schemes, and making the propagation of write notices and data lazier. The techniques are too complex and require too many structures to implement in hardware, and the problems they alleviate are much less severe in that case. However, they are quite well suited to software implementation. Although SVM is not currently a mainstream commercial technology, the techniques are valuable for understanding what is necessary for preserving different degrees of the laziness afforded by a relaxed consistency model. This section examines the techniques and their trade-offs.

The three basic functions for coherence protocols that were raised in Section 8.1 are answered somewhat differently in SVM schemes. To begin with, the protocol is invoked not only at data access faults as in the hardware cache coherence schemes but both at access faults (to find the necessary data) and at synchronization points (to communicate coherence information in write notices). The first two important functions—finding the source of coherence information and determining with which processors to communicate—depend on whether release-based or acquire-based coherence is used. In the former, we need to send write notices to all valid copies at a release, so we need a mechanism to keep track of the copies. In the latter, the acquirer communicates with only the last releaser of the synchronization variable and pulls all the necessary write notices from there to only itself, so there is no need to explicitly keep track of all copies. The last function is communication with the necessary nodes (or copies), which is typically done with point-to-point messages.

Since coherence information is not propagated at individual write faults but only at synchronization events, a new question arises: how do we determine for which pages write notices should be sent? In release-based schemes, since write notices are sent at every release to all currently valid copies, a node has only to send out write notices for writes that it performed since its previous release. All previous write notices in a causal sense (see Section 9.3.3) have already been sent to all relevant copies, either directly at the corresponding previous releases or indirectly through other processes. In acquire-based methods, we must ensure that causally necessary write notices, which may have been produced by many different nodes, will be seen even though the acquirer goes only to the previous releaser to obtain them. The releaser cannot simply send the acquirer the write notices it has produced since its last release or even the write notices it has ever produced, but must also send the causally related write notices that it has received from other nodes at its own previous acquires. In both release- and acquire-based cases, several mechanisms are available to reduce the number of write notices communicated and applied. These include version numbers and time stamps, and we shall see them as we go along. Protocols also vary in when they propagate write notices (and data) and when they apply, implementing different degrees of laziness within an acquire- or release-based approach, as we will see.

To understand the issues more clearly, let us first examine how we might implement single writer release consistency using both release-based and acquire-based approaches. Then we will do the same thing for multiple writer protocols.

Single Writer with Consistency at Release

The simplest way to maintain consistency is to send write notices at every release to all sharers of the pages that the releasing processor has written since its last release. In a single writer protocol, the copies can be kept track of by making the current owner of a page (the one with write permission) maintain the current sharing list and by transferring the list at ownership changes (when another node writes the page). At a release, a node sends write notices for all pages it has written to the nodes indicated on its sharing lists (see Exercise 9.31).

There are two performance problems with this scheme. First, since ownership transfer does not cause copies to be invalidated (read-only copies may coexist with one writable copy, unlike in hardware coherence schemes), a previous owner and the new owner may very well have the same nodes on their sharing lists. When both of them reach release points (whether of the same synchronization variable or different ones), they will both send invalidations to some of the same pages, so a page may receive multiple (unnecessary) invalidations. This problem can be solved by using a single designated place to keep track of which copies of a page have already been invalidated, for example, by using memory-based directories to keep track of sharing lists instead of maintaining them at the dynamically changing owners. The directory is looked up before write notices are sent, and invalidated copies are recorded at the directory so that multiple invalidations won't be sent to the same copy.

The second problem is that a release may invalidate a more recent copy of the page than the one that was written, which it needn't have done, as illustrated in Figure 9.20. (It won't invalidate the most recent copy at the current owner, so this is not a correctness problem.) This can be solved by associating a *version number* with each copy of a page. A node increments its version number for a page whenever it obtains ownership of that page from another node. Without directories, a processor will send write notices to all sharers at a release (since it doesn't know their version numbers) together with its version number for that page, but only the receivers that have smaller version numbers will actually invalidate their pages. With directories, the write notice traffic can be reduced as well, not just the number of invalidations applied and page faults experienced, by maintaining the version numbers of copies at the directory entry for the page and only sending write notices to copies that have lower version numbers than the releaser. Both ownership and write notice requests come to the directory, so this is easy to manage. Thus, using directories together with version numbers solves both of the preceding problems. However, this is still a release-based scheme, so invalidations may be sent out and applied earlier than necessary, causing unnecessary page faults.

Single Writer with Consistency at Acquire

Here is a simple way to use the fact that coherence activity is not needed until an acquire: the releaser still sends out write notices to all copies but does this only when the next acquire request from any process comes in, not at the release. This delays the sending of write notices. However, the incoming acquire must wait until the releaser has sent out the write notices and acknowledgments have been received, which is now in the critical path of the acquire operation. The best bet for such fundamentally release-based approaches would be to send write notices out eagerly at the release but wait for acknowledgments only before responding to the next incoming acquire request. This allows the propagation of write notices and acknowledgments to be overlapped with the computation done between the release and the next incoming acquire. Regardless of when write notices are propagated (which affects traffic), a receiving process may choose to apply them to pages as soon as they are

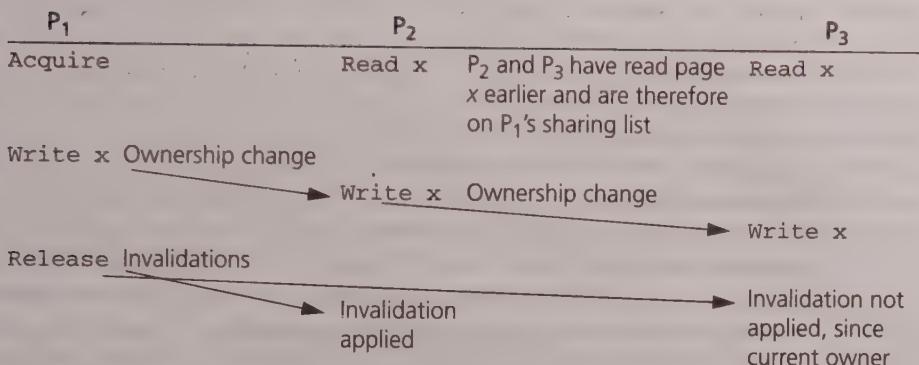


FIGURE 9.20 **Invalidation of a more recent copy in a simple single writer protocol.** Processor P₂ takes ownership from P₁ and then P₃ from P₂. But P₁'s release happens after all this. At the release, P₁ sends invalidations to both P₂ and P₃. P₂ applies the invalidation even though its copy is more recent than P₁'s (since it doesn't know) while P₃ does not apply the invalidation since it is the current owner.

received or at the next acquire that it performs, thus implementing different degrees of laziness.

Consider now the lazier, pull-based method of propagating consistency information only at an acquire, from the releaser to only that acquirer. The acquirer sends a request to the last releaser (the current holder) of the synchronization variable, whose identity it can obtain from a designated manager node for that variable. This is the only place from which the acquirer is obtaining information, yet it must see all writes that have happened before it in the causal order. With no additional support, the releaser must send to the acquirer all write notices that the releaser has either produced so far or received from others (at least since the last time it sent this acquirer write notices, if it keeps track of that). It cannot send only those that it has produced since the last release since it has no idea how many of the other necessary write notices the acquirer has already seen through previous acquires from other processes. The acquirer, too, must retain those write notices to pass on to the next acquirer.

Carrying around an entire history of write notices is obviously not a good idea. Version numbers, incremented at changes of ownership as before, can help reduce the number of invalidations applied (and hence access faults) if they are communicated along with the write notices, but they do not help reduce the number of write notices sent. The acquirer cannot communicate version numbers to the releaser to reduce traffic since it has no idea for which pages the releaser wants to send it write notices. And directories with version numbers don't help reduce the traffic either since the releaser would have to send the directory the history of write notices. In fact, what the acquirer wants is the write notices corresponding to all releases that precede it causally and that it hasn't already obtained through its previous acquires.

Keeping information at page level doesn't help to achieve this, since neither acquirer nor releaser knows which pages the other has seen write notices for and neither wants to send the information it knows for all pages. The solution is to establish a system of per-process or per-node virtual time, in which time-steps are demarcated by synchronization events. Conceptually, every node keeps track of the virtual time period up to which it has seen write notices from each other node. An acquire sends the previous releaser only this time vector; the releaser compares it with its own and sends the acquirer write notices corresponding to time periods that the releaser has seen but the acquirer hasn't. Since the partial orders to be satisfied for causality are based on synchronization events, associating time increments with synchronization events lets us represent these partial orders explicitly.

More precisely, the execution of every process is divided into a number of intervals, a new one beginning (and a local interval counter for that process incrementing) whenever the process successfully executes a release or an acquire. The intervals of different processes are partially ordered by the desired precedence relationships of causality discussed in Section 9.3.3: (1) intervals on a single process are totally ordered by the program order, and (2) an interval on process P precedes an interval on process Q if its release precedes Q 's acquire for that interval in a chain of release-acquire operations on the same variable. That is, intervals are also ordered by dependence order, which may not be statically determined. Since interval numbers are maintained and incremented locally per process, the partial order described in (2) does not mean that the acquiring process's interval number will necessarily be larger than the releasing process's interval number. What it does mean, and what we must ensure, is that if a releaser has seen write notices for interval 8 from process X , then the next (dynamic) acquirer of that synchronization variable should also have seen at least interval 8 from process X before it is allowed to complete its acquire. To keep track of what intervals a process has seen from other processes and hence preserve the partial orders, every process maintains a *vector time stamp* for each of its intervals (Keleher et al. 1994). Let V_i^P be the vector time stamp for interval i on process P . The number of elements in the vector V_i^P equals the number of processes. The entry for process P itself in V_i^P is equal to i . For any other process Q , the entry denotes the most recent interval of process Q that precedes interval i on process P in the partial orders. Thus, the vector time stamp indicates the most recent interval from every other process for which this process should have already received and applied write notices (through a previous acquire) by the time it enters interval i .

On an acquire, a process P needs to obtain from the last releaser R write notices pertaining to intervals (from any process) that R has seen before its release but the acquirer has not yet seen through a previous acquire. This is enough to ensure causality: any other intervals that P should have seen from other processes, it would have seen through previous acquires in its program order. P therefore sends its current vector time stamp (for its interval $i - 1$) to R , thus telling it which are the latest intervals from other processes it has seen before this acquire. R compares P 's incoming vector time stamp with its own, entry by entry, and piggybacks on its reply to P write notices for all intervals that are included as having been seen in R 's current time stamp but not in P 's (this is conservative since R 's current time stamp may have

seen more than R had at the time of the relevant release). Since P has now received these write notices, it sets its new vector time stamp (for the interval i that starts with the current acquire) to the pairwise maximum of R's vector time stamp and its own previous one. P is now up-to-date in the partial orders. This means that a process must retain the write notices that it has either produced or received from other processes until it is certain that no later acquirer will need them. This is unlike a release-based protocol in which write notices could be discarded by the releaser once they had been sent to all copies. It can lead to significant storage overhead and may require garbage collection techniques to keep the storage in check.

Multiple Writer with Release-Based Consistency

With multiple writers, the key new issue is the management and merging of data (e.g., diffs) from multiple writers. The type of protocol we use for write notices and consistency depends on how the propagation of data is managed; that is, whether diffs are maintained at the multiple writers or propagated at a release to a fixed home. In either case, with release-based schemes a process need only send write notices for the writes it has done since its previous release to all copies and wait for acknowledgments for them at the next incoming acquire, as in the single writer case. However, since there is no single owner (writer) of a page at a given time, we cannot rely on an owner having an up-to-date copy of the sharing list. We must either broadcast write notices or use a mechanism like a directory to keep track of copies.

The next question is, how does a process find the necessary data (diffs) when it incurs a page fault after an invalidation? If a home-based protocol is used to manage multiple writers, then this is easy: the page or diffs can be found at the home (a release must wait until the diffs reach the home before it completes so that a page fault following a dependent acquire is guaranteed to see the corresponding writes). If diffs are maintained at the writers (i.e., in a distributed form), the faulting process must know not only from where to obtain the diffs but also in what order they should be applied. This is because diffs for the same data may have been produced either in different intervals in the same process or in intervals on different processes but in the same causal chain of acquires and releases; when they arrive at a processor, they must be applied in accordance with the partial orders needed for causality. The locations of the diffs may be determined from the incoming write notices, but the order of application is difficult to determine without vector time stamps. However, vector time stamps obtain their full use only in acquire-based protocols, as we have seen. This is why, when homes are not used for data, simple directory and release-based (eager) multiple writer coherence schemes use updates rather than invalidations as the write notices: the diffs themselves are sent to the sharers at the release. Thus, the protocol and mechanisms used for consistency are changed from the single writer release-based case. Since a release waits for acknowledgments before completing, there is no ordering problem in applying diffs even though vector time stamps are not used. The diffs are guaranteed to reach processors exactly according to the desired partial order. This type of update-based, eager, multiple

writer protocol was used in the Munin system (Carter, Bennett, and Zwaenepoel 1991).

Consider the use of more sophisticated tracking mechanisms for consistency. Version numbers are not useful with multiple writer schemes. We can no longer update version numbers at ownership changes (since there are none) but only at release points. And, in that case, version numbers don't help us save write notices: since the releaser has just obtained the latest version number for the page at the release itself, there is no question of another node having a more recent version number for that page. In addition, because a releaser has to send only the write notices (or diffs) it has produced since its previous release, there is no need for vector time stamps with an update-based eager protocol. (Time stamps would have been needed to ensure that diffs were applied in the correct order if diffs were not sent out at a release but retained at the releaser in an invalidation-based protocol, as discussed previously.)

Multiple Writer with Acquire-Based Consistency

The issues and mechanisms for a multiple writer acquire-based protocol are very similar to the single writer acquire-based schemes—the main difference is in how the data is managed. With no special support, the acquirer must obtain from the last releaser all write notices that the releaser has produced or received since their last interaction, so large histories must be maintained and communicated. As in the single writer case, version numbers can help reduce the number of invalidations applied but not the number transferred. (With home-based schemes, the version number can be incremented every time a diff gets to the home, but the releaser must wait to receive this number before it satisfies the next incoming acquire; without homes, a separate version manager must be designated anyway for a page, which causes complications.) The best method is to use vector time stamps as described earlier. The vector time stamps manage the transfers of write notices (coherence information) for home-based schemes; for nonhome-based schemes, they also manage the obtaining and application of diffs (i.e., data) in the right order (the order dictated by the time stamps that came with the write notices).

To implement acquire-based LRC, then, a processor has to maintain many auxiliary data structures. These include the following:

- An array, indexed by process, for every page in its local memory, each entry of which is a list of write notices or diffs received from that process for that page.
- A separate single array, indexed by process, each entry of which is a pointer to a list of interval records. The entry for a process represents the intervals of that process for which the current process has already received write notices. An interval record points to the corresponding list of write notices, and each write notice points to its interval record.
- A free pool for creating diffs.

Since these data structures, especially diffs, may have to be kept around for a period that is determined by the partial precedence orders established at run time, they can limit the sizes of problems that can be run and the scalability of the

approach. Home-based approaches help reduce diff storage: diffs do not have to be retained at the releaser past the release, and the first array listed previously does not have to maintain lists of diffs received. Details of these mechanisms can be found in (Keleher et al. 1994; Zhou, Iftode, and Li 1996).

9.7

CONCLUDING REMARKS

The alternative approaches to supporting coherent replication in a shared address space discussed in this chapter raise many interesting sets of hardware/software trade-offs. Relaxed memory models increase the burden on application software to label programs correctly but allow compilers to perform their optimizations and hardware to exploit more low-level concurrency. COMA and related systems require greater hardware complexity but simplify the job of the programmer by reducing the importance of data placement. Their effect on performance depends greatly on the characteristics of the application (i.e., whether sharing misses or capacity misses dominate internode communication) and on the extent to which remote access latency and assist occupancy are increased. Finally, commodity-based approaches reduce system cost and provide a better incremental procurement model for users, but they often require substantially greater programming care to achieve good performance.

These alternative approaches are still controversial, and the trade-offs have not shaken out. While relaxed memory models are very useful for compilers, we will see in Chapter 11 that some modern processors are electing to implement sequential consistency at the hardware/software interface (or processor consistency in the case of the Intel Pentium Pro) with increasingly sophisticated alternative techniques to obtain overlap and hide latency. Contracts between the programmer and the system have also not been very well integrated into current programming languages and compilers. As for replication in main memory, full hardware support for COMA was implemented in the KSR1 (Frank, Burkhardt, and Rothnie 1993) but is not very popular in systems being built today because of its cost. However, approaches similar to Simple COMA are beginning to find acceptance in commercial products.

All-software approaches like page-grained SVM and fine-grained software access control have been demonstrated to achieve good performance at a relatively small scale for some classes of applications. Because they are very easy to build and deploy, they will likely be used on clusters of workstations and SMPs in several environments. However, the gap between these and the all-hardware systems is still quite large in programmability as well as in performance on a wide range of applications, more so than for message passing, and their scalability has not yet been demonstrated. The commodity-based approaches are still in the research stages, and it remains to be seen if they will become viable competitors to hardware cache-coherent machines for a large enough set of applications. It may be that the commoditization of hardware coherence assists and the methods to integrate them into memory systems will make these all-software approaches more marginal, for use largely in those environments that do not wish to purchase parallel machines but rather to use clusters of existing machines as shared address space multiprocessors.

when they are otherwise idle (or to develop early versions of parallel applications). At least in the short term, economic reasons might nonetheless provide all-software solutions with another role. Since vendors are likely to build tightly coupled systems with only a few tens or a hundred nodes, connecting these hardware-coherent systems together with a software coherence layer may be the most viable way to construct very large machines that still support the coherent shared address space programming model. While supporting this programming model on scalable systems with physically distributed memory is well established as a desirable way to build systems, only time will tell how these alternative approaches will play out.

9.8 EXERCISES

- 9.1 Why are update-based coherence schemes relatively incompatible with the sequential consistency memory model in a directory-based machine?
- 9.2 The Intel Paragon machine discussed in Chapter 7 has two processing elements per node, one of which always executes in kernel mode to support communication. Could this processor be used effectively as a programmable communication assist to support cache coherence at cache block granularity in hardware, like the programmable assist of the Stanford FLASH or the Sequent NUMA-Q?
- 9.3 In the Origin protocol discussed in Chapter 8, it would be possible for other read and write requests to come to the requestor that has outstanding invalidations before the acknowledgments for those invalidations come in.
 - a. Is this possible or a problem with delayed-exclusive replies? With eager-exclusive replies? If it is a problem, what is the simplest way to solve it?
 - b. Suppose you did indeed want to allow the requestor with invalidations outstanding to process incoming read and write requests. Consider write requests first, and construct an example where this can lead to problems. (Hint: consider the case where P_1 writes to a location A, P_2 writes to location B and then writes a flag, and P_3 spins on the flag and then reads the value of location B.) How would you allow the incoming write to be handled but still maintain correctness? Is it easier if invalidation acknowledgments are collected at the home (as in the Stanford FLASH machine) or at the requestor (as in the Origin)?
 - c. Now answer the questions in part (b) for incoming read requests.
 - d. What if you were using an update-based protocol? What complexities arise in allowing an incoming request to be processed for a block that has updates outstanding from a previous write, and how might you solve them?
 - e. Overall, would you choose to allow incoming requests to be processed while invalidations or updates are outstanding, or deny them?
- 9.4 Suppose a block needs to be written back while invalidations are pending for it. Can this lead to problems, or is it safe? If it is problematic, how might you address the problem?

- 9.5 Are eager-exclusive replies useful with an underlying SC model? Are they at all useful if the processor itself allows out-of-order completion of memory operations, unlike the MIPS R10000?
- 9.6 Do the trade-offs between collecting acknowledgments at the home and at the requestor change if eager-exclusive replies are used instead of delayed-exclusive replies?
- 9.7 If the compiler reorders accesses according to WO and the processor's memory model is SC, what is the consistency model at the programmer's interface? What if the compiler is SC and does not reorder memory operations but the processor implements RMO?
- 9.8 In addition to reordering memory operations (reads and writes) as discussed in Example 9.1 in this chapter, register allocation by a compiler can also eliminate memory accesses entirely. Consider the following example code fragment. Show how register allocation can violate SC. Can a uniprocessor compiler do this? How would you prevent it in the compiler you normally use?

P_1	P_2
$A = 1$	while (flag == 0);
$flag = 1$	$u = A$

- 9.9 Consider all the system specifications discussed in this chapter. Arrange them in order of weakness; that is, draw arcs between models such that an arc from model A to model B indicates that A is stronger than B (i.e., any execution that is correct under A is also correct under B but not necessarily vice versa).
- 9.10 Which of PC and TSO is better suited to update-based directory protocols, and why?
- 9.11 Can you describe a more relaxed system specification than release consistency without explicitly associating data with synchronization, as in entry consistency? Does it require additional programming care beyond RC?
- 9.12 Can you describe a looser set of sufficient conditions for WO? For RC?
- 9.13 Using the fence operations provided by the DEC Alpha and Sparc RMO consistency specifications, describe how you would need to insert these operations to ensure that a program obeys each of the following models: RC, WO, PSO, TSO, and SC? Which ones do you expect to be implemented efficiently in this way and which ones not, and why?
- 9.14 A *write-fence* operation (like the write-memory barrier in the Alpha architecture) stalls subsequent write operations until all of the processor's previous write operations have completed. A *full fence* stalls the processor until all of its previous memory operations have completed.
- Insert the minimum number of fence instructions into the following code to make it sequentially consistent, assuming that otherwise the system does not preserve any program orders. Don't use a full fence when a write fence will suffice.

```

ACQUIRE LOCK1
LOAD A
STORE A
RELEASE LOCK1
LOAD B
STORE B
ACQUIRE LOCK1
LOAD C
STORE C
RELEASE LOCK1

```

b. Repeat part (a) to guarantee release consistency.

- 9.15 Given the following code segments, what combination of values for (u,v,w,x) are not allowed by SC? In each case, do the IBM 370, TSO, PSO, PC, RC, and WO models preserve SC semantics without any ordering instructions or labels, or do they not? (The IBM 370 consistency model is much like TSO, except that it does not allow a read to return the value written by a previous write in program order until that write has completed.) If not, insert the necessary fence operations to make them conform to SC. Assume that all variables had the value 0 before this code fragment was reached.

a.

P_1	P_2
$A = 1$	$B = 1$
$u = A$	$v = B$
$w = B$	$x = A$

b.

P_1	P_2
$A = 1$	$B = 1$
$C = 1$	$C = 2$
$u = C$	$v = C$
$w = B$	$x = A$

- 9.16 Consider a two-level coherence protocol with snooping-based SMPs connected by a memory-based directory protocol, using release consistency. While invalidation acknowledgments are still pending for a write to a memory block, is it okay to supply the data to another processor in (a) the same SMP node, or (b) a different SMP node? Justify your answers and state any assumptions.
- 9.17 Can a program that is not properly labeled run correctly on a system that supports release consistency? If so, how, and if not, why not?

- 9.18 Why are there four flavor bits for memory barriers in the Sun Sparc V9 specification? Why not just two bits, one to wait for all previous writes to complete and another for previous reads to complete?
- 9.19 To communicate labeling information to the hardware (i.e., that a memory operation is labeled as an acquire or a release), there are two options: one is to associate the label with the address of the location; the other is to associate the label with the specific operation in the code. What are the trade-offs between the two?
- 9.20 Two processors P_1 and P_2 are executing the following code fragments under the sequential consistency (SC) and release consistency (RC) models.

P_1	P_2
LOCK (L1)	LOCK (L1)
$A = 1$	$x = A$
$B = 2$	$y = B$
UNLOCK (L1)	$x1 = A$
	UNLOCK (L1)
	$x2 = B$

Assume an architecture where both read and write misses take 100 cycles to complete. However, you can assume that accesses that are allowed to be overlapped under the consistency model are indeed fully overlapped. Acquiring a free lock from another processor or unlocking a lock takes 100 cycles, and no overlap is possible with lock-unlock operations from the same processor. Assume all the variables and locks are initially uncached and all locks are unlocked, that all memory locations are initialized to 0, and that all memory locations are distinct and map to different indices in the caches (i.e., different cache lines).

- a. What are the possible outcomes for x and y under SC? Under RC?
 - b. Assume P_1 gets the lock first. After how much time from the start of P_1 's lock operation will P_2 complete all its operations while satisfying the sufficient conditions for SC described in Chapter 5? What if it satisfies the sufficient conditions for RC described in this chapter?
- 9.21 Given the following code fragment, we want to compute its execution time under various memory consistency models. Assume a processor architecture with an arbitrarily deep write buffer. All instructions take 1 cycle, ignoring memory system effects. Both read and write misses take 100 cycles to complete (i.e., to perform globally). Locks are cacheable and loads are nonblocking. Assume all the variables and locks are initially uncached and all locks are unlocked. Further assume that once a line is brought into the cache it does not get invalidated for the duration of the code's execution. All memory locations referenced here are distinct; furthermore, they all map to different cache lines.

```

LOAD A
STORE B
LOCK (L1)
STORE C
LOAD D
UNLOCK (L1)
LOAD E
STORE F

```

- a. If the sufficient conditions for sequential consistency are maintained, how many cycles will it take to execute this code?
- b. Repeat part (a) for weak ordering.
- c. Repeat part (a) for release consistency.
- 9.22 The following code is executed on an aggressive dynamically scheduled but single-issue processor. The processor can have multiple outstanding operations, the cache allows for multiple outstanding misses, and the write buffer can hide store latencies (of course, these features may only be used if allowed by the memory consistency model).

Processor 1: <pre> sendSpecial(int value) { A = 1 LOCK(L); C = D*3; E = F*10; G = value; READY = 1; UNLOCK(L); } </pre>	Processor 2: <pre> receiveSpecial() { LOCK(L); if (READY) { D = C+1; F = E*G; } UNLOCK (L); } </pre>
---	--

Assume that locks are noncacheable and are acquired either 50 cycles from an issued request or 20 cycles from the time a release completes (whichever is later). A release takes 50 cycles to complete. Read hits take 1 cycle to complete and writes take 1 cycle to put into the write buffer. Read misses to shared variables take 50 cycles to complete. Writes take 50 cycles to complete. The write buffer on the processors is sufficiently large that it never fills completely. Only count the latencies of reads and writes to shared variables (those listed in capitals) and the locks. All shared variables are initially uncached with a value of 0. Assume that processor 1 obtains the lock first.

- a. Under SC, how many cycles will it take from the time processor 1 enters the `sendSpecial()` routine to the time that processor 2 leaves `receive`

- `Special()`? Make sure that you justify your answer. Also make sure to note the issue and completion time of each synchronization event.
- b. How many cycles will it take to return from `receiveSpecial()` under release consistency?
- 9.23 In eager release consistency, diffs, like write notices, are created and also propagated at release time, whereas in one form of lazy release consistency they are created at release time but propagated only at acquire time (that is, when the acquire synchronization request comes to the processor). In general, data may be propagated with varying degrees of laziness just like write notices.
- Describe some other possibilities for when diffs might be created, propagated, and applied in all-software lazy release consistency (think of release time, acquire time, or access fault time). What is the laziest scheme you can design?
 - What complications does each lazier scheme cause in implementation? Which scheme would you choose to implement and why?
- 9.24 Delaying the propagation of invalidations until a release point or even until the next acquire point (as in lazy release consistency) can be done in hardware-coherent systems as well. Why is LRC not used in hardware-coherent systems? Would delaying invalidations until a release (not an acquire) be advantageous?
- 9.25 Suppose you had a co-processor to perform the creation and application of diffs in an all-software SVM system and therefore did not have to perform this activity on the main processor. Considering eager release consistency and the lazy variants you designed in Exercise 9.23, comment on the extent to which protocol processing activity can be overlapped with computation on the main processor. Draw timelines to show what can be done on the main processor and on the co-processor. Do you expect the savings in performance to be substantial? What do you think would be the major benefit and major implementation complexity of having all protocol processing and management performed on the co-processor?
- 9.26 Why is garbage collection more important and more complex in TreadMarks-style lazy release consistency than in eager release consistency? What about in home-based lazy release consistency? Design a scheme for periodic garbage collection (discussing both when and how), and discuss the complications.
- 9.27 In systems like Blizzard-S or Shasta that instrument read and write operations in software to provide fine-grained access control, a key performance goal is to reduce the overhead of instrumentation. Describe some techniques that you might use to do this. To what extent do you think the techniques can be automated in a compiler or a tool for executable instrumentation?
- 9.28 When messages (e.g., page requests or lock requests) arrive at a node in a software shared memory system, whether fine grained or coarse grained, there are two major ways to handle them in the absence of a programmable communication assist. One is to interrupt the main processor, and the other is to have the main processor poll for messages.
- What are the major trade-offs between the two methods?

- b. How would you manage the polling by the main processor; in particular, when would you poll?
 - c. Which do you expect to perform better for page-based shared virtual memory and why? For fine-grained software shared memory? What application characteristics would most influence your decision?
 - d. What new issues arise and how might the trade-offs change if each node is an SMP rather than a uniprocessor?
 - e. How do you think you would organize message handling with SMP nodes? That is, where would incoming messages be handled, and how would message notification be managed?
- 9.29 List all the trade-offs you can think of between LRC based on diffs (not home based) versus based on automatic update (home based). Which do you think would perform better? What about home-based LRC based on diffs versus based on automatic update?
- 9.30 Is a properly labeled program guaranteed to run correctly under LRC? Under ERC? Under RC? Is a program that runs correctly under ERC guaranteed to run correctly under LRC? Is it guaranteed to be properly labeled (i.e., can a program that is not properly labeled run correctly under ERC)? Is a program that runs correctly under LRC guaranteed to run correctly under ERC?
- 9.31 Consider a single writer release-based protocol. On a release, does a node need to obtain the up-to-date sharing list for each page it has modified since the last release from the current owner or just send write notices to nodes on its version of the sharing list for each such page? Explain why.
- 9.32 Consider page version numbers without directories. Does this avoid the problem of sending multiple invalidates to the same copy in a single writer release-based protocol? Explain why, or give a counterexample.
- 9.33 Trace the path of a write reference in (a) a pure CC-NUMA, (b) a flat COMA, (c) an SVM with automatic update, (d) an SVM protocol without automatic update, and (e) a simple COMA. (Hint: see how it was done for read references in the chapter.)
- 9.34 You are performing an architectural study using four applications: Ocean, LU, an FFT that uses a matrix transposition between local calculations on rows (see Exercise 8.23), and Barnes-Hut. For each application, answer the following questions, assuming a page-grained SVM system (these questions were asked for a CC-NUMA system in Chapter 8):
- a. What modifications or enhancements in data structuring or layout would you use to ensure good interactions with the extended memory hierarchy?
 - b. Methodologically, what are the interactions with cache size and with granularities of allocation, coherence, and communication that you would be particularly careful to represent or not represent? What new ones become important in SVM systems that were not so important in CC-NUMA, and which ones become less important relative to others?

- c. Are the interactions with cache size as important in SVM as they are in CC-NUMA? If they are of different importance, say why.
- 9.35 Consider the FFT calculation with a matrix transpose described in Exercise 8.23. Suppose you are running this program on a page-based SVM system using an all-software, home-based multiple writer protocol.
- Would you rather use the method in which a processor reads locally allocated data and writes remotely allocated data to implement the transpose or the one in which the processor reads remote data and writes local data?
 - Now suppose you have hardware support for automatic update propagation to further speed up your home-based protocol. How does this change the trade-off, if at all?
 - What protocol artifacts limit performance, and what protocol optimizations can you think of that would substantially increase the performance of one scheme or the other?

Interconnection Network Design

We have seen throughout this book that scalable high-performance interconnection networks lie at the core of parallel computer architecture. Our generic parallel machine has three basic components: the processor-memory nodes, the node-to-network interface, and the network that holds it all together. The previous chapters have given a general understanding of the requirements placed on the interconnection network of a parallel machine; this chapter examines in depth the design of high-performance interconnection networks for parallel computers. These networks share basic concepts and terminology with local area networks (LANs) and wide area networks (WANs), which may be familiar to many readers, but the design trade-offs are quite different because of the dramatic difference of time scale.

Parallel computer networks are a rich and interesting topic because they have so many facets, but this richness also makes the topic difficult to understand in an overall sense. For example, parallel computer networks are generally wired together in a regular pattern. The topological structure of these networks has elegant mathematical properties, and there are deep relationships between these topologies and the fundamental communication patterns of important parallel algorithms. However, pseudo-random wiring patterns have a different set of nice mathematical properties and tend to have more uniform performance without really good or really bad communication patterns. There is a wide range of interesting trade-offs to examine at this abstract level, and a huge volume of research papers focus completely on this aspect of network design. On the other hand, passing information between two independent asynchronous devices across an electrical or optical link presents a host of subtle engineering issues. These are the kinds of issues that give rise to major standardization efforts. From yet a third point of view, the interactions between multiple flows of information competing for communications resources have subtle performance effects that are influenced by a host of factors. The performance modeling of networks is another huge area of theoretical and practical research. Real network designs address issues at each of these levels. The goal of this chapter is to provide a holistic understanding of the many facets of parallel computer networks so that the reader may see the diverse network design space within the larger problem of parallel machine design as driven by application demands.

As with all other aspects of design, network design involves understanding trade-offs and making compromises so that the solution is near optimal in a global sense rather than optimized for a particular component of interest. The performance

impact of the many interacting facets can be quite subtle. Moreover, no clear consensus exists in the field on the appropriate cost model for networks since trade-offs can be made between very different technologies; for example, bandwidth of the links may be traded against complexity of the switches. It is also very difficult to establish a well-defined workload against which to assess network designs since program requirements are influenced by every other level of the system design before being presented to the network. This is the kind of situation that commonly gives rise to distinct “design camps” and rather heated debates, which often neglect to bring out the differences in base assumptions. In the course of developing the concepts of computer networks, this chapter points out how the choice of cost model and of workload lead to various important design points, which reflect key technological assumptions.

Previous chapters have illuminated the factors that drive network design. The communication-to-computation ratio of the program places a requirement on the data bandwidth the network must deliver if the processors are to sustain a given computational rate. However, this load varies considerably between programs; the flow of information may be physically localized or dispersed, and it may be bursty in time or fairly uniform. In addition, the waiting time of the program is strongly affected by the latency of the network, and the time spent waiting affects the bandwidth requirement. We have seen that different programming model implementations tend to communicate at different granularities (which impacts the size of data transfers seen by the network) and that they use different protocols at the network transaction level to realize the higher-level programming model.

This chapter begins with a set of basic definitions and concepts that underlie all networks. Simple models of communication latency and bandwidth are developed in Section 10.2 to reveal the core differences in network design styles. The key components that are assembled to form networks are described concretely in Section 10.3. Section 10.4 explains the rich space of interconnection topologies in a common framework, and Section 10.5 ties the design trade-offs back to cost, latency, and bandwidth under basic workload assumptions. Section 10.6 explains the various ways that messages are routed within the topology of the network in a manner that avoids deadlock and describes the further impact of routing on communication performance. Section 10.7 dives deeper into the hardware organization of the switches that form the basic building block of networks in order to provide a more precise understanding of the engineering trade-offs of various options and the mechanics underlying the more abstract network concepts. Then Section 10.8 explores the alternative approaches to flow control within a network. With this grounding in place, Section 10.9 brings together the entire range of issues in a collection of case studies and examines the transition of parallel computer network technology into other network regimes, including the emerging system area networks (SANs).

10.1 BASIC DEFINITIONS

The job of an interconnection network in a parallel machine is to transfer information from any source node to any desired destination node, in support of the net-

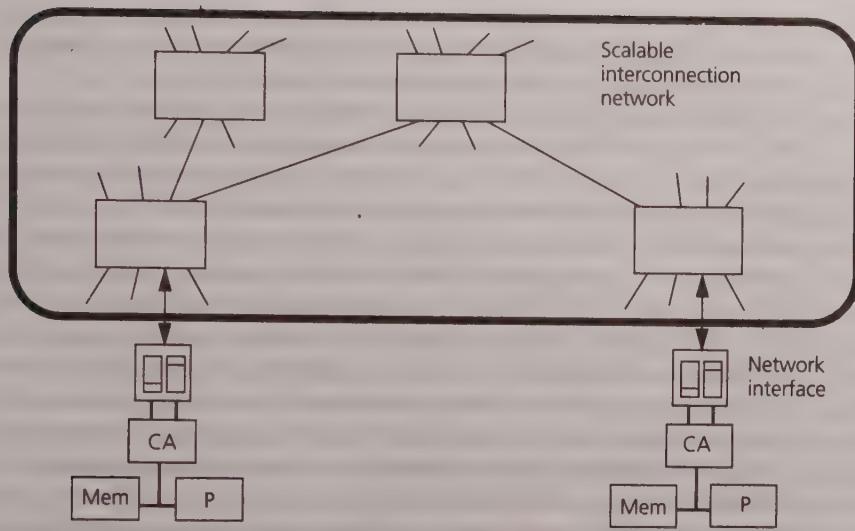


FIGURE 10.1 Generic parallel machine interconnection network. The communication assist initiates network transactions on behalf of the processor or memory controller through a network interface that causes information to be transmitted across a sequence of links and switches to a remote node where the network transaction takes place.

work transactions that are used to realize the programming model. It should accomplish this task with as small a latency as possible, and it should allow a large number of such transfers to take place concurrently. In addition, it should be inexpensive relative to the cost of the rest of the machine.

The expanded diagram for our generic large-scale parallel architecture in Figure 10.1 illustrates the structure of an interconnection network in a parallel machine. The communication assist on the source node initiates network transactions by pushing information through the network interface (NI). These transactions are handled by the communication assist, processor, or memory controller on the destination node, depending on the communication abstraction that is supported.

The network is composed of links and switches that provide a means to route the information from the source node to the destination node. A *link* is essentially a bundle of wires or fibers that carries an analog signal. For information to flow along a link, a transmitter converts digital information at one end into an analog signal that is driven down the link and converted back into digital symbols by the receiver at the other end. The *physical protocol* for converting between streams of digital symbols and an analog signal forms the lowest layer of the network design. The transmitter, link, and receiver collectively form a *channel* for digital information flow between switches (or NIs) attached to the link. The *link-level protocol* segments the stream of symbols crossing a channel into larger logical units, called *packets* or *messages*, that are interpreted by the switches in order to steer each unit arriving on an input channel to the appropriate output channel. Processing nodes communicate

across a sequence of links and switches. The *node-level protocol* embeds commands for the remote communication assist within the packets or messages exchanged between the nodes to accomplish network transactions.

Formally, a parallel machine interconnection network is a graph, where the vertices V are processing hosts or switch elements connected by communication channels $C \subseteq V \times V$. A *channel* is a physical link between host or switch elements, including a buffer to hold data as it is being transferred. It has a width w and a signaling rate $f = 1/\tau$ (for cycle time τ), which together determine the *channel bandwidth* $b = wf$. The amount of data transferred across a link in a cycle is called a physical unit, or *phit*.¹ Switches connect a fixed number of input channels to a fixed number of output channels; this number is called the *switch degree*. Hosts typically connect to a single switch but can be multiply connected with separate channels. Messages are transferred through the network from a source host node to a destination host along a path, or *route*, comprised of a sequence of channels and switches.

A useful analogy to keep in mind is a roadway system composed of streets and intersections. Each street has a speed limit and a number of lanes, determining its peak bandwidth. It may be either unidirectional (one-way) or bidirectional. Intersections allow travelers to switch among a fixed number of streets. In each trip, a collection of people travel from a source location along a route to a destination. They may use any of many potential routes, many modes of transportation, and many different ways of dealing with traffic encountered en route. A very large number of such trips may be in progress in a city concurrently, and their respective paths may cross or share segments of the route.

A network is characterized by its topology, routing algorithm, switching strategy, and flow control mechanism.

- The *topology* is the physical interconnection structure of the network graph; this may be regular, as with a two-dimensional grid (typical of many metropolitan centers), or it may be irregular. Most parallel machines employ highly regular networks. A distinction is often made between direct and indirect networks; *direct* networks have a host node connected to each switch whereas *indirect* networks have hosts connected only to a specific subset of the switches, which form the edges of the network. Many machines employ mixed strategies, so the more critical distinction is between the two types of nodes: hosts generate and remove traffic whereas switches only move traffic along.
- The *routing algorithm* determines which routes messages may follow through the network graph. The routing algorithm restricts the set of possible paths to a smaller set of legal paths. There are many different routing algorithms, providing different guarantees and offering different performance trade-offs. For example, continuing the traffic analogy, a city might eliminate gridlock by leg-

1. Since many networks operate asynchronously rather than being controlled by a single global clock, the notion of a network “cycle” is not as widely used as in dealing with processors. We could equivalently define the network cycle time as the time to transmit the smallest physical unit of information, a phit. For parallel architectures, it is convenient to think about the processor cycle time and the network cycle time in common terms. Indeed, the two technological regimes are becoming increasingly similar.

isolating that cars must travel east-west before making a single turn north or south toward their destination rather than being allowed to zigzag across town. We will see that this does indeed eliminate deadlock, but it limits a driver's ability to avoid traffic en route. In a parallel machine, we are only concerned with routes from a host to a host.

- The *switching strategy* determines how the data in a message traverses its route. There are basically two switching strategies. In *circuit switching*, the path from the source to the destination is established and reserved until the message is transferred over the circuit. (This strategy is like reserving a parade route; it is good for moving a lot of people through, but advanced planning is required and it tends to be unpleasant for any traffic that might cross or share a portion of the reserved route, even when the parade is not in sight. It is also the strategy used in phone systems, which establish a circuit through possibly many switches for each call.) The alternative is *packet switching*, in which the message is broken into a sequence of packets. A packet contains routing and sequencing information as well as data. Packets are individually routed from the source to the destination. (The analogy to traveling as small groups in individual cars is obvious.) Packet switching typically allows better utilization of network resources because links and buffers are only occupied while a packet is traversing them.
- The *flow control mechanism* determines when the message, or portions of it, moves along its route. In particular, flow control is necessary whenever two or more messages attempt to use the same network resource (e.g., a channel) at the same time. One of the traffic flows could be stalled in place, shunted into buffers, detoured to an alternate route, or simply discarded. Each of these options places specific requirements on the design of the switch and influences other aspects of the communication subsystem. (Discarding traffic is clearly unacceptable in our traffic analogy.) The minimum unit of information that can be transferred across a link and either accepted or rejected is called a *flow control unit*, or *flit*. It may be as small as a phit or as large as a packet or message.

To illustrate the difference between a phit and a flit, consider the nCUBE case study in Section 7.1.4. The links are a single bit wide, so a phit is 1 bit. However, a switch accepts incoming messages in chunks of 36 bits (32 bits of data plus 4 parity bits). It only allows the next 36 bits to come in when it has a buffer to hold it, so the flit is 36 bits. In many more recent machines, such as the T3D, the phit and flit are the same.

An important property of a topology is the *diameter* of a network, which is the length of the maximum shortest path between any two nodes. The *routing distance* between a pair of nodes is the number of links traversed en route; this is at least as large as the shortest path between the nodes and may be larger. The *average distance* is simply the average of the routing distance over all pairs of nodes; this is also the expected distance between a random pair of nodes. In a direct network, routes must be provided between every pair of switches, whereas in an indirect network, it is

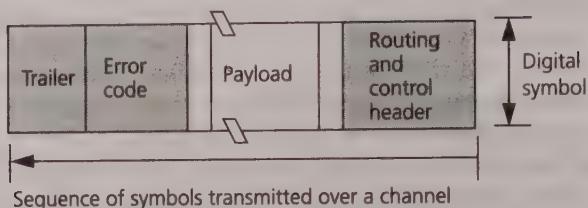


FIGURE 10.2 Typical packet format. A packet forms the logical unit of information that is steered along a route by switches. It is comprised of three parts: a header, a data payload, and a trailer. The header and trailer are interpreted by the switches as the packet progresses along the route, but the payload is not. The node-level protocol is carried in the payload.

only required that routes are provided between hosts. A network is *partitioned* if a set of links or switches are removed such that some hosts are no longer connected by routes.

Most of the discussion in this chapter centers on packets because packet switching is used in most modern parallel machine networks. Where specific important properties of circuit switching arise, they are pointed out. A *packet* is a self-delimiting sequence of digital symbols and logically consists of three parts, illustrated in Figure 10.2: a header, a payload, and a trailer. The *header* is the front of the packet and usually contains the routing and control information so that the switches and network interface can determine what to do with the packet as it arrives. The *payload* is the part of the packet containing data transmitted across the network. The *trailer* is the end of the packet and typically contains the error-checking code so that it can be generated as the message spools out onto the link. The header may also have a separate error-checking code.

The two basic mechanisms for building abstractions in the context of networks are encapsulation and fragmentation. *Encapsulation* involves carrying higher-level protocol information in an uninterpreted form within the message format of a given level. *Fragmentation* involves splitting the higher-level protocol information into a sequence of messages at a given level. Although these basic mechanisms are present in any network, the layers of abstraction in parallel computer networks tend to be much shallower than in, say, the Internet and are designed to fit together very efficiently. To make these notions concrete, observe that the header and trailer of a packet form an envelope that is interpreted by the switches and encapsulates the data payload. Information associated with the node-level protocol is contained within this payload. For example, a read request is typically conveyed to a remote memory controller in a single packet, and the cache line response is a single packet. The memory controllers are not concerned with the actual route followed by the packet or with the format of the header and trailer. At the same time, the network is not concerned with the format of the remote read request within the packet payload. A large bulk data transfer would not typically be carried out as a single packet; instead, it would be fragmented into several packets. Each would need to contain information to indicate where its data should be deposited or which fragment in the

overall data sequence it represents. Dropping down a layer, an individual packet is fragmented at the link level into a series of symbols that are transmitted in order across the link, so there is no need for sequencing information. This situation in which higher-level information is carried within an envelope, or multiple such envelopes, that is interpreted by the lower-level protocol occurs at every level (or layer) of network design.

10.2

BASIC COMMUNICATION PERFORMANCE

There is much to understand on each of the four major aspects of network design, but before going into these aspects in detail it is useful to have a general understanding of how they interact to determine the performance and functionality of the overall communication subsystem. Building on the brief discussion of networks in Chapter 7, let us look at performance from the latency and bandwidth perspectives.

10.2.1

Latency

To establish a basic performance model for understanding networks, we may expand the model for the communication time that we have used since Chapter 1. The time to transfer n bytes of information from its source to its destination has four components, as follows.

$$\text{Time}(n)_{S-D} = \text{Overhead} + \text{Routing Delay} + \text{Channel Occupancy} + \text{Contention Delay} \quad (10.1)$$

The *overhead* associated with getting the message into and out of the network on the ends of the actual transmission has been discussed extensively in dealing with the node-to-network interface in previous chapters. We have seen machines that are designed to move cache-line-sized chunks and others that are optimized for large-message DMA transfers. As to the remaining components, the routing delay and channel occupancy are effectively lumped together in previous chapters as the *unloaded latency* of the network for typical message sizes, and contention has been largely ignored. These other components are the focus of this chapter.

The *channel occupancy* provides a convenient lower bound on the communication latency, independent of where the message is going or what else is happening in the network. As we look at network design in more depth, we see below that the occupancy of each link is influenced by the channel width, the signaling rate, and the amount of control information, which is in turn influenced by the topology and routing algorithm. Whereas previous chapters were concerned with the channel occupancy seen from “outside” the network—the time to transfer the message across the bottleneck channel in the route—the view from within the network is that a channel occupancy is associated with each step along the route. The communication assist is occupied for a period of time accepting the communication request from the processor or memory controller and spooling a packet into the network.

Each channel the packet crosses en route is occupied for a period of time by the packet, as is the destination communication assist.

For example, an issue that we need to be aware of is the efficiency of the packet encoding. The packet envelope increases the occupancy because header and trailer symbols are added by the source and stripped off by the destination. Thus, for a payload of size n the occupancy of a channel is

$$\frac{n + n_E}{b}$$

where n_E is the size of the envelope and b is the raw bandwidth of the channel. This issue is addressed in the “outside view” by specifying the *effective bandwidth* of the link, derated from the raw bandwidth by

$$\frac{n}{n + n_E}$$

at least for fixed-size packets. However, within the network, packet efficiency remains a design issue. The effect is more pronounced with small packets, but it also depends on how routing is performed.

The *routing delay* is seen from outside the network as the time to move a given symbol, say, the first bit of the message, from the source to the destination. Viewed from within the network, each step along the route incurs a routing delay that accumulates into the delay observed from the outside. The routing delay is a function of the number of channels on the route, called the *routing distance*, h , and the delay Δ incurred at each switch as part of selecting the correct output port. (It is convenient to view the node-to-network interface as contributing to the routing delay like a switch.) The routing distance depends on the network topology, the routing algorithm, and the particular pair of source and destination nodes. The overall delay is strongly affected by switching and routing strategies.

With packet-switched, *store-and-forward* routing, the entire packet is received by a switch before it is forwarded on the next link, as illustrated in Figure 10.3(a). This strategy is used in most wide area networks and was used in several early parallel computers. The unloaded network latency for an n -byte packet, including envelope, with store-and-forward routing is

$$T_{sf}(n, h) = h \left(\frac{n}{b} + \Delta \right) \quad (10.2)$$

where Δ is the additional routing delay per hop.

Equation 10.2 would suggest that the network topology is paramount in determining network latency since the topology fundamentally determines the routing distance, h . In fact, the story is more complicated.

First, consider the switching strategy. With circuit switching, we expect a delay proportional to h to establish the circuit, configure each of the switches along the route, and inform the source that the route is established. After this time, the data should move along the circuit in time n/b plus an additional small delay propor-

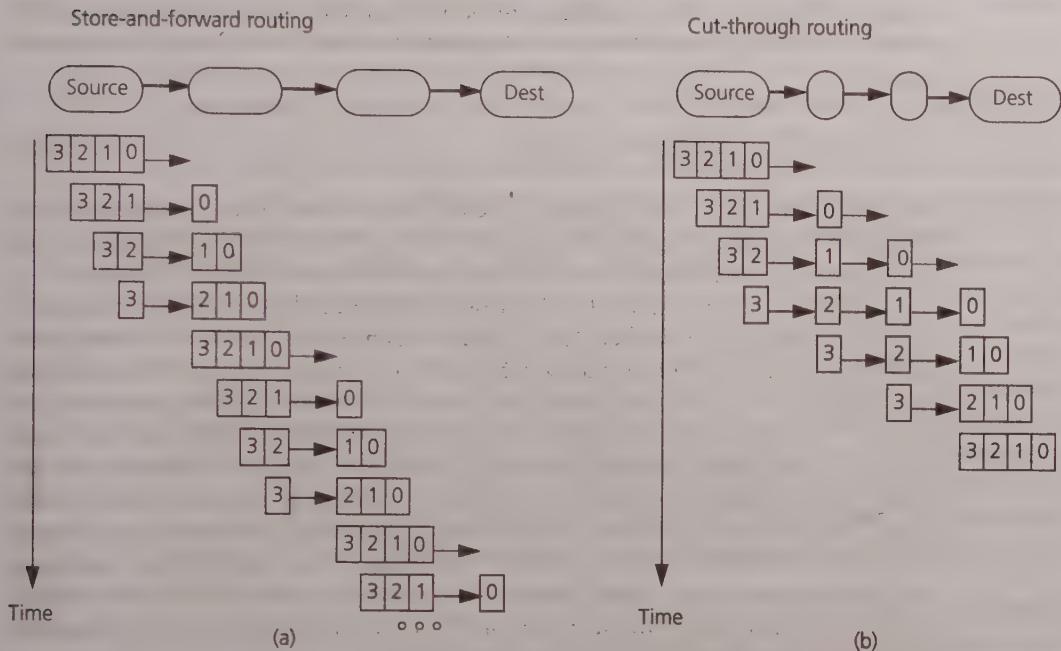


FIGURE 10.3 Store-and-forward versus cut-through routing for packet-switched networks. A four-flit packet traverses three hops from source to destination under store-and-forward and cut-through routing. Cut-through achieves lower latency by making the routing decision (gray arrow) on the first flit and pipelining the packet through a sequence of switches. Store-and-forward accumulates the entire packet before routing it toward the destination.

tional to h . Thus, the unloaded latency in units of the network cycle time, τ , for an n -byte message traveling distance h in a circuit-switched network is

$$T_{cs}(n, h) = \frac{n}{b} + h\Delta \quad (10.3)$$

In Equation 10.3, the setup and routing delay is an additive term, independent of the size of the message. Thus, as the message length increases, the routing distance, and hence the topology, becomes an insignificant fraction of the unloaded communication latency. Circuit switching is traditionally used in telecommunications networks since the call setup is short compared to the duration of the call. It is used in a minority of parallel computer networks, including the Meiko CS-2 and the BBN Butterfly. One important difference is that in parallel machines the circuit is established by routing the message through the network and holding open the route as a circuit. The more traditional approach is to compute the route on the side, configure the switches, and then transmit information on the circuit.

It is also possible to retain packet switching and yet reduce the unloaded latency from that of naive store-and-forward routing. The key concern with Equation 10.2 is that the delay is the product of the routing distance and the occupancy for the full

message. However, a long message can be fragmented into several small packets, which flow through the network in a pipelined fashion. In this case, the unloaded latency is

$$T_{sf}(n, h, n_p) = \frac{n - n_p}{b} + h \left(\frac{n_p}{b} + \Delta \right) \quad (10.4)$$

where n_p is the size of the fragments. The effective routing delay is proportional to the packet size rather than the message size. This is basically the approach adopted (in software) for traditional data communication networks such as the Internet.

In parallel computer networks, the idea of pipelining the routing and communication is carried much further. Most parallel machines use packet switching with *cut-through* routing, in which the switch makes its routing decision after inspecting only the first few phits of the header and allows the remainder of the packet to "cut through" from the input channel to the output channel, as indicated by Figure 10.3(b), so the transmission of even a single packet is pipelined. (If we may return to our vehicle analogy, cut-through routing is like what happens when a train encounters a switch in the track. The first car is directed to the proper output track and the remainder follow along behind. By contrast, store-and-forward routing is like what happens at the station, where all the cars in the train arrive and stop before the first proceeds toward the next station.) For cut-through routing, the unloaded latency has a form similar to the circuit switch case

$$T_{ct}(n, h) = \frac{n}{b} + h\Delta \quad (10.5)$$

although the routing coefficient, Δ , may differ since the mechanics of the process are rather different. Observe that with cut-through routing a single message may occupy the entire route from the source to the destination, much like circuit switching. The head of the message establishes the route as it moves toward its destination, and the route clears as the tail moves through.

The preceding discussion of communication latency addresses a message flowing from source to destination without running into traffic along the way. In this unloaded case, the network can be viewed simply as a pipeline, with a start-up cost, pipeline depth, and time per stage. The different switching and routing strategies change the effective structure of the pipeline whereas the topology, link bandwidth, and fragmentation determine the depth and time per stage. Of course, the reason that networks are so interesting is that they are not a simple pipeline but rather an interwoven fabric of portions of many pipelines. The whole motivation for using a network rather than a bus is to allow multiple data transfers to occur simultaneously. This means that one message flow may collide with others and contend for resources. Fundamentally, the network must provide a mechanism for dealing with contention. The behavior under contention depends on several facets of the network design—the topology, the switching strategy, and the routing algorithm—but the bottom line is that at any given time a channel can only be occupied by one message. If two messages attempt to use the same channel at once, one must be deferred. Typ-

ically, each switch provides some means of arbitration for the output channels. Thus, a switch will select one of the incoming packets contending for each output and the others will be deferred in some manner.

An overarching design issue for networks is how contention is handled. This issue recurs throughout the chapter, so let's first just consider what it means for latency. Clearly, contention increases the communication latency experienced by the nodes. Exactly how it increases latency depends on the mechanisms used for dealing with contention within the network, which in turn differ depending on the basic network design strategy. For example, with packet switching, contention may be experienced at each switch. Using store-and-forward routing, if multiple packets that are buffered in the switch need to use the same output channel, one will be selected and the others are blocked in buffers until they are selected. Thus, contention adds queuing delays to the basic routing delay. With circuit switching, the effect of contention arises when trying to establish a circuit; typically, a routing probe is extended toward the destination, and if it encounters a reserved channel it is retracted. The network interface retries establishing the circuit after some delay. Thus, the start-up cost in gaining access to the network increases under contention, but once the circuit is established the transmission proceeds at full speed for the entire message.

With cut-through packet switching, two packet blocking options are available. The virtual *cut-through* approach is to spool the blocked incoming packet into a buffer so that the behavior under contention degrades to that of store-and-forward routing. The *wormhole* approach buffers only a few flits in the switch and leaves the tail of the message in place along the route. The blocked portion of the message is very much like a circuit held open through a portion of the network.

Switches have limited buffering for packets, so under sustained contention the buffers in a switch may fill up. What happens to incoming packets if there is no buffer space to hold them in the switch? In traditional data communication networks, the links are long and little feedback occurs between the two ends of the channel, so the typical approach is to discard the packet. Thus, under contention the network becomes highly unreliable, and sophisticated protocols (e.g., TCP/IP slow-start) are used at the nodes to adapt the requested communication load to what the network can deliver without a high loss rate. Discarding on buffer overrun is also used with most ATM switches, even if they are employed to build closely integrated clusters. Like the wide area case, the source receives no indication that its packet was dropped, so it must rely on some kind of time-out mechanism to deduce that a problem has occurred.

In parallel computer networks, a packet headed for a full buffer is typically blocked in place, rather than discarded; this requires a handshake between the output port and input port across the link, that is, link-level flow control. Under sustained congestion, traffic "backs up" from the point in the network where contention for resources occurs toward the sources that are driving traffic into that point. Eventually, the sources experience back pressure from the network (when it refuses to accept packets), which causes the flow of data into the network to slow

down to a rate that can move through the bottleneck.² Increasing the amount of buffering within the network allows contention to persist longer without causing back pressure at the source, but it also increases the potential queuing delays within the network when contention does occur.

One of the concerns that arises in networks with message blocking is that the backup can impact traffic that is not headed for the highly contended output. Suppose that the network traffic favors one of the destinations, say, because it holds an important global variable that is widely used. This is often called a “hot spot.” If the total amount of traffic destined for that output exceeds the bandwidth of the output, this traffic will back up within the network. If this condition persists, the backlog will propagate backward through the tree of channels directed at this destination, which is called *tree saturation* (Pfister and Norton 1985). Any traffic that crosses the tree will also be delayed. In a wormhole-routed network, an interesting alternative to blocking the message is to discard it but to inform the source of the collision. For example, in the BBN Butterfly machine the source held onto the tail of the message until the head reached the destination (i.e., formed the circuit) and if a collision occurred en route, the worm was retracted all the way back to the source (Rettberg and Thomas 1986). This greatly reduces the impact of tree saturation.

We can see from this brief discussion that all aspects of network design—link bandwidth, topology, switching strategy, routing algorithm, and flow control—combine to determine the latency per message. It should also be clear that a relationship exists between latency and bandwidth. If the communication bandwidth demanded by the program is low compared to the available network bandwidth, collisions will be few, buffers will tend to be empty, and latency will stay low, especially with cut-through routing. As the bandwidth demand increases, latency will increase due to contention.

An important and often overlooked point is that parallel computer networks are effectively a *closed system* with feedback from the network to its traffic sources. The load placed on the network depends on the rate at which processing nodes request communication, which in turn depends on how fast the network delivers this communication. Program performance is affected most strongly by latency when some kind of dependence is involved: the program must wait until a read completes or until a message is received to continue; while it waits, the load placed on the network drops and the latency decreases. This situation is very different from that of file transfers across the country contending with unrelated traffic. In a parallel machine, a program largely contends with itself for communication resources. If the machine is used in a multiprogrammed fashion, parallel programs may also contend with one another, but the request rate of each will be reduced as the service rate of the network is reduced due to the contention. Since low-latency communication is critical

2. This situation is exactly like multiple lanes of traffic converging on a narrow tunnel or bridge. When the traffic flow is less than the flow rate of the tunnel, almost no delay occurs, but when the inbound flow exceeds the tunnel bandwidth, traffic backs up. When the traffic jam fills the available storage capacity of the roadway, the aggregate traffic moves forward slowly enough that the aggregate bandwidth is equal to that of the tunnel.

to parallel program performance, the emphasis in this chapter is on cut-through packet-switched networks.

10.2.2 Bandwidth

Network bandwidth is critical to parallel program performance, in part because higher bandwidth decreases occupancy, in part because higher bandwidth reduces the likelihood of contention, and in part because phases of a program may push a large volume of data around without waiting for transmission of individual data items to be completed. Since networks behave like pipelines, it is possible to deliver high bandwidth even when the latency is large.

It is useful to look at bandwidth from two points of view: the “global” aggregate bandwidth available to all the nodes through the network and the “local” individual bandwidth available to a node. If the total communication volume of a program is M bytes and the aggregate communication bandwidth of the network is B bytes per second, then clearly the communication time is at least M/B seconds. On the other hand, if all of the communication is to or from a single node, this estimate is far too optimistic; the communication time would be determined by the bandwidth through that single node.

Let us look first at the bandwidth available to a single node and see how it may be influenced by network design choices. We have seen that the effective local bandwidth is reduced from the raw link bandwidth by the density of the packet

$$b \frac{n}{n + n_E}$$

Furthermore, if the switch blocks the packet for the routing delay of Δ cycles while it makes its routing decision, then the effective local bandwidth is further derated to

$$b \left(\frac{n}{n + n_E + w\Delta} \right)$$

since $w\Delta$ is the opportunity to transmit data that is lost while the link is blocked. Thus, network design issues such as the packet format and the routing algorithm will influence the bandwidth seen by even a single node. If multiple nodes are communicating at once and contention arises, the perceived local bandwidth will drop further (and the latency will rise). Contention at the endpoints happens in any network if multiple nodes send messages to the same node, but it may occur within the interior of the network as well. The choice of network topology and routing algorithm affects the likelihood of contention within the network.

If many of the nodes are communicating at once, it is useful to focus on the global bandwidth that the network can support rather than only the bandwidth available to each individual node. First, we should sharpen the concept of the aggregate communication bandwidth of a network. The most common notion of aggregate bandwidth is the *bisection bandwidth* of the network, which is the sum of the bandwidths of the minimum set of channels that, if removed, partition the network into two equal unconnected sets of nodes. This is a valuable concept because, if the

communication pattern is completely uniform, half of the messages are expected to cross the bisection in each direction. We will see in the following that the bisection bandwidth per node varies dramatically in different network topologies. However, bisection bandwidth is not entirely satisfactory as a metric of aggregate network bandwidth because communication is not necessarily distributed uniformly over the entire machine. If communication is localized rather than uniform, bisection bandwidth will give a pessimistic estimate of communication time. An alternative notion of global bandwidth that caters to localized communication patterns would be the sum of the bandwidth of the links from the nodes to the network. The concern with this notion of global bandwidth is that the internal structure of the network may not support it. Clearly, the available aggregate bandwidth of the network depends on the communication pattern; in particular, it depends on how far the packets travel, so we should look at this relationship more closely.

The total bandwidth of all the channels (or links) in the network is the number of channels, C , times the bandwidth per channel, that is, Cb bytes per second, Cw bits per cycle, or C phits per cycle. If each of N hosts issues a packet every M cycles with an average routing distance of h , then each packet occupies, on average, h channels for $l = n/w$ cycles, and the total load on the network is Nhl/M phits per cycle. The average link utilization is at least

$$\rho = M \frac{C}{Nhl} \quad (10.6)$$

and this obviously must be less than one. One way of looking at this is that the number of links per node, C/N , reflects the communication bandwidth (phits per cycle per node) available, on average, to each node. This bandwidth is consumed in direct proportion to the routing distance and the message size. The number of links per node is a static property of the topology. The average routing distance is determined by the topology, the routing algorithm, the program communication pattern, and the mapping of the program onto the machine. Good communication locality may yield a small h , whereas random communication will travel the average distance and really bad patterns may traverse the full diameter. The message size is determined by the program behavior and the communication abstraction. In general, the aggregate communication requirement in Equation 10.6 says that as the machine is scaled up, the channels per node must scale with the increase in expected latency.

In practice, several factors limit the channel utilization, ρ , well below unity. The load may not be perfectly balanced over all the links. Even if it is balanced, the routing algorithm may prevent all the links from being used for the particular communication pattern employed in the program. And even if all the links are usable and the load is balanced over the duration, stochastic variations in the load and contention for low-level resources may arise. All these factors affect the network's *saturation point*, which represents the total channel bandwidth it can usefully deliver. As illustrated in Figure 10.4, if the bandwidth demand placed on the network by the processors (called the *offered bandwidth*) is moderate, the latency remains low, and the delivered bandwidth increases with the offered bandwidth. However, at some point, demanding more bandwidth only increases the contention for resources and the

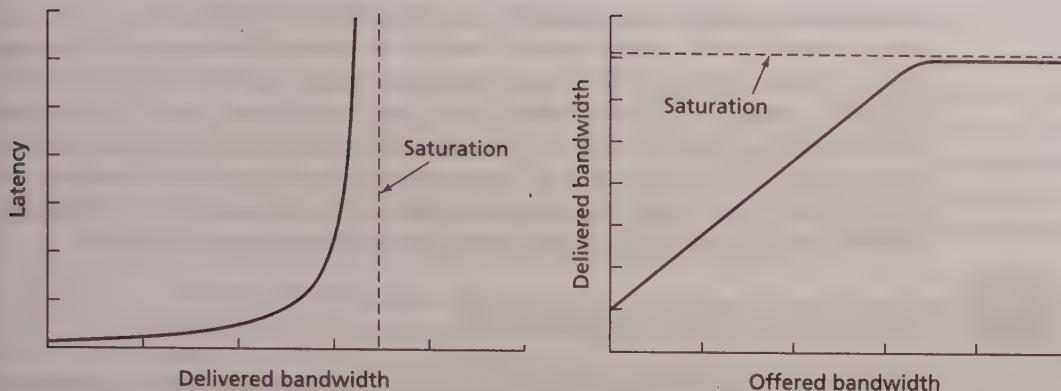


FIGURE 10.4 Typical network saturation behavior. Networks can provide low latency when the requested bandwidth is well below that which can be delivered. In this regime, the delivered bandwidth scales linearly with that requested. However, at some point, the network saturates and additional load causes the latency to increase sharply without yielding additional delivered bandwidth.

latency increases dramatically. The network is essentially moving as much traffic as it can, so additional requests just get queued up in the buffers. Increasing offered bandwidth does not increase what is delivered. We attempt to design parallel machines so that the network stays out of saturation, either by providing ample communication bandwidth or by limiting the demands placed by the processors.

A word of caution is in order regarding the dramatic increase in latency illustrated in Figure 10.4 as the network load approaches saturation. The behavior illustrated in the figure is typical of all queuing systems (and networks) under the assumption that the load placed on the system is independent of the response time. The sources keep pushing messages into the system faster than it can service them, so a queue of arbitrary length builds up somewhere and the latency grows with the length of this queue. In other words, this simple analysis assumes an *open system*, whereas in reality, parallel machines are closed systems. There is only a limited amount of buffering in the network and, usually, only a limited amount of communication buffering in the network interfaces. Thus, if these “queues” fill up, the sources will slow down, reducing their demand to the service rate since there is no place to put the next packet until one is removed. The flow control mechanisms affect this coupling between source and sink. Moreover, dependences within the parallel programs inherently embed some degree of end-to-end flow control because a processor must receive remote information before it can do additional work that depends on the information and generate additional communication traffic. Nonetheless, it is important to recognize that a shared resource, such as a network link, is not expected to be 100% utilized even in the best of circumstances.

This brief performance modeling of parallel machine networks shows that the latency and bandwidth of real networks depends on all aspects of the network design, which we will examine in some detail in the remainder of the chapter. Performance modeling of networks is itself a rich area with a voluminous literature

base, and the interested reader should consult the following references as a starting point: Agarwal (1991), Dally (1990b), Karol et al. (1987), Kermani and Kleinrock (1979), Kruskal and Snir (1983), and Peterson and Davie (1996). In addition, it is important to note that performance is not the only driving factor in network designs. Cost and fault tolerance are two other critical criteria. For example, the wiring complexity of the network is a critical issue in several large-scale machines. As these issues depend quite strongly on specifics of the design and the technology employed, we will discuss them along with examining the design alternatives.

10.3

ORGANIZATIONAL STRUCTURE

This section outlines the basic organizational structure of a parallel computer network. It is useful to think of this issue in the more familiar terms of the processor organization and the applicable engineering constraints. We normally think of the processor as being composed of datapath, control logic, and memory interface, including perhaps the on-chip portions of the memory hierarchy. The datapath is further broken down into ALU, register file, pipeline latches, and so forth. The control logic is built up from examining the data transfers that take place in the datapath. Local connections within the datapath are short and scale well with improvements in VLSI technology whereas control wires and buses are long and become slower relative to gates as chip density increases. A very similar notion of decomposition and assembly applies to the network. Scalable interconnection networks are composed of three basic components: links, switches, and network interfaces. A basic understanding of these components, their performance characteristics, and their inherent costs is essential for evaluating network design alternatives. The set of operations the components perform is quite limited, fundamentally moving packets toward their intended destination.

10.3.1

Links

A link is a cable of one or more electrical wires or optical fibers with a connector at each end that is attached to a switch or network interface port. It allows an analog signal to be transmitted from one end, received at the other, and sampled to obtain the original digital information stream. In practice, there is tremendous variation in the electrical and physical engineering of links; however, their essential logical properties can be characterized along three independent dimensions: length, width, and clocking.

1. A *short link* is one in which only a single logical value can be on the link at any time; a *long link* is viewed as a transmission line where a series of logical values propagate along the link simultaneously at a fraction of the speed of light (1–2 feet per ns, depending on the specific medium).
2. A *narrow link* is one in which data, control, and timing information are multiplexed onto each wire, such as on a single serial link; a *wide link* is one that can simultaneously transmit data and control information. In either case, net-

work links are typically narrower than to internal processor datapaths, say, 4 to 16 data bits.

3. Clocking may be synchronous or asynchronous. In the *synchronous* case, the source and destination operate on the same global clock, so data is sampled at the receiving end according to the common clock; in the *asynchronous* case, the source encodes its clock in some manner within the analog signal that is transmitted, and the destination recovers the source clock from the signal and transfers the information into its own clock domain.

A short electrical link behaves like a conventional connection between digital components. The signaling rate is essentially determined by the time to charge the wire until it represents a logical value on both ends. This time increases only logarithmically with length if enough power can be used to drive the link.³ In addition, the wire must be terminated properly to avoid reflections, which is why it is important that the link be point to point, as opposed to multipoint, like a bus.

The CRAY T3D is a good example of a wide, short, synchronous link design. Each bidirectional link contains 24 bits in each direction: 16 for data, 4 for control, and 4 providing flow control for the link in the reverse direction so that a switch will not try to deliver flits into a full buffer. The entire machine operates under a single 150-MHz clock. A flit is a single phit of 16 bits. Two of the control bits identify the phit type (00 no info, 01 routing tag, 10 packet, 11 end of packet).

In a long wire or optical fiber, the signal propagates along the link from source to destination. For a long link, the delay is clearly linear in the length of the wire. The signaling rate is determined by the time to correctly sample the signal at the receiver, so the length is limited by the signal decay along the link. If more than one wire is in the link, the signaling rate and wire length are also limited by the signal skew across the wires.

A correctly sampled analog signal can be viewed as a stream of digital symbols (phits) delivered from source to destination over time. Logical values on each wire may be conveyed by voltage levels or voltage transitions. Typically, the encoding of digital symbols is chosen so that it is easy to identify common failures (such as stuck-at faults and open connections) and easy to maintain clocking. Within the stream of symbols, individual packets must be identified. Thus, part of the signaling convention of a link is its *framing*, which identifies the start and end of each packet. In a wide link, distinct control lines may identify the head and tail phits. For example, a packet line goes high with the first header phit and stays high until the last tail phit. In the T3D, the routing tag phit and end-of-packet phit provide packet framing. In a narrow link, special control symbols are inserted in the stream to provide framing. In an asynchronous serial link, the clock must be extracted from the

3. The RC delay of a wire increases with the square of the length, so for a fixed amount of signal drive, the network cycle time is strongly affected by length. However, if the driver strength is increased using a driver tree, the time to drive the load of a longer wire only increases logarithmically. (If τ_{inv} is the propagation delay of a basic gate, then the effective propagation delay of a short wire of length l grows as $t_s = K\tau_{inv} \log l$.)

incoming analog signal as well; this is typically done with a unique synchronization burst in the sequence of binary values (Peterson and Davie 1996).

The CRAY T3E network provides a convenient contrast to the T3D. It uses a long, wide, asynchronous link design. The link is 14 bits wide in each direction, operating at 375 MHz. Each “bit” is conveyed by a low-voltage differential signal (LVDS) with a nominal swing of 600 mV on a pair of wires; that is, the receiver senses the difference in the two wires rather than the voltage relative to ground. The clock is sent along with the data. The maximum transmission distance is approximately 1 meter, but even at this length multiple bits will be on the wire at a time. A flit contains five phits, so the switches operate at 75 MHz on 70-bit quantities containing one 64-bit word plus control information. Flow control information is carried on data packets and idle symbols over the link in the reverse direction. The sequence of flits is framed into single-word and eight-word read and write request packets, message packets, and other special packets. The maximum data bandwidth of a link is 500 MB/s.

In general, the encoding of the packet within the frame is interpreted by the nodes attached to the link. Typically, the envelope is interpreted by the switch to do routing and error checking. The payload is delivered uninterpreted to the destination host, at which point further layers or internal envelopes are interpreted and peeled away. However, the destination node may need to inform the source whether it was able to hold the data. This requires some kind of node-to-node information that is distinct from the actual communication, for example, an acknowledgment in the reverse direction. With wide links, control lines may run in both directions to provide this information. Narrow links are almost always bidirectional so that special flow control signals can be inserted into the stream in the reverse direction as in the T3E.⁴

The Scalable Coherent Interface (SCI) defines both a long, wide copper link and a long, narrow fiber link. The links are unidirectional and nodes are always organized into rings. The copper link comprises 18 pairs of wires using differential signaling on both edges of a 250-MHz clock. It carries 16 bits of data, the clock, and a flag bit. The fiber link is serial and operates at 1.25 Gb/s. Packets are a sequence of 16-bit phits, with the header consisting of a destination node number phit and a command phit. The trailer consists of a 32-bit CRC (cyclic redundancy check) word. The flag bit provides packet framing by distinguishing idle symbols from packet phits. At least one idle phit occurs between successive packets.

Many evaluations of networks treat links as having a fixed cost. Common sense would suggest that the cost increases with the length of the link and its width. This is actually a point of considerable debate within the field because the relative quality of different networks depends on the cost model that is used in the evaluation. Much of the cost is in the connectors and the labor involved in attaching them, so the fixed cost is substantial. The connector cost increases with width whereas the wire cost increases with width and length. In many cases, the key constraint is the

4. This view of flow control as inherent to the link is quite different from the view in more traditional networking applications, where flow control is realized on top of the link-level protocol by special packets.

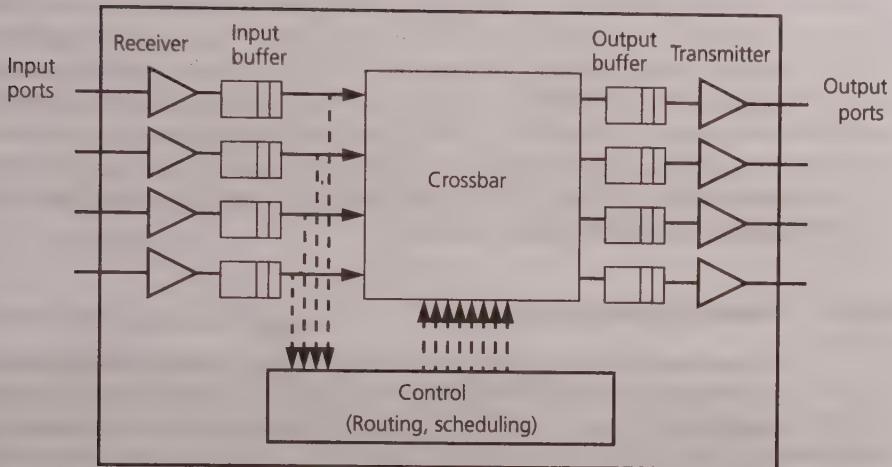


FIGURE 10.5 Basic switch organization. A set of input ports is connected to a set of output ports through a crossbar. The control logic affects the input/output connection at each point in time.

cross-sectional area of a bundle of links, say, at the bisection; this increases with width.

10.3.2 Switches

A switch consists of a set of input ports, a set of output ports, an internal “crossbar” connecting each input to every output, internal buffering, and control logic to effect the input/output connection at each point in time, as illustrated in Figure 10.5. Usually, the number of input ports is equal to the number of output ports, which is called the *degree* of the switch.⁵ Each output port includes a transmitter to drive the link. Each input port includes a matching receiver. The input port has a synchronizer in most designs to align the incoming data with the local clock domain of the switch. This is essentially a FIFO, so it is natural to provide some degree of buffering with each input port. There may also be buffering associated with the outputs or shared buffering for the switch as a whole. The complexity of the control logic depends on the routing and scheduling algorithm, as we will discuss. At the very least, it must be possible to determine the output port required by each incoming packet and to arbitrate among input ports that need to connect to the same output port.

5. As with most rules, there are exceptions. For example, in the BBN Monarch design, two distinct kinds of switches were used that had an unequal number of input and output ports (Rettberg et al. 1990). Switches that routed packets to output ports based on routing information in the header could have more outputs than inputs. An alternative device, called a *concentrator*, routed packets to any output port, which were fewer than the number of input ports and all went to the same node.

Many evaluations of networks treat the switch degree as its cost. This is clearly a major factor, but again there is room for debate. The cost of some parts of the switch is linear in the degree, for example, the transmitters, receivers, and port buffers. However, the internal interconnect cost may increase with the square of the degree. The amount of internal buffering and the complexity of the routing logic also increase more than linearly with the degree. With recent VLSI switches, the dominant constraint tends to be the number of pins, which is proportional to the number of ports times the width of each port.

10.3.3 Network Interfaces

The network interface (NI) contains one or more input/output ports to source packets to and sink packets from the network under the direction of the communication assist, which connects it to the processing node as we have seen in previous chapters. The network interface, or host nodes, behave quite differently than switch nodes and may be connected via special links. The NI formats the packets and constructs the routing and control information. It may have substantial input and output buffering compared to a switch. It may perform end-to-end error checking and flow control. Clearly, its cost is influenced by its storage capacity, processing complexity, and number of ports.

10.4

INTERCONNECTION TOPOLOGIES

Now that we understand the basic factors determining the performance and the cost of networks, we can examine each of the major dimensions of the design space in relation to these factors. This section covers the set of important interconnection topologies. Each topology is really a class of networks scaling with the number of host nodes N , so we want to understand the key characteristics of each class as a function of N . In practice, the topological properties, such as distance, are not entirely independent of the physical properties, such as length and width, because some topologies fundamentally require longer wires when packed into a physical volume, so it is important to understand both aspects.

10.4.1

Fully Connected Network

A fully connected network is essentially a single switch, which connects all inputs to all outputs. The diameter is 1 link. The degree is N . The loss of the switch wipes out the whole network; however, the loss of a link removes only one node. One such network is simply a bus, and this provides a useful reference point to describe the basic characteristics. It has the nice property that the cost scales as $O(N)$. Unfortunately, only one data transmission can occur on a bus at once, so the total bandwidth is $O(1)$, as is the bisection. In fact, the bandwidth scaling is worse than $O(1)$ because the clock rate of a bus decreases with the number of ports due to RC delays. (An Ethernet is really a bit-serial, distributed bus; it just operates at a low

enough frequency that a large number of physical connections are possible.) Another fully connected network is a crossbar. It provides $O(N)$ bandwidth, but the cost of the interconnect is proportional to the number of cross-points, or $O(N^2)$. In either case, a fully connected network is not scalable in practice. This is not to say they are not important. Individual switches are often fully connected internally and provide the basic building block for larger networks. A key metric of technological advance in networks is the degree of a cost-effective switch. With increasing VLSI chip density, the number of nodes that can be fully connected by a cost-effective switch is increasing.

10.4.2 Linear Arrays and Rings

The simplest network is a linear array of nodes numbered consecutively $0, \dots, N - 1$ and connected by bidirectional links. The diameter is $N - 1$, the average distance is roughly $2/3 N$, and removal of a single link partitions the network, so the bisection width is 1 link. Routing in such a network is trivial since there is exactly one route between any pair of nodes. To describe the route from node A to node B , let us define $R = B - A$ to be the *relative address* of B from A . This signed $\log N$ bit number is the number of links to cross to get from A to B with the positive direction being away from node 0. Since there is a unique route between a pair of nodes, clearly the network provides no fault tolerance. The network consists of $N - 1$ links and can easily be laid out in $O(N)$ space using only short wires. Any contiguous segment of nodes provides a subnetwork of the same topology as the full network.

A *ring* or *torus* of N nodes can be formed by simply connecting the two ends of an array. With unidirectional links, the diameter is $N - 1$, the average distance is $N/2$, the bisection width is 1 link, and there is one route between any pair of nodes. The relative address of B from A is $(B - A) \bmod N$. With bidirectional links, the diameter is $N/2$, the average distance is $N/3$, the degree of the node is 2, and the bisection is 2. There are two routes (two relative addresses) between pairs of nodes, so the network can function with degraded performance in the presence of a single faulty link. The network is easily laid out with $O(N)$ space using only short wires, as indicated by Figure 10.6, by simply folding the ring. The network can be partitioned into smaller subnetworks; however, the subnetworks are linear arrays rather than rings.

Although these one-dimensional networks are not scalable in any practical sense, they are an important building block conceptually and in practice. The simple routing and low hardware complexity of rings has made them very popular for local area interconnects, including FDDI, FiberChannel Arbitrated Loop, and Scalable Coherent Interface (SCI). Since they can be laid out with very short wires, it is possible to make the links very wide. For example, the KSR1 used a 32-node ring that was 128 bits wide as a building block. SCI obtains its bandwidth by using 16-bit links.

10.4.3 Multidimensional Meshes and Tori

Rings and arrays generalize naturally to higher dimensions, including 2D grids and 3D cubes, with or without end-around connections. A d -dimensional array consists

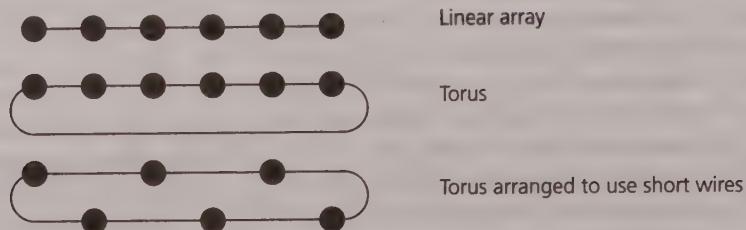


FIGURE 10.6 Linear and ring topologies. The linear array and torus are easily laid out to use uniformly short wires. The distance and cost grow as $O(N)$ whereas the aggregate bandwidth is only $O(1)$.

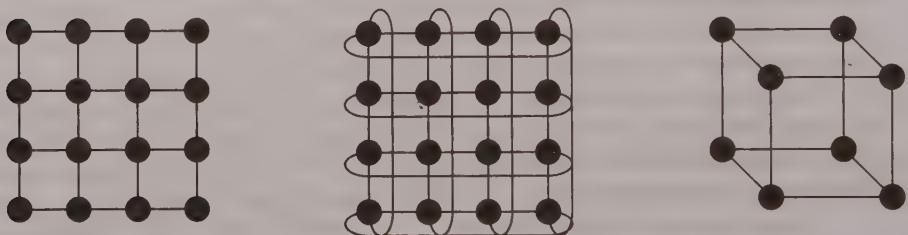


FIGURE 10.7 Grid, torus, and cube topologies. Grids, tori, and cubes are special cases of k -ary d -cube networks, which are constructed with k nodes in each of d dimensions. Low-dimensional networks pack well in physical space with short wires.

of $N = k_{d-1} \times \dots \times k_0$ nodes, each identified by its d -vector of coordinates (i_{d-1}, \dots, i_0) , where $0 \leq i_j \leq k_j - 1$ for $0 \leq j \leq d-1$. Figure 10.7 shows the common cases of two and three dimensions. For simplicity, we will assume the length along each dimension is the same, so $N = k^d$ ($k = \sqrt[d]{N}$, $r = \log_d N$). This is called a d -dimensional k -ary mesh. (In practice, engineering constraints often result in nonuniform dimensions, but the theory is easily extended to handle that case.) Each node is a switch addressed by a d -vector of radix k coordinates and is connected to the nodes that differ by one in precisely one coordinate. The node degree varies between d and $2d$, inclusive, with nodes in the middle having full degree and the corners having minimal degree.

For a d -dimensional k -ary torus, the edges wrap around from each face, so every node has degree $2d$ (in the bidirectional case) and is connected to nodes differing by one $(\bmod k)$ in each dimension. We will refer to arrays and tori collectively as meshes. The d -dimensional k -ary unidirectional torus is a very important class of networks, often called a k -ary d -cube, employed widely in modern parallel machines. These networks are usually configured as direct networks, so an additional switch degree is required to support the bidirectional host connection from each switch. They are generally viewed as having low degree, 2 or 3, so the network scales by increasing k along some or all of the dimensions.

To define the routes from node A to node B in a d -dimensional array, let $R = (b_{d-1} - a_{d-1}, \dots, b_0 - a_0)$ be the relative address of B from A . A route must cross $r_i = b_i - a_i$ links in each dimension i , where the sign specifies the appropriate direction. The simplest approach is to traverse the dimensions in order, so for $i = 0 \dots d-1$, travel r_i hops in the i th dimension. This corresponds to traveling between two locations in a metropolitan grid by driving in, say, the east-west direction, turning once, and driving in the north-south direction. Of course, we can reach the same destination by first traveling north-south and then east-west or by zigzagging anywhere between these routes. In general, we may view the source and destination points as corners of a subarray and follow any path between the corners that reduces the relative address from the destination at each hop.

The diameter of the network is $d(k-1)$. The average distance is simply the average distance in each dimension, roughly

$$d\frac{2}{3}k$$

If k is even, the bisection of a d -dimensional k -ary array is k^{d-1} bidirectional links. This is obtained by simply cutting the array in the middle by a (hyper)plane perpendicular to one of the dimensions. (If k is odd, the bisection may be a little bit larger.) For a unidirectional torus, the relative address and routing generalizes along each dimension just as for a ring. All nodes have degree d (plus the host degree), and k^{d-1} links cross the middle in each direction.

It is clear that a two-dimensional mesh can be laid out with $O(N)$ space in a plane with short wires and three-dimensional mesh in $O(N)$ volume in free space. In practice, engineering factors come into play. It is not really practical to build a huge 2D structure, and mechanical issues arise in how 3D is utilized, as illustrated by Example 10.1.

EXAMPLE 10.1 Using a direct 2D mesh topology, such as in the Intel Paragon, where a single cabinet holds 64 processors forming a 4-wide by 16-high array of nodes (each node containing a message processor and one to four compute processors), how might you configure cabinets to construct a large machine with only short wires?

Answer Although there are many possible approaches, the one used by Intel is illustrated in Figure 1.24 of Chapter 1. The cabinets stand on the floor, and large configurations are formed by attaching these cabinets side by side, forming a $16 \times k$ array. The largest configuration was a 1,824-node machine at Sandia National Laboratory configured as a 16×114 array. The bisection bandwidth is determined by the 16 links that cross between cabinets. ■

Other machines have found alternative strategies for dealing with real-world packaging restrictions. The MIT J-machine is a 3D torus where each board comprises an 8×16 torus in the first two dimensions. Larger machines are constructed by stacking these boards next to one another, with board-to-board connections providing the links in the third dimension. The Intel ASCI Red machine with 4,536 compute nodes is constructed as 85 cabinets. It allows long wires to run between

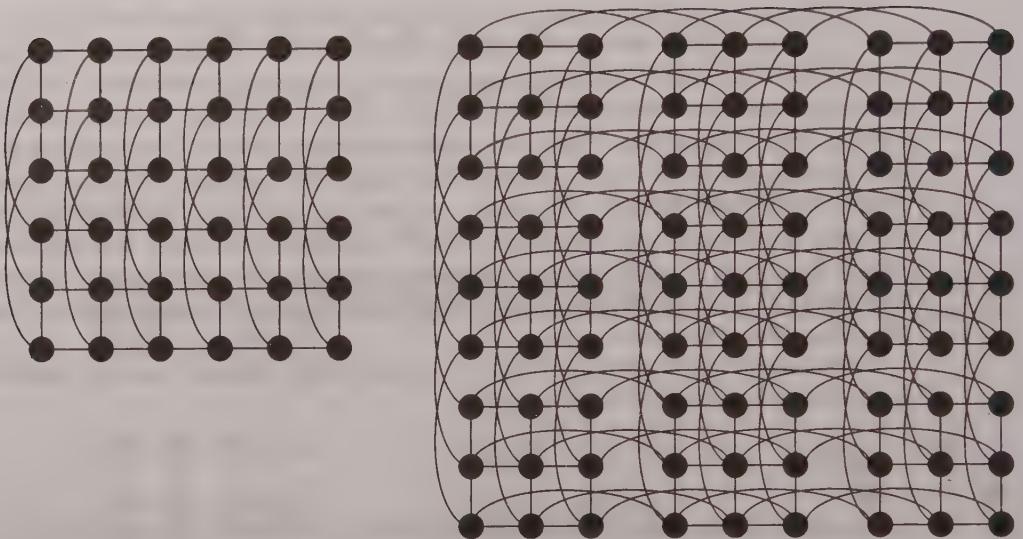


FIGURE 10.8 Embeddings of many logical dimensions in two physical dimensions. A higher-dimensional k -ary d -cube can be laid out in 2D by replicating a 2D slice and then connecting slices across the remaining dimensions. The wiring complexity of these higher-dimensional networks can be easily seen in the figure.

cabinets in each dimension. In general, with higher-dimensional meshes, several logical dimensions are embedded in each physical dimension using longer wires. Figure 10.8 shows a $6 \times 3 \times 2$ array and a four-dimensional 3-ary array embedded in a plane. It is clear that, for a given physical dimension, the average wire length and the number of wires increases with the number of logical dimensions.

10.4.4 Trees

In meshes, the diameter and average distance increases with the d th root of N . Many other topologies exist where the routing distance grows only logarithmically. The simplest of these is a tree. A binary tree has degree 3. Typically, trees are employed as indirect networks with hosts as the leaves, so for N leaves the diameter is $2 \log N$. (Such a topology could be used as a direct network of $N = k \log k$ nodes.) In the indirect case, we may treat the binary address of each node as a $d = \log N$ bit vector specifying a path from the root of the tree—the high-order bit indicates whether the node is below the left or right child of the root and so on down the levels of the tree. The levels of the tree correspond directly to the “dimension” of the network. One way to route from node A to node B would be to go all the way up to the root and then follow the path down specified by the address of B . Of course, we really only need to go up to the first common parent of the two nodes before heading down. Let $R = B \oplus A$, the bitwise xor of the node addresses, be the relative address of A and B .

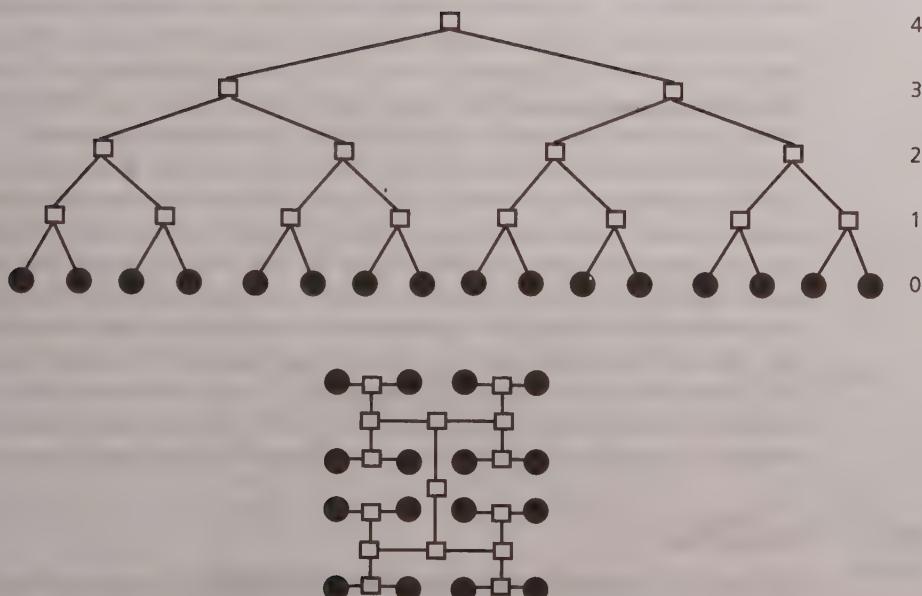


FIGURE 10.9 Binary trees. Trees are a simple network with logarithmic depth that can be laid out efficiently in 2D by using an H-tree configuration. Routing is simple, and they contain trees as subnetworks. However, the bisection bandwidth is only $O(1)$.

and i be the position of the most significant 1 in R . The route from node A to node B is simply $i + 1$ hops up followed by $i + 1$ hops down, with the direction at each branch specified by the low-order $i + 1$ bits of B .

Formally, a complete indirect binary tree is a network of $2N - 1$ nodes organized as $d + 1 = \log_2 N + 1$ levels. Host nodes occupy level 0 and are identified by a d -bit address $A = a_{d-1}, \dots, a_0$. A switch node is identified by its level i and its $d - i$ bit address $A^{(i)} = a_{d-1}, \dots, a_i$. A switch node $[i, A^{(i)}]$ is connected to a parent $[i+1, A^{(i+1)}]$ and two children $[i-1, A^{(i)}||0]$ and $[i-1, A^{(i)}||1]$ where the vertical bars indicate bitwise concatenation. There is a unique route between any pair of nodes by going up to the least common ancestor, so no fault tolerance is present. The average distance is almost as large as the diameter and the tree partitions into subtrees. One virtue of the tree is the ease of supporting broadcast or multicast operations from one node to many.

Clearly, by increasing the branching factor of the tree the routing distance is reduced. In a k -ary tree, each node has k children, the height of the tree is $d = \log_k N$, and the address of a host is specified by a d -vector of radix k coordinates describing the path down from the root.

One potential problem with trees is that they seem to require long wires. After all, when we draw a tree in a plane, the lines near the root usually grow exponentially in length with the number of levels, as illustrated in the top portion of Figure 10.9, resulting in an $O(N \log N)$ layout with $O(N)$ long wires. This is really a matter of

how you look at it. The same 16-node tree is laid out compactly in two dimensions using a recursive “H-tree” pattern, which allows an $O(N)$ layout with only $O\sqrt{N}$ long wires (Bhatt and Leiserson 1982). We can imagine using the H-tree pattern with multiple nodes on a chip, among nodes (or subtrees) on a board, or between cabinets on a floor, but the linear layout might be used between boards.

The more serious problem with the tree is its bisection. Removing a single link near the root bisects the network. It has been observed that computer scientists have a funny notion of trees; real trees get thicker toward the trunk. An even better analogy is the human circulatory system where the heart forms the root and the cells the leaves. Blood cells are routed up the veins to the root and down the arteries to the cells. Bandwidth is essentially constant across each level so that blood flows evenly. This idea has been addressed in an interesting variant of tree networks, called *fat-trees*, where the upward link to the parent has twice the bandwidth of the child links. Of course, packets don’t behave quite like blood cells, so some issues need to be sorted out on exactly how to wire this up. These will fall out easily from butterflies.

10.4.5 Butterflies

The constriction at the root of the tree can be avoided if there are “lots of roots.” This is provided by an important logarithmic network called a butterfly. (The butterfly topology arises in many settings in the literature. It is the inherent communication pattern on an element-by-element level of the FFT, the Batcher odd-even merge sort, and other important parallel algorithms. It is isomorphic to topologies in the networking literature, including the Omega and SW-Banyan networks, and is closely related to the shuffle-exchange network and the hypercube, which we will discuss later.) Given 2×2 switches, the basic building block of the butterfly is obtained by simply crossing one of each pair of edges, as illustrated in the top of Figure 10.10. This is a tool for correcting one bit of the relative address—going straight leaves the bit the same, crossing flips the bit. These 2×2 butterflies are composed into a network of $N = 2^d$ nodes in $\log_2 N$ levels of switches by systematically changing the cross edges as shown by the 16-node butterfly illustrated in the bottom portion of Figure 10.10. This configuration shows an indirect network with unidirectional links going upward so that hosts deliver packets into level 0 and receive packets from level d . Each level corrects one additional bit of the relative address. Each node at level d forms the root of a tree with all the hosts as leaves, and from each host is a tree of routes reaching every node at level d .

A d -dimensional indirect butterfly has $N = 2^d$ host nodes and $d2^{d-1}$ switch nodes of degree 2 organized as d levels of $N/2$ nodes each. A switch node at level i , $[i, A]$ has its outputs connected to nodes $[i + 1, A]$ and $[i + 1, A \oplus 2^i]$. To route from A to B , compute the relative address $R = A \oplus B$ and at level i use the “straight edge” if r_i is 0 and the cross edge otherwise. The diameter is $\log N$. In fact, all routes are $\log N$ long. The bisection is $N/2$. (A slightly different formulation with only one host connected to each edge switch has bisection N but twice as many switches at each level and one additional level.)

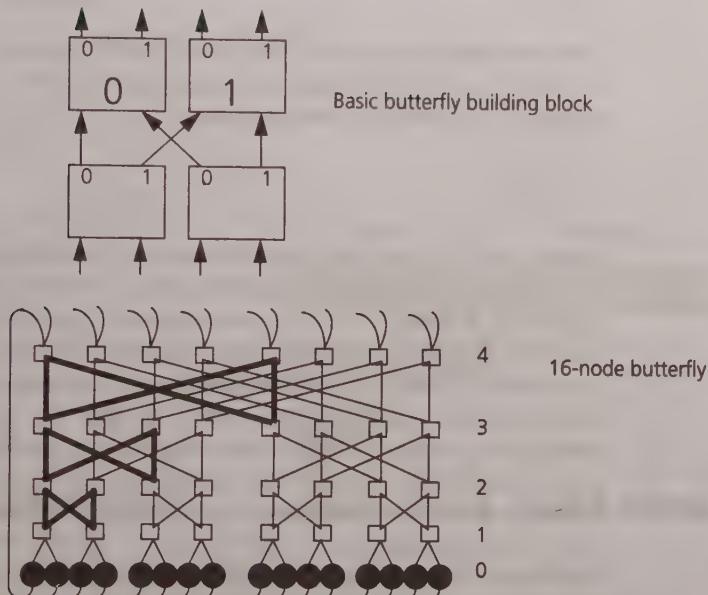


FIGURE 10.10 Butterfly. The butterfly is a logarithmic depth network constructed by composing 2×2 blocks that correct one bit in the relative address. It can be viewed as a tree with multiple roots.

A d -dimensional k -ary butterfly is obtained using switches of degree k , a power of two. The address of a node is then viewed as a d -vector of radix k coordinates, so each level corrects $\log k$ bits in the relative address. In this case, there are $\log_k N$ levels. In effect, this fuses adjacent levels into a higher radix butterfly.

There is exactly one route from each host output to each host input, so no inherent fault tolerance is present in the basic topology. However, unlike the 1D mesh or the tree where a broken link partitions the network, there is the potential for fault tolerance in the butterfly. For example, the route from A to B may be broken, but there is a path from A to another node C and from C to B . There are many proposals for making the butterfly fault tolerant by just adding a few extra links. One simple approach is to add an extra level to the butterfly so there are two routes to every destination from every source. This approach was used in the BBN T2000.

The butterfly appears to be a qualitatively more scalable network than meshes and trees because each packet crosses $\log N$ links and there are $N \log N$ links in the network, so on average it should be possible for all the nodes to send messages anywhere all at once. By contrast, a 2D torus or tree has only two links per node, so nodes can only send messages a long distance infrequently and very few nodes are close. A similar argument can be made in terms of bisection. For a random permutation of data among the N nodes, $N/2$ messages are expected to cross the bisection in each direction. The butterfly has $N/2$ links across the bisection whereas the d -dimensional mesh has only

$$\frac{d-1}{N^{\frac{d}{d}}}$$

links and the tree only one. Thus, as the machine scales, a given node in a butterfly can send every other message to a node on the other side of the machine whereas a node in a 2D mesh can send only every

$$\sqrt[2]{\frac{N}{2}} \text{th}$$

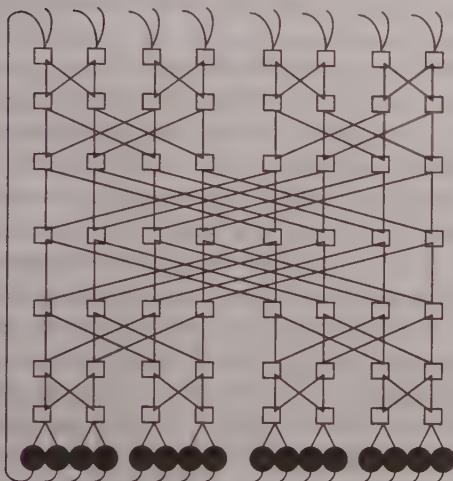
message and the tree only every N th message to the other side.

This analysis has two potential problems, however. The first is cost. In a tree or d -dimensional mesh, the cost of the network is a fixed fraction of the cost of the machine. For each host node there is one switch and d links. In the butterfly, the cost of the network per node increases with the number of nodes since for each host node there are $\log N$ switches. Thus, neither scales perfectly. The real question comes down to what a switch costs relative to a processor and what fraction of the overall cost of the machine we are willing to invest in the network to be able to deliver a certain communication performance. If the switch and link is 10% of the cost of the node, then on a 1,024-processor machine the network will be only one-half the total cost with a butterfly. On the other hand, if the switch is equal in cost to a node, we are unlikely to consider more than a low-dimensional network. Conversely, if we reduce the dimension of the network, we may be able to invest more in each switch.

The second problem is that even though the butterfly has enough links to support the bandwidth \times distance product of a random permutation, the topology of the butterfly will not allow an arbitrary permutation of N messages among the N nodes to be routed without conflict. A path from an input to an output blocks the paths of many other input/output pairs because there are shared edges. In fact, even when allowed to go through the butterfly twice, permutations exist that cannot be routed without conflicts. However, if two butterflies are laid back to back so that a message goes forward through one and in the reverse direction through the other, then for any permutation there exists a choice of intermediate positions that allows a conflict-free routing of the permutation. This back-to-back butterfly is called a Benes network (Benes 1965; Leighton 1992), and it has been extensively studied because of its elegant theoretical properties. It is often seen as having little practical significance because it is costly to compute the intermediate positions and the permutation has to be known in advance. On the other hand, there is another interesting theoretical result that says that on a butterfly any permutation can be routed with very few conflicts (with high probability) by first sending every message to a random intermediate node and then routing the messages to the desired destination (Leighton 1992). These two results come together in a very nice practical way in the fat-tree network, as follows.

A d -dimensional k -ary fat-tree is formed by taking a d -dimensional k -ary Benes network and folding it back on itself at the high-order dimension, as illustrated in Figure 10.11. The collection of $N/2$ switches at level i is viewed as N^{d-i} “fat nodes”

16-node Benes network (unidirectional)



16-node 2-ary fat-tree (bidirectional)

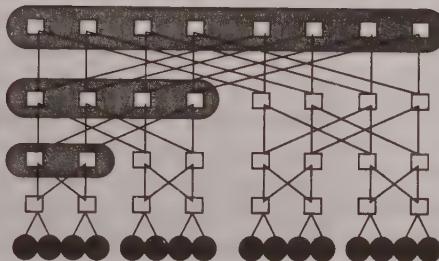


FIGURE 10.11 Benes network and fat-tree. A Benes network is constructed essentially by connecting two butterflies back to back. It has the interesting property that it can route any permutation in a conflict-free fashion, given the opportunity to compute the route off-line. Two forward-going butterflies do not have this property. The fat-tree is obtained by folding the second half of the Benes network back on itself and fusing the two directions so that it is possible to turn around at each level. Collections of switches serve as fat nodes.

of 2^{i-1} switches. The edges of the forward-going butterfly go up the tree toward the roots and the edges of the reverse butterfly go down toward the leaves. To route from A to B , pick a random node C in the least common ancestor fat node of A and B and take the unique tree route from A to C and the unique tree route back down from C to B . Let i be the highest dimension of difference in A and B ; then there are 2^i root nodes to choose from, so the longer the routing distance the more the traffic can be distributed. This topology clearly has a great deal of fault tolerance—it has the bisection of the butterfly, the partitioning properties of the tree, and allows essentially all permutations to be routed with very little contention. It is used in the Connection Machine CM-5 and the Meiko CS-2. In the CM-5, the randomization on the upward path is done dynamically by the switches; in the CS-2, the source node chooses the ancestor. A particularly important practical property of butterflies and fat-trees is that the nodes have a fixed degree independent of the size of the network. This allows networks of any size to be constructed with the same switches. As is indicated in Figure 10.11, the physical wiring complexity of the higher levels of a fat-tree or any butterfly-like network becomes critical, since a large number of long wires connect to different places.

10.4.6 Hypercubes

It may seem that the straight edges in the butterfly are a target for potential optimization since they take a packet forward to the next level in the same column. Consider what happens if we collapse all the switches in a column into a single $\log N$ degree switch. This has brought us full circle; it is a d -dimensional 2-ary torus! Actually, we need to split the switches in a column in half and associate them with the two adjacent nodes. It is called a *hypercube* or *binary n -cube*. Each of the $N = 2^d$ nodes is connected to the d nodes that differ by exactly one bit in address. The relative address $R(A, B) = A \oplus B$ specifies the dimensions that must be crossed to go from A to B . Clearly, the length of the route is equal to the number of ones in the relative address. The dimensions can be corrected in any order (corresponding to the different ways of getting between opposite corners of the subcube), and the butterfly routing corresponds exactly to dimension order routing, called *e-cube routing* in the hypercube literature. Fat-tree routing corresponds to picking a random node in the subcube defined by the high-order bit in the relative address, sending the packet “up” to the random node and back “down” to the destination. Observe that the fat-tree uses distinct sets of links for the two directions, so to get the same properties we need a pair of bidirectional links between nodes in the hypercube.

The hypercube is an important topology that has received tremendous attention in the theoretical literature. For example, lower-dimensional meshes can be embedded one to one in the hypercube by choosing an appropriate labeling of the nodes. Recall from digital design that a graycode sequence orders the numbers from 0 to $2^d - 1$ so that adjacent numbers differ by 1 bit. This shows how to embed a 1D mesh into a d -cube, and it can be extended to any number of dimensions (see Exercise 10.7). Clearly, butterflies, shuffle-exchange networks, and the like embed easily. (Interestingly, a d -cube does not quite embed a $d - 1$ level tree because one extra node is in the d -cube.)

Practically speaking, the hypercube was used by many of the early large-scale parallel machines, including the Cal Tech research prototypes (Seitz 1985), the first three Intel iPSC generations (Ratner 1985), and three generations of nCUBE machines. Later large-scale machines, including the Intel Delta, the Intel Paragon, and the CRAY T3D, use low-dimensional meshes. One of the reasons for the shift is that in practice the hypercube topology forces the designer to use switches of a degree that supports the largest possible configuration. Ports are wasted in smaller configurations. The k -ary d -cube approach provides the practical scalability of allowing arbitrarily sized configurations to be constructed with a given set of components, that is, with switches of fixed degree. Nonetheless, this begs the question, what should the degree be?

The general trend in network design in parallel machines is toward switches that can be wired in an arbitrary topology. We see this, for example, in the IBM SP-2, SGI Origin, Myricom network, and most ATM switches. The designer may choose to adopt a particular regular topology or may wire together configurations of different sizes differently. At any point in time, technology factors such as pin-out and chip area limit the largest potential degree.

10.5

EVALUATING DESIGN TRADE-OFFS IN NETWORK TOPOLOGY

The k -ary d -cube provides a convenient framework for evaluating design alternatives for direct networks. The design question can be posed in two ways. Given a choice of dimension, the design of the switch is determined and we can ask how the machine scales. Alternatively, for the machine scale of interest, that is, $N = k^d$, we may ask what the best dimensionality is under salient cost constraints. We have the 2D torus at one extreme, the hypercube at the other, and a spectrum of networks between. As with most aspects of architecture, the key to evaluating trade-offs is to define the cost model and performance model and then to optimize the design accordingly. Network topology has been a point of lively debate over the history of parallel architectures. To a large extent, this is because different positions make sense under different cost models and the technology keeps changing. Once the dimensionality (or degree) of the switch is determined, the space of candidate networks is relatively constrained, so the question is how large a degree is worth working toward.

Let's collect what we know about this class of networks in one place. The total number of switches is N , regardless of degree; however, the switch degree is d , so the total number of links is $C = Nd$ and there are $2wd$ pins per node. The average routing distance is

$$d\left(\frac{k-1}{2}\right),$$

the diameter is $d(k - 1)$, and $k^{d-1} = N/k$ links cross the bisection in each direction (for even k). Thus, there are $2Nw/k$ wires crossing the middle of the network.

If our primary concern is the routing distance, then we are inclined to maximize the dimension and build a hypercube. This would be the case with store-and-forward routing, assuming that the degree of the switch and the number of links were not a significant cost factor. In addition, we get to enjoy its elegant mathematical properties. Accordingly, this was the topology of choice for most of the first-generation large-scale parallel machines. However, with cut-through routing and a more realistic hardware cost model, the choice is much less clear. If the number of links or the switch degree is the dominant cost, we are inclined to minimize the dimension and build a mesh. For the evaluation to make sense, we want to compare the performance of design alternatives with roughly equal cost. Different assumptions about what aspects of the system are costly lead to very different conclusions.

The assumed communication pattern influences the decision too. If we look at the worst-case traffic pattern for each network, we will prefer high-dimensional networks where essentially all the paths are short. If we look at patterns where each node is communicating with only one or two near neighbors, we will prefer low-dimensional networks since only a few of the dimensions are actually used.

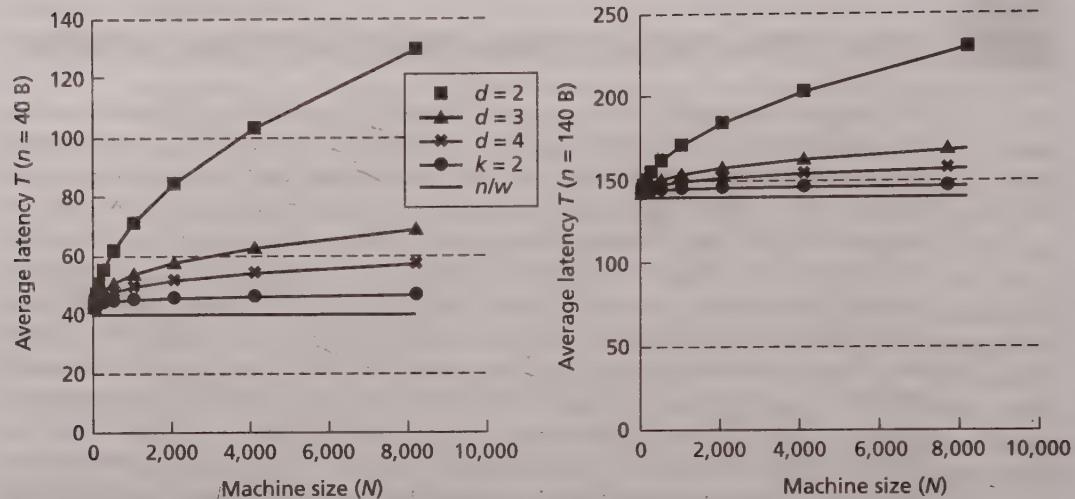


FIGURE 10.12 Unloaded latency scaling of networks for various dimensions with fixed link width. The n/w line shows the channel occupancy component of message transmission, that is, the time for the bits to cross a single channel, which is independent of network topology. The curves show the additional latency due to routing.

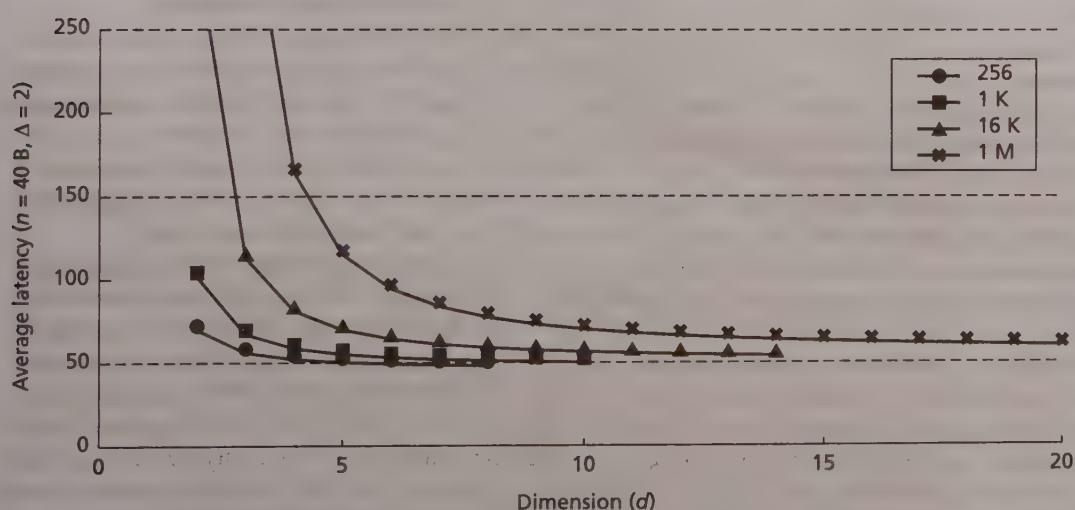
10.5.1 Unloaded Latency

Figure 10.12 shows the increase in average unloaded latency under our model of cut-through routing for 2-, 3-, and 4-cubes, as well as binary d -cubes ($k = 2$), as the machine size is scaled up. It assumes unit routing delay per stage ($\Delta = 1$) and shows message sizes of 40 and 140 bytes, with $w = 1$ byte. The bottom line shows the portion of the latency resulting from channel occupancy. As we should expect, for smaller messages (or larger routing delay per stage) the scaling of the low-dimension networks is worse because a message experiences more routing steps, on average. However, in making comparisons across the curves in this figure, we are tacitly assuming that the difference in degree is not a significant component of the system cost. In addition, 1 cycle routing is very aggressive; more typical values for high-performance switches are 4–8 network cycles (see Table 10.1). On the other hand, larger message sizes are also common.

To focus our attention on the dimensionality of the network as a design issue, we can fix the cost model and the number of nodes that reflects our design point and examine the performance characteristics of networks with fixed costs for a range of d . Figure 10.13 shows the unloaded latency for short messages as a function of the dimensionality for four machine sizes. For large machines, the routing delays in low-dimensionality networks dominate. For higher dimensionality, the latency approaches the channel time. This “equal number of nodes” cost model has been widely used to support the view that low-dimensional networks do not scale well.

Table 10.1 Link Width and Routing Delay for Various Parallel Machine Networks

Machine	Topology	Cycle Time (ns)	Channel Width (bits)	Routing Delay (cycles)	Flit (data bits)
nCUBE/2	Hypercube	25	1	40	32
TMC CM-5	Fat-Tree	25	4	10	4
IBM SP-2	Banyan	25	8	5	16
Intel Paragon	2D Mesh	11.5	16	2	16
Meiko CS-2	Fat-Tree	20	8	7	8
CRAY T3D	3D Torus	6.67	16	2	16
DASH	Torus	30	16	2	16
J-Machine	3D Mesh	31	8	2	8
Monsoon	Butterfly	20	16	2	16
SGI Origin	Hypercube	2.5	20	16	160
Myricom	Arbitrary	6.25	16	50	16

**FIGURE 10.13** Unloaded latency for k -ary d -cubes with equal node count ($n = 40$ B, $\Delta = 2$) as a function of degree. With the link width and routing delay fixed, the unloaded latency for large networks rises sharply at low dimensions due to routing distance.

It is not surprising that higher-dimensional networks are superior under the cost model of Figure 10.13 since the added switch degree, number of channels, and channel length come essentially for free. The high-dimension networks have a much larger number of wires and pins and bigger switches than the low-dimension networks. For the rightmost end of the graph in Figure 10.13, the network design is quite impractical. The cost of the network is a significant fraction of the cost of the large-scale parallel machine, so it makes sense to compare equal cost designs under appropriate technological assumptions. As chip size and density improve, the switch internals tend to become a less significant cost factor whereas pins and wires remain critical. In the extreme, the physical volume of the wires presents a fundamental limit on the amount of interconnection that is possible. So let's compare these networks under assumptions of equal wiring complexity.

One sensible comparison is to keep the total number of wires per node constant, that is, fix the number of pins, $2dw$. Let's take as our baseline a 2-cube with channel width $w = 32$, so there are a total of 128 wires per node. With more dimensions, there are more channels and they must each be thinner. In particular, $w_d = \lfloor 64/d \rfloor$. So in an 8-cube, the links are only 8 bits wide. Assuming 40- and 140-byte messages, a uniform routing delay of 2 cycles per hop, and uniform cycle time, the unloaded latency under equal pin scaling is shown in Figure 10.14. This figure shows a very different story. As a result of narrower channels, the channel time becomes greater with increasing dimension; this mitigates the reduction in routing delay stemming from the smaller routing distance. The very large configurations still experience large routing delays for low dimensions, regardless of the channel width, but all the configurations have an optimum unloaded latency at modest dimension.

If the design is not limited by pin count, the critical aspect of the wiring complexity is likely to be the number of wires that cross through the middle of the machine. If the machine is viewed as laid out in a plane, the physical bisection width grows only with the square root of the area, and in three-space it only grows as the two-thirds power of the volume. Even if the network has a high logical dimension, it must be embedded in a small number of physical dimensions, so the designer must contend with the cross-sectional area of the wires crossing the midplane.

We can focus on this aspect of the cost by comparing designs with an equal number of wires crossing the bisection. At one extreme, the hypercube has N such links. Let us assume these have unit size. A 2D torus has only $2\sqrt{N}$ links crossing the bisection, so each link could be $\sqrt{N}/2$ times the width of that used in the hypercube. By the equal bisection criteria, we should compare a 1,024-node hypercube with bit-serial links with a torus of the same size using 32-bit links. In general, the d -dimensional mesh with the same bisection width as the N -node hypercube has links of width $w_d = \sqrt[d]{N}/2 = k/2$. Assuming cut-through routing, the average latency of an n -byte packet to a random destination on an unloaded network is as follows.

$$\begin{aligned} T(n, N, d) &= \frac{n}{w_d} + \Delta \cdot d \left(\frac{k-1}{2} \right) \\ &= \frac{n}{k/2} + \Delta \cdot d \left(\frac{k-1}{2} \right) = \frac{n}{\sqrt[d]{N}/2} + \Delta \cdot d \left(\frac{\sqrt[d]{N}-1}{2} \right) \end{aligned} \quad (10.7)$$

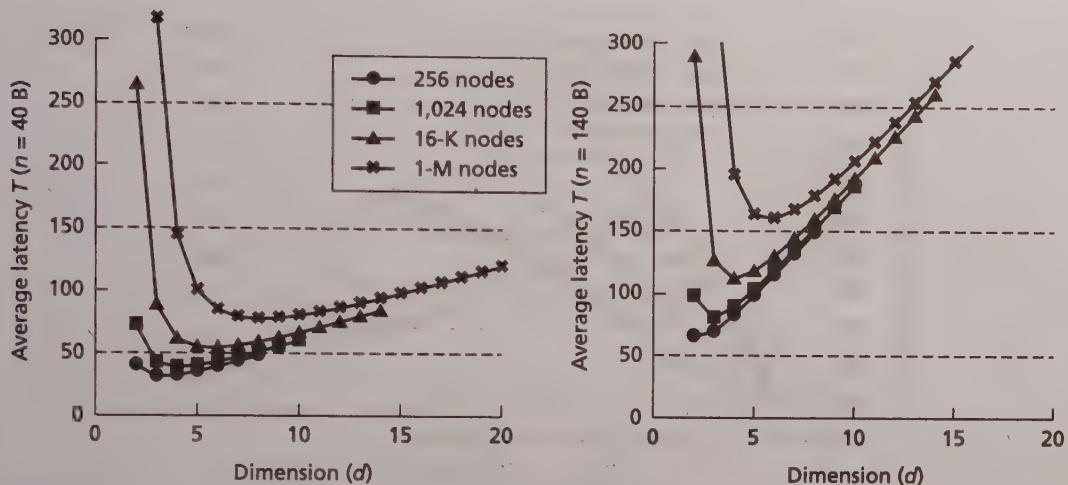


FIGURE 10.14 Unloaded latency for k -ary d -cubes with equal pin count ($n = 40$ B and $n = 140$ B, $\Delta = 2$). With equal pin count, higher dimensions imply narrower channels, so the optimal design point balances the routing delay (which increases with lower dimension) against channel time (which increases with higher dimension).

Thus, increasing the dimension tends to decrease the routing delay but increase the channel time, as with equal pin count scaling. (The reader can verify that the minimum latency is achieved when the two terms are essentially equal.)

Figure 10.15 shows the average latency for 40-byte messages, assuming $\Delta = 2$, as a function of the dimension for a range of machine sizes. As the dimension increases from $d = 2$, the routing delay drops rapidly, whereas the channel time increases steadily throughout as the links get thinner. (The $d = 2$ point is not shown for $N = 1M$ nodes because it is rather ridiculous; the links are 512 bits wide and the average number of hops is 1,023.) For machines up to a few thousand nodes, $\sqrt[3]{N}/2$ and $\log N$ are very close, so the impact of the additional channels on channel width becomes the dominant effect. If large messages are considered, the routing component becomes even less significant. For large machines, the low-dimensional meshes under this scaling rule become impractical because the links become very wide.

Thus far, we have concerned ourselves with the wiring cross-sectional area, but we have not worried about the wire length. If a d -cube is embedded in a plane, that is, if $d/2$ dimensions are embedded in each physical dimension such that the distance between the centers of the nodes is fixed, then each additional dimension increases the length of the longest wire by a \sqrt{k} factor. Thus, the length of the longest wire in a d -cube is $k^{n/2-1}$ times that in the 2-cube. Accounting for increased wire length further strengthens the argument for a modest number of dimensions. This accounting might be done in three ways. If we assume that multiple bits are pipelined on the wire, then the increased length effectively increases the routing delay. If

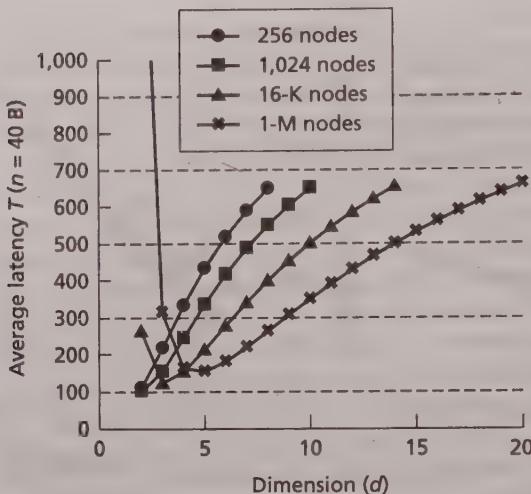


FIGURE 10.15 Unloaded latency for k -ary d -cubes with equal bisection width ($n = 40$ B, $\Delta = 2$). The balance between routing delay and channel time shifts even more in favor of low-degree networks with an equal bisection width scaling rule.

the wires are not pipelined, then the cycle time of the network is increased as a result of the time to drive the wire, which is logarithmic in the wire length.

The embedding of the d -dimensional network into few physical dimensions introduces second-order effects that may enhance the benefit of low dimensions. If a high-dimension network is embedded systematically into a plane, the wire density tends to be highest near the bisection and low near the perimeter. A 2D mesh has uniform wire density throughout, so it makes better use of the area that it occupies.

We should also look at the trade-offs in network design from a bandwidth viewpoint. The key factor influencing latency with equal wire complexity scaling is the increased channel bandwidth at low dimension. Wider channels are beneficial if most of the traffic arises from or is delivered to one or a few nodes. If traffic is localized so that each node communicates with just a few neighbors, only a few of the dimensions are utilized, and again the higher-link bandwidth dominates. If a large number of nodes are communicating throughout the machine, then we need to model the effects of contention on the observed latency and see where the network saturates.

Before leaving the examination of trade-offs for latency in the unloaded case, we should note that the evaluation is rather sensitive to the relative time to cross a wire and to cross a switch. If the routing delay per switch is 20 times that of the wire, the picture is very different, as shown in Figure 10.16. This is the reason for using a higher-dimensionality network in the SGI Origin.

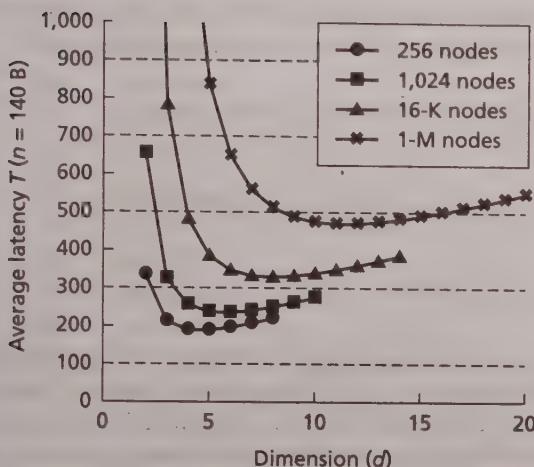


FIGURE 10.16 Unloaded latency for k -ary d -cubes with equal pin count and larger routing delays ($n = 140$ B, $\Delta = 20$). When the time to cross a switch is significantly larger than the time to cross a wire, as is common in practice, higher-degree switches become much more attractive.

10.5.2 Latency under Load

In order to analyze the behavior of a network under load, we need to capture the effects of traffic congestion on all the other traffic that is moving through the network. These effects can be subtle and far reaching. Returning to the traffic analogy, notice when you are next driving down a loaded freeway, for example, that where a pair of freeways merge and then split, the traffic congestion is even worse than a series of on-ramps merging into a single freeway. There is far more driver-to-driver interaction in the interchange, and at some level of traffic load the whole thing just seems to stop. Networks behave in a similar fashion, but there are many more interchanges. In order to evaluate these effects in a family of topologies, we must take a position on the traffic pattern, the routing algorithm, the flow control strategy, and a number of detailed aspects of the internal design of the switch. We can then either develop a queuing model for the system or build a simulator for the proposed set of designs. The trick, as in most other aspects of computer design, is to develop models that are simple enough to provide intuition at the appropriate level of design yet accurate enough to offer useful guidance as the design refinement progresses.

We will use a closed-form model of contention delays developed in Agarwal (1991) for random traffic k -ary d -cubes using dimension-order cut-through routing and unbounded internal buffers, so flow control and deadlock issues do not arise. The model predictions correlate well with simulation results for networks meeting the same assumptions. This model is based on earlier work by Kruskal and Snir (1983) modeling the performance of indirect (Banyan) networks. Without going through the derivation, the main result is that we can model the latency for random

communication of messages of size n on a k -ary d -cube with channel width w at a load corresponding to an aggregate channel utilization of ρ by

$$T(n, k, d, w, \rho) = \frac{n}{w} + h_{\text{ave}}(\Delta + W(n, k, d, w, \rho)), \text{ where}$$

$$W(n, k, d, w, \rho) = \frac{n}{w} \cdot \frac{\rho}{1 - \rho} \cdot \frac{h_{\text{ave}} - 1}{h_{\text{ave}}^2} \cdot \left(1 + \frac{1}{d}\right), \text{ and where} \quad (10.8)$$

$$h_{\text{ave}} = d \left(\frac{k - 1}{2} \right)$$

Using this model, we can compare the latency under load for networks of low or high dimension of various sizes, as we did for unloaded latency. Figure 10.17 shows the predicted latency on a 1,024-node 32-ary 2-cube and a 1,000-node 10-ary 3-cube as a function of the requested aggregate channel utilization, assuming equal channel width, for relatively small message sizes of 4, 8, 16, and 40 phits. We can see from the right end of the curves that the two networks saturate at roughly the same channel utilization; however, this saturation point decreases rapidly with the message size. The left end of the curves indicates the unloaded latency. The higher-degree switch enjoys a lower base routing delay with the same channel time since there are fewer hops and equal channel widths. As the load increases, this difference becomes less significant. Notice how large the contended latency is compared to the unloaded latency. Clearly, in order to deliver low-latency communication to the user program, it is important that the machine is designed so that the network does not go into saturation easily, either by providing excess network bandwidth or by conditioning the processor load.

The data in Figure 10.17 raises a basic trade-off in network design. How large a packet should the network be designed for? The data shows clearly that networks move small packets more efficiently than large ones. However, smaller packets have worse packet efficiency due to the routing and control information contained in each one and require more network interface events for the same amount of data transfer. For any given technology and detailed design, there is an optimal point.

We must be a bit careful about the conclusions we draw from Figure 10.17 regarding the choice of network dimension. The curves in the figure show how efficiently each network utilizes the set of channels it has available to it. The figure suggests that both use their channels with roughly equal effectiveness. However, the higher-dimensional network has a much greater available bandwidth per node; it has 1.5 times as many channels per node and each message uses fewer channels. In a k -ary d -cube, the available phits per cycle under random communication are

$$\frac{Nd}{d \frac{(k-1)}{2}}$$

or $2/(k-1)$ phits per cycle per node ($2w/k - 1$ bits per cycle). The 3-cube in our example has almost four times as much available bandwidth at the same channel utili-

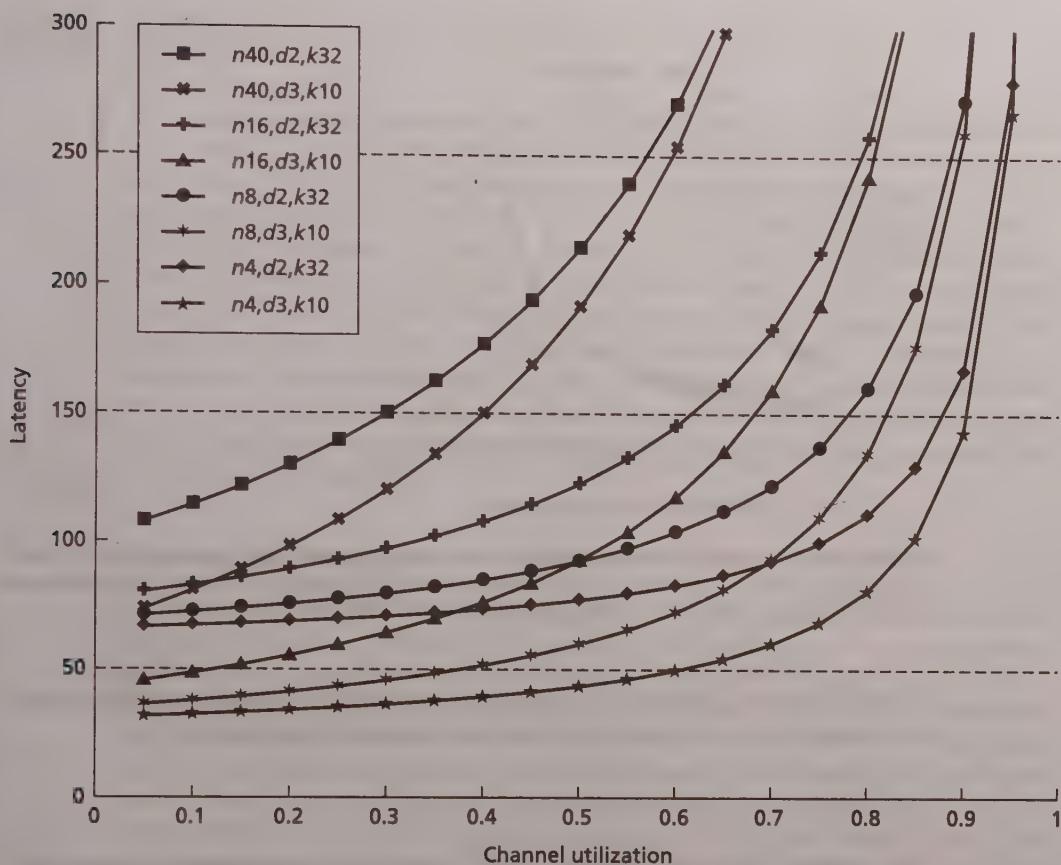


FIGURE 10.17 Latency with contention versus load for 32-ary 2-cube and 10-ary 3-cube with routing delay 2. At low channel utilization, the higher-dimensional network has significantly low latency with equal channel width, but as the utilization increases they converge toward the same saturation point.

lization, assuming equal channel width. Thus, if we look at latency against delivered bandwidth, the picture looks rather different, as shown in Figure 10.18. The 2-cube starts out with a higher base latency and saturates before the 3-cube begins to feel the load.

This comparison brings us back to the question of appropriate equal cost comparison. As an exercise, you can investigate the curves for equal pin-out and equal bisection comparison. Widening the channels shifts the base latency down by reducing channel time, increases the total available bandwidth, and reduces the waiting time at each switch since each packet is serviced faster. Thus, the results are quite sensitive to the cost model.

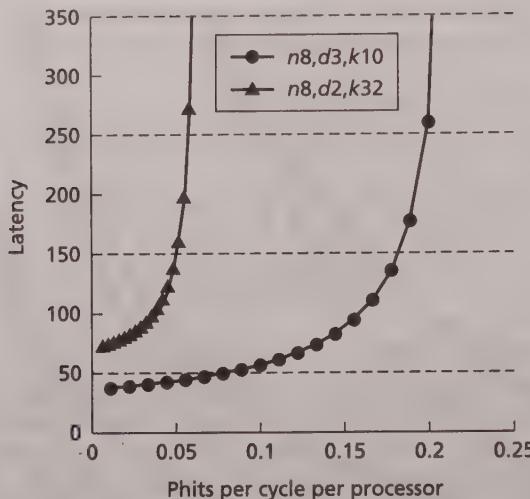


FIGURE 10.18 Latency versus phits per cycle with contention. Comparing the 32-ary 2-cube and 10-ary 3-cube with routing delay 2 at equal average traffic per link shows that the higher-degree networks handle greater load before saturating.

Some interesting observations arise when the width is scaled against dimension. For example, using the equal bisection rule, the capacity per node is

$$C(N, d) = W_d \cdot \frac{2}{k-1} \approx 1$$

The aggregate capacity for random traffic is essentially independent of dimension! Each host can expect to drive, on average, a fraction of a bit per network clock period. This observation yields a new perspective on low-dimension networks. Generally, the concern is that each of several nodes must route messages a considerable distance along one dimension. Thus, each node must send packets infrequently. Under the fixed bisection width assumption, with low dimension the channel becomes a shared resource pool for several nodes whereas high-dimension networks partition the bandwidth resource for traffic in each of several dimensions. When a node uses a channel in the low-dimension case, it uses it for a shorter amount of time. In most systems, pooling results in better utilization than partitioning.

In current machines, the area occupied by a node is much larger than the cross section of the wires, and the scale is generally limited to a few thousand nodes. In this regime, the bisection of the machine is often realized by bundles of cables. This represents a significant engineering challenge, but it is not a fundamental limit on the machine design. The tendency is to use wider links and faster signaling in topologies with short wires, as illustrated by Table 10.1, but it is not as dramatic as the equal bisection scaling rule would suggest.

10.6 ROUTING

Recall that the routing algorithm of a network determines which of the possible paths from source to destination are used as routes and how the route followed by each particular packet is determined. We have seen, for example, that in a k -ary d -cube the set of shortest routes is completely described by the relative address of the source and destination, which specifies the number of links that need to be crossed in each dimension. Dimension order routing restricts the set of legal paths so that there is exactly one route from each source to each destination—the one obtained by first traveling the correct distance in the low-order dimension, then the next dimension, and so on. This section describes the different classes of routing algorithms that are used in modern machines and the key properties of good routing algorithms, such as producing a set of deadlock-free routes, maintaining low latency, spreading load evenly, and tolerating faults.

10.6.1 Routing Mechanisms

Let's start with the nuts and bolts. Recall that the basic operation of a switch is to monitor the packets arriving at its inputs and for each input packet to select an output port on which to send it out. Thus, a routing algorithm is a function $R : N \times N \rightarrow C$, which at each switch maps the destination node n_d to the next channel on the route. High-speed switches basically use three mechanisms to determine the output channel from information in the packet header: arithmetic, source-based port select, and table lookup. In parallel computer networks, the switch needs to be able to make the routing decision for all its inputs every few cycles, so the mechanism needs to be simple and fast.

Simple arithmetic operations are sufficient to select the output port in most regular topologies. For example, in a 2D mesh, each packet can carry the signed distance to travel in each dimension $[\Delta x, \Delta y]$ in the packet header. The routing operation at switch ij is given by the following:

Direction	Condition
West ($-x$)	$\Delta x < 0$
East ($+x$)	$\Delta x > 0$
South ($-y$)	$\Delta x = 0, \Delta y < 0$
North ($+y$)	$\Delta x = 0, \Delta y > 0$
Processor	$\Delta x = 0, \Delta y = 0$

To accomplish this kind of routing, the switch needs to test the address in the header and decrement or increment one routing field. Typically, routes in a grid are determined by first moving in the Δx direction and then in the Δy . More generally, in a k -ary d -cube, the routes are determined by moving in each dimension from lowest numbered to highest, called *dimension order* routing. For a binary cube, the switch computes the position of the first bit that differs between the destination and the local node address (or the first nonzero bit if the packet carries the relative address

of the destination) and traverses the link in this dimension, called *e-cube* routing. This kind of mechanism is used in Intel and nCUBE hypercubes, the Paragon, the Cal Tech Torus Routing Chip (Seitz and Su 1993), and the J-machine, among others.

A more general approach is *source-based routing*, in which the source builds a header consisting of the output port number for each switch along the route, p_0, p_1, \dots, p_{h-1} . Each switch simply strips off the port number from the front of the message and sends the message out on the specified channel. This allows a very simple switch with little control state and without even arithmetic units to support sophisticated routing functions on arbitrary topologies. All of the intelligence is in the host nodes. It has the disadvantage that the header tends to be large and usually of variable size. If the switch degree is d and routes of length h are permitted, the header may need to carry $h \log d$ routing bits. This approach is used in MIT Parc and Arctic routers, Meiko CS-2, and Myrinet.

A third approach, which is general purpose and allows for a small fixed-size header, is *table-driven routing*, in which each switch contains a routing table R and the packet header contains a routing field i so that the output port is determined by indexing into the table by the routing field, $o = R[i]$. This is used, for example, in HPPI and ATM switches. Generally, the table entry also gives the routing field for the next step in the route, $o, i' = R[i]$, to allow more flexibility in the table configuration. The disadvantage of this approach is that the switch must contain a sizable amount of routing state, and it requires additional switch-specific messages or some other mechanism for establishing the contents of the routing table. Fairly large tables are required to simulate even simple routing algorithms. This approach is better suited to LAN and WAN traffic where only a few of the possible routes among the collection of nodes are used at a time, most of which are long-lasting connections. By contrast, in a parallel machine there is often traffic among all the nodes.

Traditional networking routers contain a full processor that can inspect the incoming message, perform an arbitrary calculation to select the output port, and build a packet containing the message data for that output. This kind of approach is employed in routers and (sometimes) bridges that connect completely different networks (e.g., those that route between Ethernet, FDDI, ATM) or at least different data link layers; it really does not make sense at the time scale of the communication within a high-performance parallel machine.

10.6.2 Deterministic Routing

A routing algorithm is *deterministic* (or *nonadaptive*) if the route taken by a message is determined solely by its source and destination regardless of other traffic in the network. For example, dimension order routing is deterministic; the packet will follow its path regardless of whether a link along the way is blocked. Dimension order and *e-cube* routing are examples of deterministic algorithms. *Adaptive* routing algorithms allow the route for a packet to be influenced by traffic it meets along the way. For example, in a mesh, the route could zigzag toward its destination if links along the dimension order path were blocked or faulty. In a fat-tree, the upward path toward the common ancestor could steer away from blocked links rather than fol-

lowing a specific path determined when the message was injected into the network. If a routing algorithm only selects shortest paths toward the destination, it is *minimal*; otherwise it is *nonminimal*. Allowing multiple routes between each source and destination is clearly required for adaptation (and fault tolerance), and it also provides a way to spread load over more links. These virtues are enjoyed by source-based and table-driven routing; but the choice is made when the packet is injected. Adaptive routing delays the choice until the packet is actually moving through the network, which clearly makes the switch more complex but has the potential of obtaining better link utilization.

We will concentrate first on deterministic routing algorithms and develop an understanding of some of the most popular routing algorithms and the techniques for proving them deadlock-free before investigating adaptive routing.

10.6.3 Deadlock Freedom

In our discussions of network latency and bandwidth, we have tacitly assumed that messages make forward progress, so it is meaningful to talk about performance. This section shows how to go about proving that a network is deadlock-free. Recall that *deadlock* occurs when a packet waits for an event that cannot occur, for example, when no message can advance toward its destination because the queues of the message system are full and each is waiting for another to make resources available. This can be distinguished from *indefinite postponement*, which occurs when a packet waits for an event that can occur but never does, and from *livelock*, which occurs when the routing of a packet never leads to its destination. Indefinite postponement is primarily a question of fairness, and livelock can only occur with adaptive nonminimal routing. Being free from deadlock is a basic property of well-designed networks that must be addressed from the very beginning.

Deadlock can occur in a variety of situations. A “head-on” deadlock may occur when two nodes attempt to send to each other and each begins sending before either receives. It is clear that if they both attempt to complete sending before receiving, neither will make forward progress. We saw this situation at the user message-passing layer using synchronous send and receive, and we saw it at the node-to-network interface layer. Within the network, it could potentially occur with half-duplex channels or if the switch controller were not able to transmit and receive simultaneously on a bidirectional channel. We should think of the channel as a shared resource that is acquired incrementally, first at the sending end and then at the receiver. In each case, the solution is to ensure that nodes can continue to receive while being unable to send. A reliable network can only be deadlock-free if the nodes are able to remove packets from the network even when they are unable to send packets. (Alternatively, we might recover from the deadlock by eventually detecting a time-out and aborting one or more of the packets, effectively preempting the claim on the shared resource. This does raise the possibility of indefinite postponement, which we will address later.) In this head-on situation there is no routing involved; instead, the problem is due to constraints imposed by the switch design.

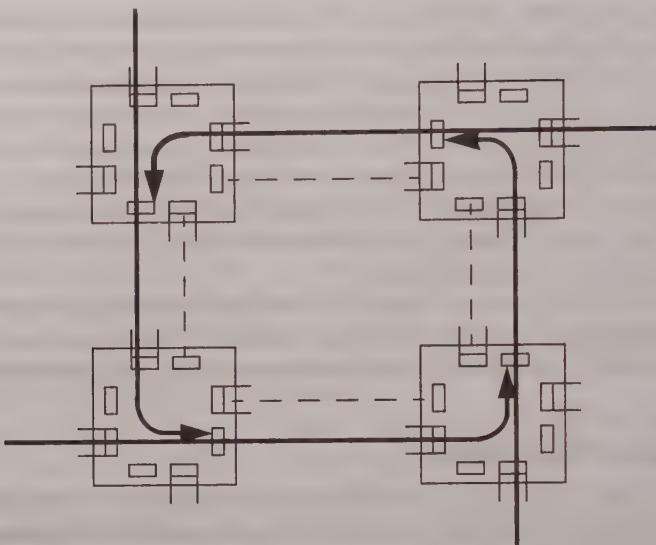


FIGURE 10.19 Examples of network routing deadlock. Each of four switches has four input and output ports. Four packets have each acquired an input port, an output buffer, and an input port, and all are attempting to acquire the next output buffer as they turn left. None will relinquish their output buffer until it moves forward, so none can progress.

A more interesting case of deadlock occurs when multiple messages are competing for resources within the network, as in the routing deadlock illustrated in Figure 10.19. Here we have several messages moving through the network where each message consists of several flits. We should view each channel in the network as having associated with it a certain amount of buffer resources; these may be input buffers at the channel destination, output buffers at the channel source, or both. In our example, each message is attempting to turn to the left, and all of the packet buffers associated with the four channels are full. No message will release a packet buffer until after it has acquired a new packet buffer into which it can move. One could make this example more elaborate by separating the switches with additional switches and channels, but it is clear that the channel resources are allocated incrementally within the network on a distributed basis as a result of messages being routed through, and the resources are nonpreemptible, at least without packet loss. Hence, there is a potential for deadlock.

This routing deadlock can occur with store-and-forward or with cut-through routing, although with cut-through there are greater opportunities for deadlock since each packet stretches over several flit buffers. Only the header flits of a packet carry routing information, so once the head of a message is spooled forward on a channel, all of the remaining flits of the message must spool along on the same channel. Thus, a single packet may hold onto channel resources across several switches. The essential point in these examples is that resources are logically associated with

channels and that messages introduce dependences between these resources as they move through the network.

The basic technique for proving a network deadlock-free is to articulate the dependences that can arise between channels as a result of messages moving through the network and to show that there are no cycles in the resulting channel dependence graph; this implies that no traffic patterns can lead to deadlock. The most common way of doing this is to number the channel resources such that each legal route follows a monotonically increasing (or decreasing) sequence; therefore, no dependence cycles can arise. For a butterfly, this is trivial because the network itself is acyclic. It is also simple for trees and fat-trees as long as the upward and downward channels are independent. For networks with cycles in the channel graph, the situation is more interesting.

To illustrate the basic technique for showing a routing algorithm to be deadlock-free, let us show that Δ_x, Δ_y routing on a k -ary 2D array is deadlock-free. To prove this, view each bidirectional channel as a pair of unidirectional channels numbered independently. Assign each positive- x channel $\langle i, y \rangle \rightarrow \langle i + 1, y \rangle$ the number i , and similarly number the negative- x channels starting from 0 at the most positive edge. Number the positive- y channel $\langle x, j \rangle \rightarrow \langle x, j + 1 \rangle$ the number $N + j$, and similarly number the negative- y edges from the most positive edge. This numbering is illustrated in Figure 10.20. Any route consisting of a sequence of consecutive edges in one x direction, a 90-degree turn, and a sequence of consecutive edges in one y direction is strictly increasing. The channel dependence graph has a node for every unidirectional link in the network, and there is an edge from node A to node B if it is possible for a packet to traverse channel A and then channel B . All edges in the channel dependence graph go from lower-numbered nodes to higher-numbered ones, so there are no cycles in the channel dependence graph (even though there are many cycles in the network).

This proof easily generalizes to any number of dimensions and, since a binary d -cube consists of a pair of unidirectional channels in each dimension, to show that e -cube routing is deadlock-free on a hypercube. Observe, however, that the proof does not apply to k -ary d -cubes in general because the channel number decreases at the wrap-around edges. Indeed, it is not hard to show that for $k > 4$, dimension order routing will even introduce a dependence cycle on a unidirectional torus ($d = 1$).

Notice that the deadlock-free routing proof applies even if only a single flit of buffering is on each channel and that the potential for deadlock exists in a k -ary d -cube even with multiple packet buffers and store-and-forward routing since a single message may fill up all the packet buffers along its route. However, if the use of channel resources is restricted, it is possible to break the deadlock. For example, consider the case of a unidirectional torus with multiple packet buffers per channel and store-and-forward routing. Suppose that one of the packet buffers associated with each channel is reserved for messages destined for nodes with a larger number than their source, that is, packets that do not use wraparound channels. This means that it will always be possible for positive-going messages to make progress. Although wraparound messages may be postponed, the network does not deadlock. This solution is typical of the family of techniques for making store-and-forward

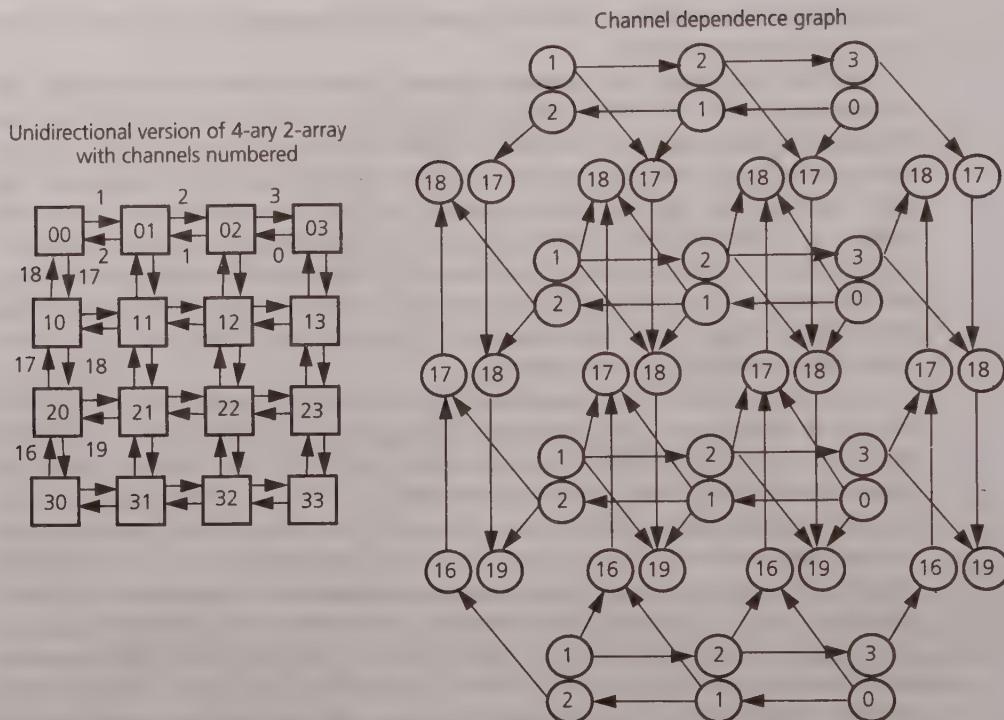


FIGURE 10.20 Channel ordering in the network graph and corresponding channel dependence graph. To show that the routing algorithm is deadlock-free, it is sufficient to demonstrate that the channel dependence graph has no cycles.

packet-switched networks deadlock-free; there is a concept of a *structured buffer pool* (in which certain buffers have specific functions) and the routing algorithm restricts the assignment of buffers to packets to break dependence cycles. This solution is not sufficient for wormhole routing since it tacitly assumes that packets of different messages can be interleaved as they move forward.

Observe that deadlock-free routing does not mean the system is deadlock-free. The network is deadlock-free only as long as it is drained into the NIs, even when the NIs are unable to send. If two-phase protocols are employed, we need to ensure that fetch deadlock is avoided. This means either providing two logically independent networks or ensuring that the two phases are decoupled through NI buffers as discussed in Chapter 7. Of course, the program may still have deadlocks, such as circular waits on locks or head-on collision using synchronous message passing. We have worked our way down from the top, showing how to make each of these layers deadlock-free as long as the next layer below is deadlock-free.

Given a network topology and a set of resources per channel, there are two basic approaches for constructing a deadlock-free routing algorithm: restrict the paths that packets may follow or restrict how resources are allocated. This observation

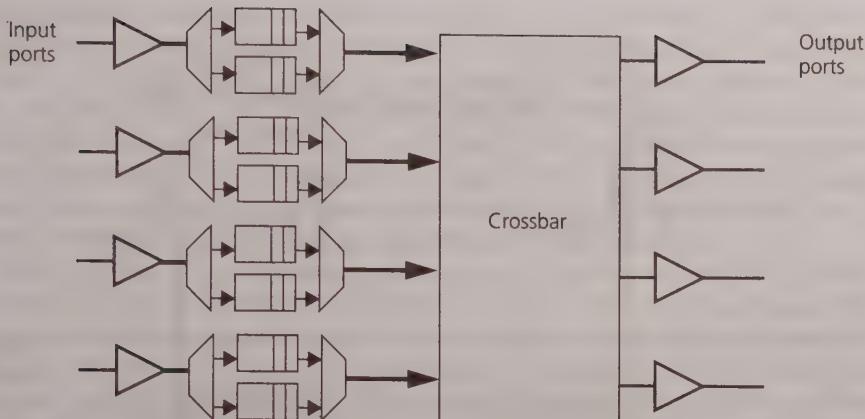


FIGURE 10.21 Multiple virtual channels in a basic switch. Each physical channel is shared by multiple virtual channels. The input ports of the switch split the incoming virtual channels into separate buffers; however, these are multiplexed through the switch to avoid expanding the crossbar.

raises a number of interesting questions. Is there a general technique for producing deadlock-free routes with wormhole routing on an arbitrary topology? Can such routes be adaptive? Is some minimum amount of channel resources required?

10.6.4 Virtual Channels

The basic technique for making networks with wormhole routing deadlock-free is to provide multiple buffers with each physical channel and to split these buffers into a group of *virtual channels*. Going back to our basic cost model for networks, this does not increase the number of links in the network nor the number of switches. In fact, it does not even increase the size of the crossbar internal to each switch since only one flit at a time moves through the switch for each output channel. As indicated in Figure 10.21, it does require additional selectors and multiplexers within the switch to allow the links and the crossbar to be shared among multiple virtual channels per physical channel.

Virtual channels are used to avoid deadlock by systematically breaking cycles in the channel dependence graph. Consider, for example, the four-way routing deadlock cycle of Figure 10.19. Suppose we have two virtual channels per physical channel, and messages at a node numbered higher than their destination are routed on the high channels while messages at a node numbered less than their destinations are routed on the low channels. As illustrated in Figure 10.22, the dependence cycle is broken. Applying this approach to the k -ary d -cube, treat the channel labeling as a radix $d + 1 + k$ number of the form ivx , where i is the dimension, x is the coordinate of the source node of the channel in dimension i , and v is the virtual channel number. In each dimension, if the destination node has a smaller coordinate than the source node in that dimension (i.e., if the message must use a wraparound edge),

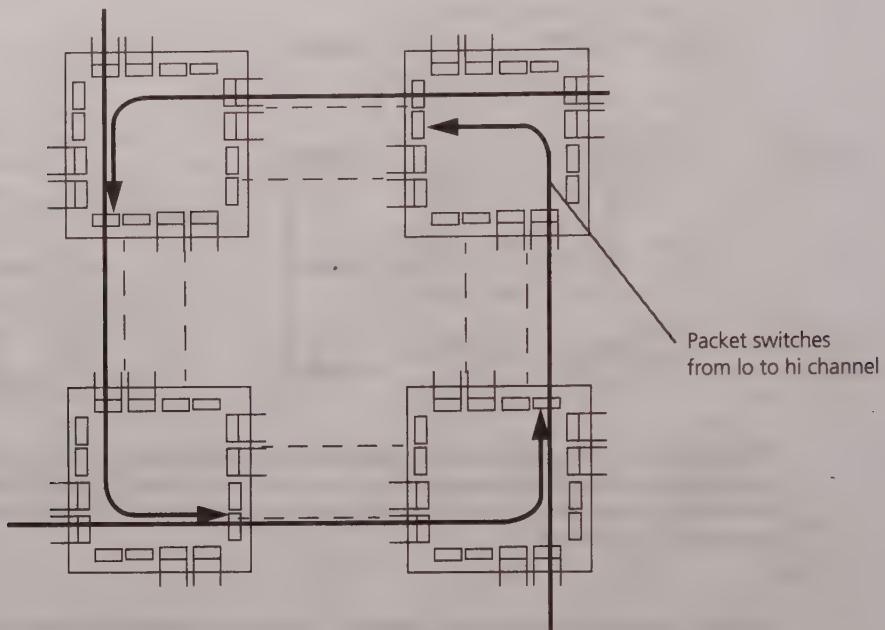


FIGURE 10.22 Breaking deadlock cycles with virtual channels. Each physical channel is broken into two virtual channels; call them lo and hi. The virtual channel “parity” of the input port is used for the output except on turns north to west, which make a transition from lo to hi.

use the $v = 1$ virtual channel in that dimension. Otherwise, use the $v = 0$ channel. You can verify that dimension order routing is deadlock-free with this assignment of virtual channels. Similar techniques can be employed with other popular topologies (Dally and Seitz 1987). Notice that with virtual channels we need to view the routing algorithm as a function $R : C \times N \rightarrow C$ because the virtual channel selected for the output depends on which channel it came in on.

10.6.5 Up*-Down* Routing

Are virtual channels required for deadlock-free wormhole routing on an arbitrary topology? No. If we assume that all the channels are bidirectional, there is a simple algorithm for deriving deadlock-free routes for an arbitrary topology. Not surprisingly, it restricts the set of legal routes. The general strategy is similar to routing in a tree, where routes go up the tree away from the source and then down to the destination. We assume the network consists of a collection of switches, some of which have one or more hosts attached to them. Given the network graph, we want to number the switches so that the numbers increase as we get farther away from the hosts. One approach is to construct a spanning tree of the graph with hosts at the leaves and numbers increasing toward the root. It is clear that for any source host,

any destination host can be reached by an *up*^{*}-*down*^{*} path consisting of a sequence of zero or more up channels (toward higher-numbered nodes), a single turn, and a series of zero or more down channels. Moreover, the set of routes following such paths are deadlock-free. The network graph may have cycles, but the channel dependence graph under *up*^{*}-*down*^{*} routing does not. The up channels form a directed acyclic graph (DAG) and the down channels form a DAG. The up channels depend only on lower-numbered up channels, and the down channels depend only on up channels and higher-numbered down channels.

This style of routing was developed for Autonet (Anderson et al. 1992), which was intended to be self-configuring. Each of the switches contained a processor that could run a distributed algorithm to determine the topology of the network and find a unique spanning tree. Each of the hosts would compute *up*^{*}-*down*^{*} routes as a restricted shortest-paths problem. Then routing tables in the switches could be established. A breadth-first search variant is used by Atomic (Felderman et al. 1994) and Myrinet (Boden et al. 1995), where the switches are passive and the hosts determine the topology by probing the network. Each host runs an algorithm that partitions the network into levels with host nodes at level zero and each switch at the level corresponding to its maximum distance from a host. The numbering is given by a breadth-first search from the highest numbered switch and the algorithm determines the set of source-based routes from the host to the other nodes. A key challenge in automatic mapping of networks, especially with simple switches that only move messages through without any special processing, is determining when two distinct routes through the network lead to the same switch (Mainwaring et al. 1997). One solution is to try returning to the source by reversing routes from previously known switches; another is detecting when identical paths exist from two supposedly distinct switches to the same host.

10.6.6

Turn-Model Routing

We have seen that a deadlock-free routing algorithm can be constructed by restricting the set of routes within a network or by providing buffering with each channel that is used in a structured fashion. How much do we need to restrict the routes? Is there a minimal set of restrictions or a minimal combination of routing restrictions and buffers? An important development in this direction is turn-model routing (Glass and Ni 1992). Consider, for example, a 2D array. There are eight possible turns, which form two simple cycles, as shown in Figure 10.23. (The figure is illustrating cycles appearing in the network involving multiple messages. There is a corresponding cycle in the channel dependence graph.) Dimension order routing prevents the use of four of the eight turns—when traveling in $\mp x$ it is legal to turn in $\mp y$, but once a packet is traveling in $\mp y$ it can make no further turns. The illegal turns are indicated by gray lines in the figure. Intuitively, it seems possible to prevent cycles by eliminating only one turn in each cycle.

Of the 16 different ways to prohibit two turns in a 2D array, 12 prevent deadlock. These consist of the three unique algorithms shown in Figure 10.24 and rotations of these. The west-first algorithm is so named because no turn is allowed into the $-x$

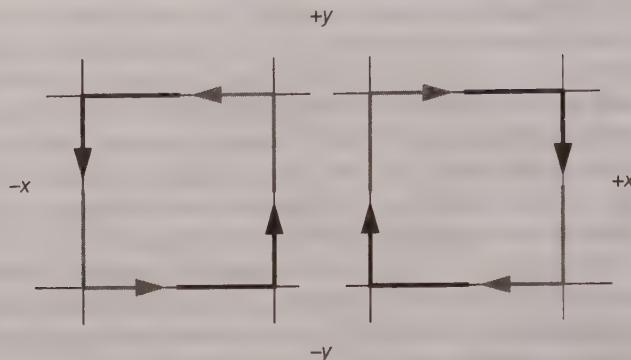


FIGURE 10.23 Turn restrictions of Δx , Δy routing. Dimension order routing on a 2D array prohibits the use of four of the eight possible turns, thereby breaking both of the simple dependence cycles. A deadlock-free routing algorithm can be obtained by prohibiting only two turns.

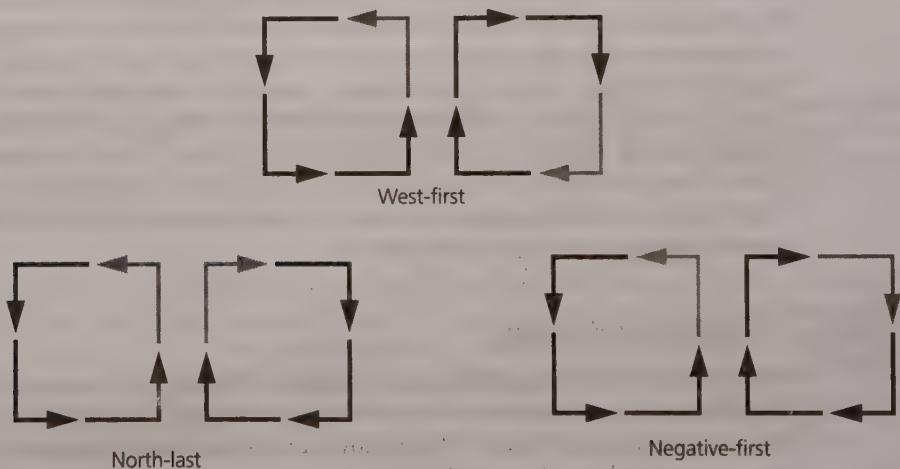


FIGURE 10.24 Minimal turn-model routing in 2D. Only two of the eight possible turns need be prohibited in order to obtain a deadlock-free routing algorithm. Legal turns are shown for three such algorithms.

direction; therefore, if a packet needs to travel in this direction, it must do so before making any turns. Similarly, in north-last there is no way to turn out of the $+y$ direction, so the route must make all its other adjustments before heading in this direction. Finally, negative-first prohibits turns from a positive direction into a negative direction, so the route must go as negative as its needs to before heading in either positive direction.

Each of these turn-model algorithms allows complex, even nonminimal routes. For example, Figure 10.25 shows some of the routes that might be taken under

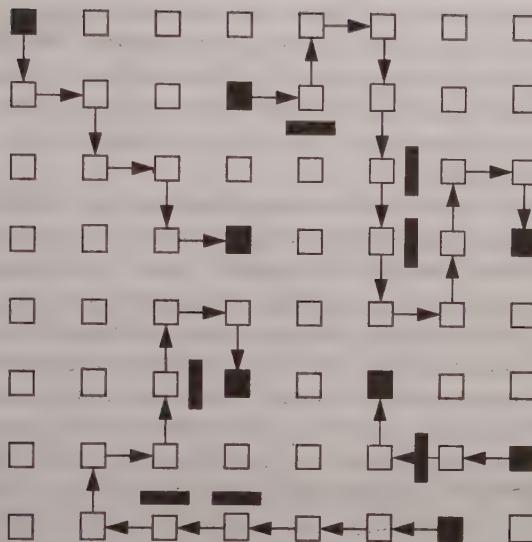


FIGURE 10.25 Examples of legal west-first routes in an 8×8 array. Substantial routing adaptation is obtained with turn-model routing, thus providing the ability to route around faults in a deadlock-free manner.

west-first routing. The elongated rectangles indicate blockages or broken links that might cause such a set of routes to be used. It should be clear that minimal turn models allow a great deal of flexibility in route selection. There are many legal paths between pairs of nodes.

The turn-model approach can be combined with virtual channels, and it can be applied in any topology. (In some networks, such as unidirectional d -cubes, virtual channels are still required.) The basic method is as follows: (1) partition the channels into sets according to the direction they route packets (excluding wraparound edges); (2) identify the potential cycles formed by “turns” between directions; and (3) prohibit one turn in each abstract cycle, being careful to break all the complex cycles as well. Finally, wraparound edges can be incorporated as long as they do not introduce cycles. If virtual channels are present, treat each set of channels as a distinct virtual direction.

Up* down* is essentially a turn-model algorithm with the assumption of bidirectional channels, using only two directions. Indeed, in reviewing the up*-down* algorithm, many shortest paths may conform to the up*-down* restriction, and certainly many nonminimal up*-down* routes are in most networks. Virtual channels allow the routing restrictions to be loosened even further.

10.6.7

Adaptive Routing

The fundamental advantage of loosening the routing restrictions is that it allows multiple legal paths between pairs of nodes. This is essential for fault tolerance. If

the routing algorithm allows only one path, failure of a single link will effectively leave the network disconnected. With multipath routing, it may be possible to steer around the fault. In addition, it allows traffic to be spread more broadly over available channels and thereby improves the utilization of the network. When a vehicle is parked in the middle of the street, it is often nice to have the option of driving around the block.

Simple deterministic routing algorithms can introduce tremendous contention within the network, even when the communication load is spread evenly over independent destinations. For example, Figure 10.26 shows a simple case where four packets are traveling to distinct destinations from distinct sources in a 2D mesh; under dimension order routing they are all forced to travel through the same link. The communication is completely serialized through the bottleneck while links for other shortest paths are unused. A multipath routing algorithm could use alternative channels, as indicated in the right portion of the figure. For any network topology there exist bad permutations (Gottlieb and Kruskal 1984), but simple deterministic routing makes these bad permutations much easier to run across. The particular example in Figure 10.26 has important practical significance. A common global communication pattern is a transpose. On a 2D mesh with dimension order routing, all the packets in a row must go through a single switch before filling in the column.

Multipath routing can be incorporated as an extension of any of the basic switch mechanisms. With source-based routing, the source simply chooses among the legal routes and builds the header accordingly. No change to the switch is required. With table-driven routing, this can be accomplished by setting up table entries for multiple paths. For arithmetic routing, additional control information would need to be incorporated in the header and interpreted by the switch.

Adaptive routing is a form of multipath routing where the choice of routes is made dynamically by the switch in response to traffic encountered en route. Formally, an adaptive routing function is a mapping of the form $R_A : C \times N \times \Sigma \rightarrow C$, where Σ represents the switch state. In particular, if one of the desired outputs is blocked or failed, the switch may choose to send the packet on an alternative channel. Minimal adaptive routing will only route packets along shortest paths to their destination, that is, every hop must reduce the distance to the destination. An adaptive algorithm that allows all shortest paths to be used is *fully adaptive*, otherwise it is *partially adaptive*. An interesting extreme case of nonminimal adaptive routing is what is called “hot potato” routing. In this scheme the switch never buffers packets. If more than one packet is destined for the same output channel, the switch sends one toward its destination and “misroutes” the rest onto other channels.

Adaptive routing is not widely used in current parallel machines, although it has been studied extensively in the literature (Ngai and Seitz 1989; Linder and Harden 1991), especially through the Chaos router (Kostantantindou and Snyder 1991). The CRAY T3E provides minimal adaptive routing in a cube. The nCUBE/3 is to provide minimal adaptive routing in a hypercube. The network proposed for the Tera machine (Alverson et al. 1990) is to use hot potato routing, with 128-bit packets delivered in one 3-ns cycle.

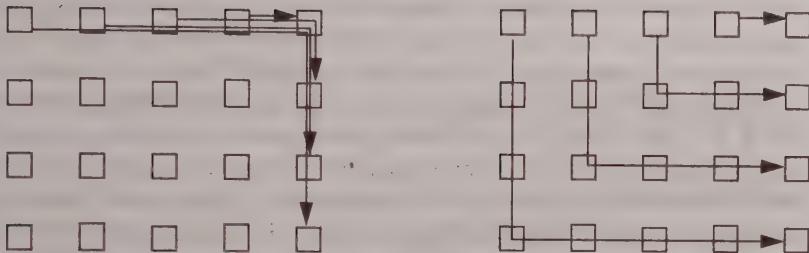


FIGURE 10.26 Routing path conflicts under deterministic dimension order routing. Several messages from distinct source to distinct destination contend for resources under dimension order routing, whereas an adaptive routing scheme may be able to use disjoint paths.

Although adaptive routing has clear advantages, it is not without its disadvantages. Clearly, it adds to the complexity of the switch, which can only make the switch slower. The reduction in bandwidth can outweigh the gains of the more sophisticated routing—a simple deterministic network in its linear operating regime is likely to outperform a clever adaptive network in saturation. In nonuniform networks, such as a d -dimensional array, adaptivity hurts performance on uniform random traffic. Stochastic variations in the load introduce temporary blocking in any network. For switches at the boundary of the array, this will tend to propel packets toward the center. As a result, contention forms in the middle of the array that is not present under deterministic routing. Adaptive routing can cause problems with certain kinds of nonuniform traffic as well, as we will see in Section 10.8.3.

Nonminimal adaptive routing tends to perform poorly as the network reaches saturation because packets traverse extra links and hence consume more bandwidth. The throughput of the network tends to drop off as load is increased rather than flattening at the saturation point, as illustrated in Figure 10.17.

Recently, there have been a number of proposals for low-cost partially and fully adaptive routing that use a combination of a limited number of virtual channels and restrictions on the set of turns (Chien and Kim 1992; Schwiebert and Jayasimha 1995). It appears that most of the advantages of adaptive routing, including fault tolerance and channel utilization, can be obtained with a very limited degree of adaptability.

10.7

SWITCH DESIGN

Ultimately, the design of a network boils down to the design of the switch and how the switches are wired together. The degree of the switch, its internal routing mechanisms, and its internal buffering determine what topologies can be supported and what routing algorithms can be implemented. Now that we understand the higher-level network design issues, let us return to switch design in more detail. Like any

other hardware component of a computer system, a network switch comprises data-path, control, and storage. This basic structure was illustrated at the beginning of the chapter in Figure 10.5. Throughout the early history of parallel computing, switches were built from a large number of low-integration components occupying a board or a rack. Since the mid-1980s, most parallel computer networks are built around single-chip VLSI switches—exactly the same technology as the microprocessor. (This transition began in LANs a decade later.) Thus, switch design is tied to the same technological trends discussed in Chapter 1: decreasing feature size, increasing area, and increasing pin count. We should view modern switch design from a VLSI perspective.

10.7.1 Ports

The total number of pins is essentially the total number of input and output ports times the channel width. Since the perimeter of the chip grows slowly compared to area, switches tend to be pin limited. This pushes designers toward narrow, high-frequency channels. Very high-speed serial links are especially attractive because they use the least pins and eliminate problems with skew across the bit lines in a channel. However, with serial links the clock and all control signals must be encoded within the framing of the serial bit stream. With parallel links, one of the wires is essentially a clock for the data on the others. Flow control is realized using an additional wire, providing a ready/acknowledge handshake.

10.7.2 Internal Datapath

The datapath is the connectivity between each of a set of input ports (i.e., input latches, buffers, or FIFO) and every output port. This is generally referred to as the internal crossbar, although it can be realized in many different ways. A *nonblocking crossbar* is one in which each input port can be connected to a distinct output in any permutation simultaneously. Logically, for an $n \times n$ switch the nonblocking crossbar is nothing more than an n -way multiplexer associated with each destination, as shown in Figure 10.27(a). The multiplexer may be implemented in a variety of different ways, depending on the underlying technology. For example, in VLSI it is typically realized as a single bus with n tristate drivers, shown in Figure 10.27(b). In this case, the control path provides n enable points per output. A technique that is becoming increasingly common is to use a memory as a crossbar by writing for each input port and reading for each output port; see Figure 10.27(c).

It is clear that the hardware complexity of the crossbar is determined by the wires. There are nw data wires in each direction, requiring $(nw)^2$ area. There are also n^2 control wires, which add to this significantly. How do we expect switches to track improvements in VLSI technology? Assume that the area of the crossbar stays constant, but the feature size decreases. The ideal VLSI scaling law says that if the feature size is reduced by a factor of s (including the gate thickness) and the voltage level is reduced by the same factor, then the speed of the transistors improves by a factor of $1/s$, the propagation delay of wires connecting neighboring transistors

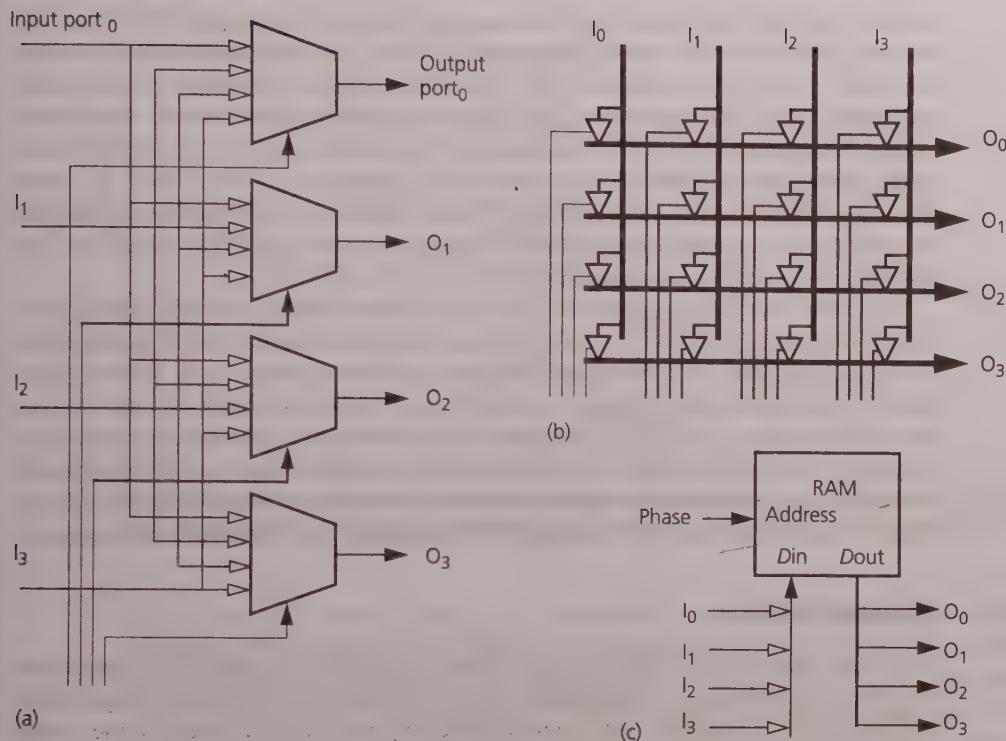


FIGURE 10.27 Crossbar implementations. The crossbar internal to a switch can be implemented as (a) a collection of multiplexers, (b) a grid of tristate drivers, or (c) via a conventional static RAM that time-multiplexes across the ports.

improves by a factor of $1/s$, and the total number of transistors per unit area increases by $1/s^2$ with the same power density. For switches, this means that the wires get thinner and closer together, so the degree of the switch can increase by a factor of $1/s$. Notice that the switch degree improves as only the square root of the improvement in logic density. The bad news is that these wires run the entire length of the crossbar, hence the length of the wires stays constant. The wires get thinner, so they have more resistance. The capacitance is reduced and the net effect is that the propagation delay is unchanged (Bakoglu 1990). In other words, ideal scaling gives us an improvement in switch degree for the same area but no improvement in speed. The speed does improve by a factor of $1/s$ if the voltage level is held constant, but then the power increases with $1/s^2$. Increases in chip area will allow larger degree, but the wire lengths increase and the propagation delays will increase.

Some degree of confusion exists over the term “crossbar.” In the traditional switching literature, multistage interconnection networks that have a single controller are sometimes called crossbars, even though the datapath through the switch is organized as an interconnection of small crossbars. In many cases these are connected in a manner similar to a butterfly topology, called a Banyan network. A

Banyan network is nonblocking if its inputs are sorted, so some nonblocking crossbars are built as batcher sorting networks in front of a Banyan network (Peterson and Davie 1996). An approach that has many aspects in common with Benes networks is to employ a variant of the butterfly, called a delta network, and use two of these networks in series. The first serves to randomize a packet's position relative to the input ports and the second routes to the output port. This is used, for example, in some commercial ATM switches (Turner 1988). In VLSI switches, it is usually more effective to actually build a nonblocking crossbar since it is simple, fast, and regular. The key limit is pins anyway.

It is clear that VLSI switches will continue to advance with the underlying technology, although the growth rate is likely to be slower than the rate of improvement in storage and logic. The hardware complexity of the crossbar can be reduced if we give up the nonblocking property and limit the number of inputs that can be connected to outputs at once. In the extreme, this reduces to a bus with n drivers and n output selects. However, the most serious issue in practice turns out to be the length of the wires and the number of pins, so reducing the internal bandwidth of the individual switches provides little savings and a significant loss in network performance.

10.7.3 Channel Buffers

The organization of the buffer storage within the switch has a significant impact on the switch performance. Traditional routers and switches tend to have large SRAM or DRAM buffers external to the switch fabric whereas, in VLSI switches, the buffering is internal to the switch and comes out of the same silicon budget as the data-path and the control section. There are four basic options: no buffering (just input and output latches), buffers on the inputs, buffers on the outputs, or a centralized shared buffer pool. A few flits of buffering on input and output channels decouples the switches on either end of the link and tends to provide a significant improvement in performance. As chip size and density increase, more buffering is available and the network designer has more options, but still the buffer real estate comes at a premium and its organization is important. Like so many other aspects in network design, the issue is not just how effectively the buffer resources are utilized but how the buffering affects the utilization of other components of the network.

Intuitively, we might expect sharing of the switch storage resources to be harder to implement but to allow better utilization of these resources than partitioning the storage among ports. All of the communication ports need to access the shared pool simultaneously, requiring a very high-bandwidth memory. More surprisingly, sharing the buffer pool on demand can hurt the network utilization in some cases because a single congested output port can hog most of the buffer pool and thereby prevent other traffic from moving through the switch.

Input Buffering

One attractive approach is to provide independent FIFO buffers with each input port, as illustrated in Figure 10.28. Each buffer needs to be able to accept a phit

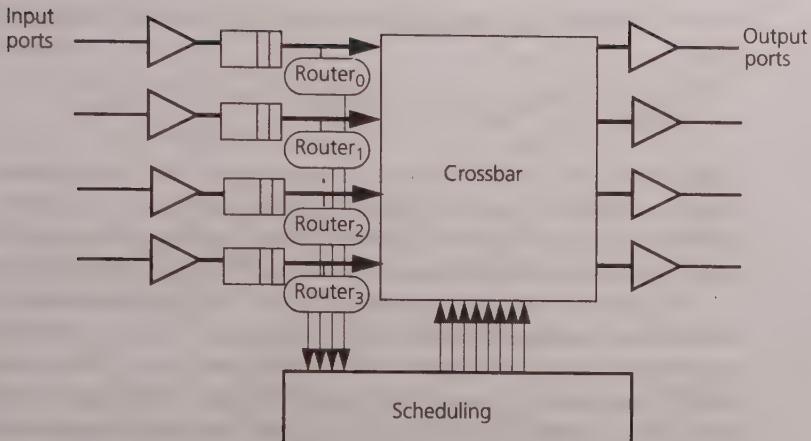


FIGURE 10.28 Input buffered switch. A FIFO is provided at each of the input ports, but the controller can only inspect and service the packets at the heads of the input FIFOs.

every cycle and deliver one phit to an output, so the internal bandwidth of the switch is easily matched to the flow of data coming in. The operation of the switch is relatively simple; it monitors the head of each input FIFO, computes the desired output port of each, and schedules packets to move through the crossbar accordingly. Typically, routing logic is associated with each input port to determine the desired output. This is trivial for source-based routing; it requires an arithmetic unit per input for algorithmic routing and, typically, a routing table per input with table-driven routing. With cut-through routing, the decision logic does not make an independent choice every cycle but only every packet. Thus, the routing logic is essentially a finite state machine, which spools all the flits of a packet to the same output channel before making a new routing decision at the packet boundary (Seitz and Su 1993).

One problem with the simple input buffered approach is the occurrence of “head-of-line” blocking. Suppose that two ports have packets destined for the same output port. One of them will be scheduled onto the output and the other will be blocked. The packet just behind the blocked packet may be destined for one of the unused outputs (there are guaranteed to be unused outputs), but it will not be able to move forward. This head-of-line blocking problem is familiar in our vehicular traffic analogy; it corresponds to having only one lane approaching an intersection. If the car ahead is blocked in attempting to turn, there is no way to proceed down the empty street ahead.

We can easily estimate the effect of head-of-line blocking on channel utilization. If we have two input ports and randomly pick an output for each, the first succeeds and the second has a 50/50 chance of picking the unused output. Thus, the expected number of packets per cycle moving through the switch is 1.5 and, hence, the expected utilization of each output is 75%. Generalizing this, if $E(n, k)$ is the

expected number of output ports covered by k random inputs to an n -port switch, then

$$E(n, k+1) = E(n, k) + \frac{n - E(n, k)}{n}$$

Computing this recurrence up to $k = n$ for various switch sizes reveals that the expected output channel utilization for a single cycle of a fully loaded switch quickly drops to about 65%. Queuing theory analysis shows that the expected utilization in steady state with input queuing is 59% (Karol, Hluchyj, and Morgan 1987).

The impact of head-of-line blocking can be more significant than this simple probabilistic analysis indicates. Within a switch, there may be bursts of traffic for one output followed by bursts for another, and so on. Even though the traffic is evenly distributed, given a large enough window, each burst results in blocking on all inputs (Li 1988). Even if there is no contention for an output within a switch, the packet at the head of an input buffer may be destined for an output that is blocked due to congestion elsewhere in the network. Still, the packets behind it cannot move forward. In a wormhole routed network, the entire worm will be stuck in place, effectively consuming link bandwidth without going anywhere. A more flexible organization of buffer resources might allow packets to slide ahead of packets that are blocked.

Output Buffering

The basic enhancement we need to make to the switch is to provide a way for it to consider multiple packets at each input as candidates for advancement to the output port. A natural option is to expand the input FIFOs to provide an independent buffer for each output port so that packets sort themselves by destination upon arrival, as indicated by Figure 10.29. (This is the kind of switch assumed by the conventional delay analysis in Section 10.5; the analysis is simplified because the switch does not introduce additional contention effects internally.) With a steady stream of traffic on the inputs, the outputs can be driven at essentially 100%. However, the advantages of such a design are not without a cost; additional buffer storage and internal interconnect are required.⁶ Along with the sorting stage and the wider multiplexers, this may increase the switch cycle time or increase its routing delay.

It is a matter of perspective whether the buffers in Figure 10.29 are associated with the input or the output ports. If viewed as output port buffers, the key property is that each output port has enough internal bandwidth to receive a packet from every input port in one cycle. This could be obtained with a single output FIFO, but it would have to run at an internal clock rate of n times that of the input ports.

6. It is possible to provide the capability of the output buffered switch but avoid the storage and interconnect penalty (Joerg 1994). The set of buffers at each input forms a pool and each output has a list of pointers to packets destined for it. The timing requirement in this design is the ability to push n pointers per cycle into the output port buffer rather than n packets per cycle.

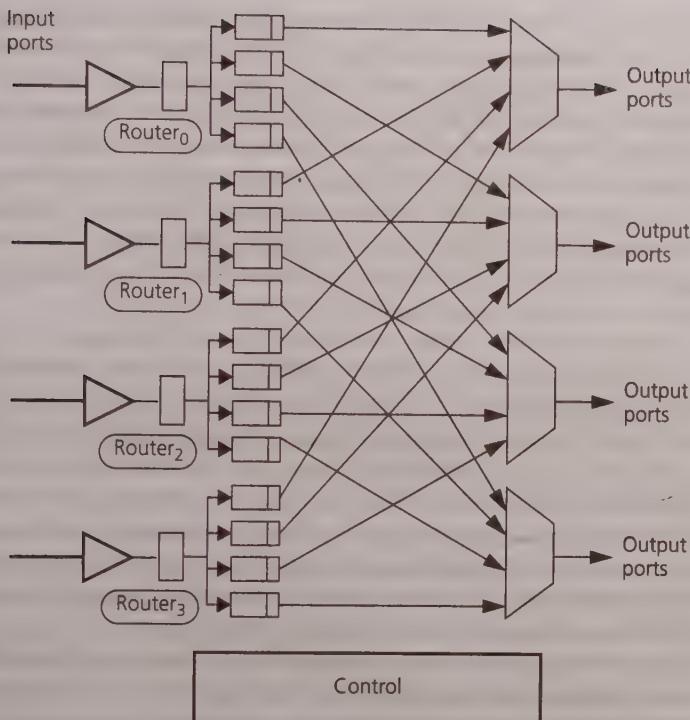


FIGURE 10.29 Switch design to avoid head-of-line blocking. Packets are sorted by output port at each input port so that the controller can schedule a packet for an output port if any input has a packet destined for that output.

Shared Pool

With a shared pool, each of the input ports deposits data into a central memory, and each of the output buffers reads from it. Head-of-line blocking is avoided because input ports can write to the pool regardless of output port, assuming space is available. The challenge is to match the bandwidth of the n -input and n -output ports. A common trick is to make the internal datapath to the pool $2n$ times as wide as the links. Each input port buffers $2n$ phits before writing it to the pool, and each output port gets $2n$ phits at a time. Often these shared pools are built from the SRAM technology used for caches.

Virtual Channels Buffering

Virtual channels suggest an alternative way of organizing the internal buffers of the switch. Recall that a set of virtual channels provides transmission of multiple independent packets across a single physical link. As illustrated in Figure 10.21, to support virtual channels the flow across the link is split upon arrival at an input port

into distinct channel buffers. These are multiplexed together again, either before or after the crossbar, onto the output ports. If one of the virtual channel buffers is blocked, it is natural to consider advancing the other virtual channels toward the outputs. Again, the switch has the opportunity to select among multiple packets at each input for advance to the output ports; however, in this case the choice is among different virtual channels rather than different output ports. It is possible that all the virtual channels will need to route to the same output port, but expected coverage of the outputs is much better. In a probabilistic analysis, we can ask: what is the expected number of distinct output ports covered by choosing among vn requests for n ports, where v is the number of virtual channels?

Simulation studies show that large (256- to 1,024-node) 2-ary butterflies using wormhole routing with moderate buffering (16 flits per channel) saturate at a channel utilization of about 25% under random traffic. If the same 16 flits of buffering per channel are distributed over a larger number of virtual channels, the saturation bandwidth increases substantially. It exceeds 40% with just two virtual channels (8-flit buffers) and is nearly 80% at 16 channels with single-flit buffers (Dally 1990a). While this study keeps the total buffering per channel fixed, it does not really keep the cost constant. Notice that a routing decision needs to be computed for each virtual channel rather than each physical channel, if packets from any of the channels are to be considered for advancement to output ports.

By now you are probably jumping a step ahead to additional cost-performance trade-offs that might be considered. For example, if the crossbar has an input per virtual channel, then multiple packets can advance from a single input at once. This increases the probability of each packet advancing and, hence, the channel utilization. The crossbar increases in size in only one dimension, and the multiplexers are eliminated since each output port is logically a vn -way multiplexer. Switch design allows a great deal of room for innovation.

10.7.4 Output Scheduling

We have seen routing mechanisms that determine the desired output port for each input packet, datapaths that provide a connection from the input ports to the outputs, and buffering strategies that allow multiple packets per input port to be considered as candidates for advancement to the output port. A key missing component in the switch design is the *scheduling algorithm*, which selects the packets to advance in each cycle. Given a selection, the remainder of the switch control asserts the control points in the crossbars or multiplexers and the buffers or latches to effect the register transfer from each selected input to the associated output. As with the other aspects of switch design, there is a spectrum of solutions varying from simple to complex.

A simple approach is to view the scheduling problem as n independent arbitration problems, one for each output port. Each candidate input buffer has a request line to each output port and a grant line from each port, as indicated by Figure 10.30. (The figure shows four candidate input buffers driving three output ports to indicate that routing logic and arbitration input is on a per-input-buffer

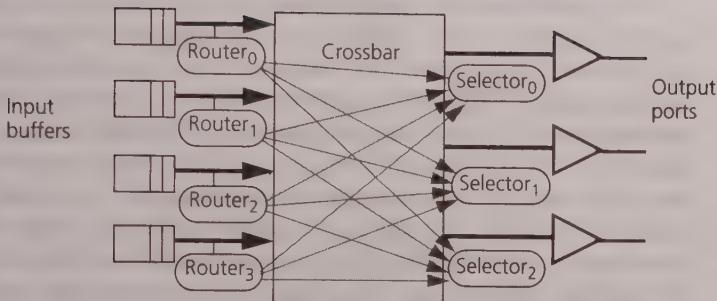


FIGURE 10.30 Control structure for output scheduling. Associated with each input buffer is routing logic to determine the output port, a request line, and a grant line per output. Each output has selection logic to arbitrate among asserted requests and to assert one grant, causing a flit to advance from the input buffer to the output port.

basis rather than per input port.) The routing logic computes the desired output port and asserts the request line for the selected output port. The output port scheduling logic arbitrates among the requests, selects one, and asserts the corresponding grant signal. Specifically, with the crossbar using tristate drivers in Figure 10.27(b), output port j enables input buffer i by asserting control enable e_{ij} . The input buffer logic advances its FIFO as a result of one of the grant lines being asserted.

An additional design question is the arbitration algorithm used for scheduling flits onto the output. Options include static priority, random, round-robin, and oldest-first scheduling. Each of these have different performance characteristics and implementation complexity. Clearly, static priority is the simplest; it is simply a priority encoder. However, in a large network it can cause indefinite postponement. In general, scheduling algorithms that provide fair service to the inputs perform better. Round-robin requires extra state to change the order of priority in each cycle. Oldest-first tends to have the same average latency as random assignment but significantly reduces the variance in latencies (Dally 1990a). One way to implement oldest-first scheduling is to use a control FIFO of input port numbers at each output port. When an input buffer requests an output, the request is enqueued. The oldest request at the head of the FIFO is granted.

It is useful to consider the implementation of various routing algorithms and topologies in terms of Figure 10.30. For example, in a direct d -cube there are $d + 1$ inputs (let's number them i_0, \dots, i_d) and $d + 1$ outputs (numbered o_1, \dots, o_{d+1}) with the host connected to input i_0 and output o_{d+1} . The straight path $i_j \rightarrow o_j$ corresponds to routing in the same dimension; other paths are a change in dimension. With dimension order routing, packets can only increase in dimension as they cross the switch. Thus, the full complement of request/grant logic is not required. Input j need only request outputs $j, \dots, d + 1$ and output j need only grant inputs $0, \dots, j$. Obvious static priority schemes assign priority in increasing or decreasing numerical order.

What are the implementation requirements of adaptive routing? First, the routing logic for an input must compute multiple candidate outputs according to the specific rules of the algorithm, for example, turn restrictions or plane restrictions. For partially adaptive routing, this may be only a couple of candidates. Each output receives several requests and can grant one. (Or if the output is blocked, it may not grant any.) The tricky point is that an input may be selected by more than one output. It will need to choose one, but what happens to the other outputs? Should it iterate on its arbitration and select another input or go idle? This problem can be formalized as one of *on-line bipartite matching* (Karp, Vazirani, and Vazirani 1990). The requests define a bipartite graph with inputs on one side and outputs on the other. The grants (one per output) define a matching of input/output pairs within the request graph. The maximum matching would allow the largest number of inputs to advance in a cycle, which ought to give the highest channel utilization. Viewing the problem in this way, the switch scheduling logic should approximate a fast parallel matching algorithm (Anderson et al. 1992). The basic idea is to form a tentative matching using a simple greedy algorithm, such as random selection among requests at each output followed by selection of grants at each input; then, for each unselected output, try to make an improvement to the tentative matching. In practice, the improvement diminishes after a couple of iterations. Clearly, this is another case of sophistication versus speed. If the scheduling algorithm increases the switch cycle time or routing delay, it may be better to accept a little extra blocking and get the job done faster.

This maximum matching problem applies to the case where multiple virtual channels are multiplexed through each input of the crossbar, even with deterministic routing. (Indeed, the technique was proposed for the AN2 ATM switch to address the situation of scheduling cells from several “virtual circuits.”⁷) Each input port has multiple buffers that it can schedule onto its crossbar input, and these may be destined for different outputs. The selection of the outputs determines which virtual channel to advance. If the crossbar is widened, rather than multiplexing the inputs, the matching problem vanishes and each output can make a simple independent arbitration.

10.7.5 Stacked Dimension Switches

Many aspects of switch design are simplified if there are only two inputs and two outputs, including the control, arbitration, and datapaths. Several designs, including the torus routing chip (Seitz and Su 1993), the J-machine, and the CRAY T3D, have used a simple 2×2 building block and stacked these to construct switches of higher dimension, as illustrated in Figure 10.31. If we have in mind a d -cube, traffic continuing in a given dimension passes straight through the switch in that dimension, whereas if it needs to turn into another dimension it is routed vertically through the

7. Virtual circuits should not be confused with virtual channels. The former is a technique for associating routing of resources along an entire source-to-destination route. The latter is a strategy for structuring the buffering associated with each link.

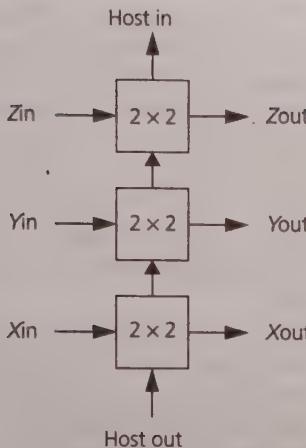


FIGURE 10.31 Stacked dimension switch. Traffic continuing in a dimension passes straight through one 2×2 switch, whereas when it turns to route in another dimension it is routed vertically up the stack.

switch. Notice that this adds a hop for all but the lowest dimension. This same technique yields a topology called *cube-connected cycles* when applied to a hypercube. Each $n \times n$ node of the hypercube is replaced by a ring of n 2×2 nodes.

10.8 FLOW CONTROL

In this section, we consider in detail what happens when multiple flows of data in the network attempt to use the same shared network resources at the same time. Some action must be taken to control these flows. If no data is to be lost, some of the flows must be blocked while others proceed. The problem of flow control arises in all networks and at many levels, but it is qualitatively different in parallel computer networks from that in local and wide area networks. In parallel computers, network traffic needs to be delivered about as reliably as traffic across a bus, and a very large number of concurrent flows occur on very small time scales. No other networking regime has such stringent demands. We will look briefly at some of these differences and then examine in detail how flow control is addressed at the link level and end-to-end in parallel machines.

10.8.1 Parallel Computer Networks versus LANs and WANs

To build intuition for the unique flow control requirements of the parallel machine networks, let us take a little digression and examine the role of flow control in the networks we deal with every day for file transfer and the like. We will look at three examples: Ethernet-style collision-based arbitration, FDDI-style global arbitration, and unarbitrated wide area networks.

In an Ethernet, the entire network is effectively a single shared wire, like a bus, only longer. (The aggregate bandwidth is equal to the link bandwidth.) However, unlike a bus, there is no explicit arbitration unit. A host attempts to send a packet by first checking that the network appears to be quiet and then (optimistically) driving its packet onto the wire. All nodes watch the wire, including the hosts attempting to send. If only one packet is on the wire, every host will “see” it, and the host specified as the destination will pick it up. If there is a collision, every host will detect the garbled signal, including the multiple senders. The minimum that a host may drive a packet (i.e., the minimum channel time) is about 50 μ s; this is to allow time for all hosts to detect collisions.

The flow control aspect is how the retry is handled. On a collision, each sender backs off for a random amount of time and then retries. With each repeated collision, the retry interval from which the random delay is chosen is increased. The collision detection is performed within the network interface hardware, and the retry is handled by the lowest level of the Ethernet driver. If there is no success after a large number of tries, the Ethernet driver gives up and drops the packet. However, the message operation originated from some higher-level communication software layer that has its own delivery contract. For example, the TCP/IP layer will detect the delivery failure via a time-out and engage its own adaptive retry mechanism, just as it does for wide area connection, which we will look at next. The UDP layer will ignore the delivery failure, leaving it to the user application to detect the event and retry. The basic concept of the Ethernet rests on the assumption that the wire is very fast compared to the communication capability of the hosts. (This assumption was reasonable in the mid-1970s when Ethernet was developed.) A great deal of study has been given to the properties of collision-based media access control, but basically as the network reaches saturation, the delivered bandwidth drops precipitously.

Ring-based LANs, such as token ring and FDDI, use a distributed form of global arbitration to control access to the shared medium. A special arbitration token circulates the ring when there is an empty slot. A host that desires to send a packet waits until the token comes around, grabs it, and drives the packet onto the ring. After the packet is sent, the arbitration token is returned to the ring. In effect, flow control is performed at the hosts on every packet as part of gaining access to the ring. Even if the ring is idle, a host must wait for the token to traverse half the ring, on average. (This is why the unloaded latency for FDDI is generally higher than that of Ethernet.) However, under high load, the full link capacity can be used. Again, the basic assumption underlying this global arbitration scheme is that the network operates on a much smaller timescale than the communication operations of the hosts.

In the wide area case, each TCP connection (and each UDP packet) follows a path through a series of switches, bridges, and routers across media of varying speeds between source and destination. Since the Internet is a graph rather than a simple linear structure, at any point a set of incoming flows may have a collective bandwidth greater than the outgoing link. Traffic will back up as a result. Wide area routers provide a substantial amount of buffering to absorb stochastic variations in the flows, but if the contention persists, these buffers will eventually fill up. In this case, most routers will just drop the packets. Wide area links may be stretches of

fibers many miles long, so when the packet is driven onto a link it is hard to know if it will find buffer space available when it arrives at the other end. Furthermore, the flows of data through a switch are usually unrelated transfers. They are not, in general, a collection of flows from within a single parallel program, which imposes a degree of inherent end-to-end flow control by waiting for data it needs before continuing. The TCP layer provides end-to-end flow control and adapts dynamically to the perceived characteristics of the route occupied by a connection. It assumes that packet loss (detected via time-out) is a result of contention at some intermediate point, so when it experiences a loss, it sharply decreases the send rate (by reducing the size of its burst window). It slowly increases this rate (i.e., the window size) as data is transferred successfully (detected via acknowledgments from destination to source) until it once again experiences a loss. Thus, each flow is controlled at the source governed by the occurrence of time-outs and acknowledgments.

Of course, the wide area case operates on a timescale of fractions of a second whereas parallel machine networks operate on a scale of nanoseconds. Thus, one should not expect the techniques to carry over directly. Interestingly, the TCP flow control mechanisms do work well in the context of collision-based arbitration such as Ethernet (partly because the software overhead time tends to give a chance for the network to clear). However, with the emergence of high-speed, switched local and wide area networks, especially ATM, the flow control problem has taken on many more of the characteristics of the parallel machine case. Most commercial ATM switches provide a sizable amount of buffering per link (typically 64 to 128 cells per link) but drop cells when this is exceeded. Each cell is 53 bytes, so at the 155-Mb/s OC-3 rate, a cell transfer time is 2.7 μ s. Buffers fill very rapidly compared to typical LAN/WAN end-to-end times. The TCP mechanisms can be ineffective in the ATM settings when contending with more aggressive protocols, such as UDP, prompting the ATM standardization efforts to include link-level flow and rate control measures.

10.8.2 Link-Level Flow Control

Essentially all parallel machine interconnection networks provide link-level flow control. The basic problem is illustrated in Figure 10.32. Data is to be transferred from an output port of one node across a link to an input port of a node operating autonomously. The storage may be a simple latch, a FIFO, or buffer memory. The link may be short or long, wide or narrow, synchronous or asynchronous. The key point is that, as a result of circumstances at the destination node, storage at the input port may not be available to accept the transfer, so the data must be retained at the source until the destination is ready. This may cause the buffers at the source to fill, and it in turn may exert pressure back on its sources.

The implementation of link-level flow control differs depending on the design of the link, but the main idea is the same. The destination node provides feedback to the source, indicating whether it is able to receive additional data on the link. The source holds onto the data until the destination indicates that it is able. Before we examine how this feedback is incorporated into the switch operation, let us look at how the flow control is implemented on different kinds of links.

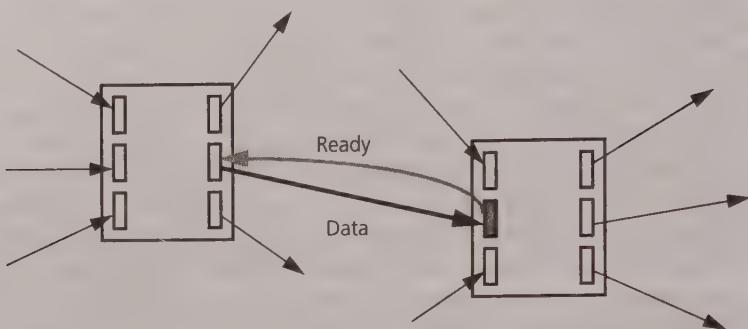


FIGURE 10.32 Link-level flow control. As a result of circumstances at the destination node, the storage at the input port may not be available to accept the data transfer, so the data must be retained at the source until the destination is ready.

With short-wide links, the transfer across the link is essentially like a register transfer within a machine, extended with a couple of control signals. We may view the source and destination registers as being extended with a full-empty bit, as illustrated in Figure 10.33. If the source is full and the destination is empty, the transfer occurs, the destination becomes full, and the source becomes empty (unless it is refilled from its source). With synchronous operation (e.g., in the CRAY T3D, IBM SP-2, TMC CM-5, and MIT J-machine), the flow control determines whether a transfer occurs for the clock cycle. It is easy to see how this is realized with edge-triggered or multiphase level-sensitive designs. If the switches operate asynchronously, the behavior is much like a register transfer in a self-timed design. The source asserts the request (req) signal when it is full and ready to transfer; the destination uses this signal to accept the value (when the input port is available) and asserts an acknowledgement (ack) when it has accepted the data. With short-narrow links, the behavior is similar, except that a series of phits is transferred for each req/ack handshake.

The req/ack handshake can be viewed as the transfer of a single token or credit between the source and the destination. When the destination frees the input buffer, it passes the token to the source (i.e., increases its credit). The source uses this credit when it sends the next flit and must wait until its account is refilled. For long links, this credit scheme is expanded so that the entire pipeline associated with the link propagation delay can be filled. Suppose that the link is sufficiently long that several flits are in transit simultaneously. As indicated by Figure 10.34, it will also take several cycles for acks to propagate in the reverse direction, so a number of acks (credits) may be in transit as well. The obvious credit-based flow control is for the source to keep account of the available slots in the destination input buffer. The counter is initialized to the buffer size. It is decremented when a flit is sent, and the output is blocked if the counter reaches zero. When the destination removes a flit from the input buffer, it returns a credit to the source, which increments the counter. The input buffer will never overflow; there is always room to drain the link into the buffer. This approach is most attractive with wide links, which have dedicated control lines

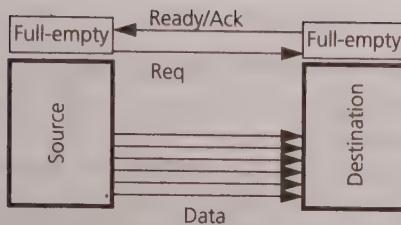


FIGURE 10.33 Simple link-level handshake. The source asserts its request when it has a flit to transmit; the destination acknowledges the receipt of a flit when it is ready to accept the next one. Until that time, the source repeatedly transmits the flit.

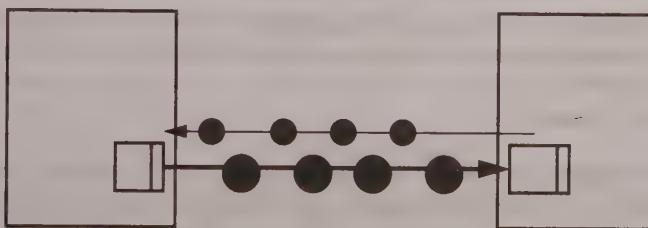


FIGURE 10.34 Transient flits and acks with long links. With longer links, the flow control scheme needs to allow more slack in order to keep the link full. Several flits can be driven onto the wire before waiting for acks to come back.

for the reverse ack. For narrow links that multiplex ack symbols onto the opposite-going channel, the ack per flit can be reduced by transferring bigger chunks of credit. However, the problem remains that the approach is not very robust to the loss of credit tokens.

Ideally, when the flows are moving forward smoothly there is no need for flow control. The flow control mechanism should be a governor that gently nudges the input rate to match the output rate. The propagation delays on the links give the system momentum. An alternative approach to link-level credits is to view the destination input buffer as a staging tank with a low-water mark and a high-water mark, as in Figure 10.35. When the fill level drops below the low mark, a GO symbol is sent to the source, and when it goes over the high mark, a STOP symbol is generated. There must be enough room below the low mark to withstand a full round-trip delay (the GO propagating to the source, being processed, and the first of a stream of flits propagating to the destination). In addition, there must be enough headroom above the high mark to absorb the round-trip's worth of flits that may be in flight. A nice property of this approach is that redundant GO symbols may be sent anywhere below the high mark and STOP symbols may be sent anywhere above the low mark with no harmful effect, so they are simply sent periodically in the two regimes. The fraction of the link bandwidth used by flow control symbols can be reduced by

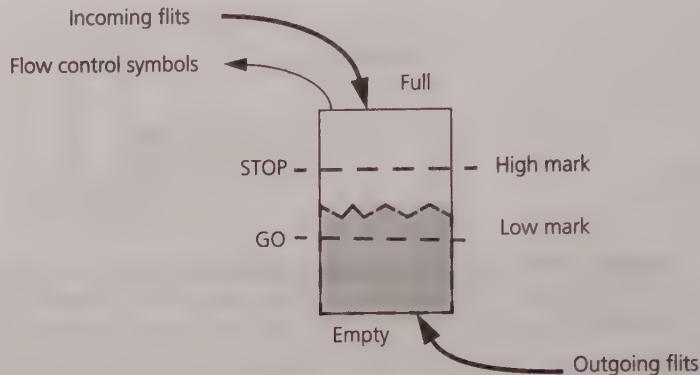


FIGURE 10.35 Slack buffer operation. When the fill level drops below the low mark, a GO symbol is sent to the source, and when it goes over the high mark, a STOP symbol is generated.

increasing the amount of storage between the low and high marks in the input buffer. This approach is used, for example, in Cal Tech routers (Seitz and Su 1993) and the Myrinet commercial follow-on (Boden et al. 1995). Similar techniques are used in modems.

It is worth noting that the link-level flow control is used on host-switch links as well as switch-switch links. In fact, it is generally carried over to the processor-NI interface as well. However, the techniques may vary for these different kinds of interfaces. For example, in the Intel Paragon the 175-MB/s network links are all short with very small flit buffers. However, the NIs have large input and output FIFOs. The communication assist includes a pair of DMA engines that can burst at 300 MB/s between memory and the network FIFOs. It is essential that the output (input) buffer not hit full (empty) in the middle of a burst because holding the bus transaction would hurt performance and potentially lead to deadlock. Thus, the burst is matched to the size of the middle region of the buffer and the high/low marks to the control turnaround between the NI and the DMA controller.

10.8.3 End-to-End Flow Control

Link-level flow control exerts a certain amount of end-to-end control because if congestion persists, buffers will fill up and the flow will be controlled all the way back to the source host nodes, called *back pressure*. For example, if k nodes are sending data to a single destination, they must eventually all slow to an average bandwidth of $1/k$ th the output bandwidth. If the switch scheduling is fair and all the routes through the network are symmetric, back pressure will be enough to affect this. The problem is that by the time the sources feel the back pressure and regulate their output flow, all the buffers in the tree from the hot spot to the sources are full.

Hot Spots

The hot spot problem received quite a bit of attention as the technology reached a point where machines of several hundred to a thousand processors became reasonable (Pfister and Norton 1985). If a thousand processors deliver on average any more than 0.1% of their traffic to any one destination, that destination becomes saturated. If this situation persists, as it might if frequent accesses are made to an important shared variable, a saturation tree forms in front of this destination that will eventually reach all the way to the sources. At this point, all the remaining traffic in the system is severely impeded. The problem is particularly pernicious in butterflies since there is only one route to each destination and a great deal of sharing takes place among routes from one destination. Adaptive routing proves to make the hot spot problem even worse because traffic destined for the hot spot is sure to run into contention and to be directed to alternate routes. Eventually, the entire network clogs up. Large network buffers do not solve the problem; they just delay the onset. The time for the hot spot to clear is proportional to the total amount of hot spot traffic buffered in the network, so adaptivity and large buffers increase the time required for the hot spot to clear after the load is removed.

Various mechanisms have been developed to mitigate the causes of hot spots, such as having all nodes that need to increment a shared variable perform a parallel scan operation, as discussed in Chapter 7, or to implement combining fetch&add operations within the network (Pfister et al. 1985; Gottlieb, Lubachevsky, and Rudolph 1983). However, these only address situations where the problematic traffic is logically related. The more fundamental problem is that link-level flow control is like stopping on the freeway. Once the traffic jam forms, you are stuck. The better solution is not to get on the freeway at such times. With fewer packets inside the network, the normal mechanisms can do a better job of getting traffic to its destinations. This is one of the reasons that the BBN butterfly retracts the circuit established by a message upon collision.

Global Communication Operations

Problems with simple back pressure have been observed with completely balanced communication patterns, such as each node sending k packets to every other node. This occurs in many situations, including transposing a global matrix, converting between blocked and cyclic layouts, or the decimation step of an FFT (Brewer and Kuszmaul 1994; Dusseau et al. 1996). Even if the topology is robust enough to avoid serious internal bottlenecks on these operations, which is true of a fat-tree but not a low-degree dimension order mesh (Leighton 1992), a temporary backlog can have a cascading effect. When one destination falls behind in receiving packets from the network, a backlog begins to form. If priority is given to draining the network, this node will fall behind in sending to the other nodes. They, in turn, will send more than they receive and the backlog will tend to grow.

Simple end-to-end protocols in the global communication routines have been shown to mitigate this problem in practice. For example, a node may wait after

sending a certain amount of data until it has also received this amount, or it may wait for chunks of its data to be acknowledged. These precautions keep the processors more closely in step and introduce small gaps in the traffic flow, which decouples the processor-to-processor interaction through the network. (Interestingly, this technique is employed with the metering lights on heavily used bridges. Periodically, a waveform of cars is injected into the bridge, separated by small gaps. This reduces the stochastic temporary blocking and avoids cascading blockage.)

Admission Control

With shallow, cut-through networks, the latency is low below saturation. Indeed, in most modern parallel machine networks, a single small message (or perhaps a few messages) will occupy an entire path from source to destination. If the remote network interface is not ready to accept the message, it is better to keep it within the source NI rather than blocking traffic within the network. One approach to achieving this is to perform NI-to-NI credit-based flow control. One study examining such techniques for a range of networks (Callahan and Goldstein 1995) indicates that allowing a single outstanding message per pair of NIs gives good throughput and maintains low latency.

10.9 CASE STUDIES

Networks are a fascinating area of study from a practical design and engineering viewpoint because they do one simple operation—move information from one place to another—and yet there is a huge space of design alternatives. Although the most apparent characteristics of a network are its topology, link bandwidth, switching strategy, and routing algorithm, several more characteristics must be specified to completely describe the design. These include the cycle time, link width, switch buffer capacity and allocation strategy, routing mechanism, switch output selection algorithm, and flow control mechanism. Each component of the design can be understood and optimized in isolation, but they all interact in interesting ways to determine the network performance on any particular traffic pattern in the context of the node architecture and the dependences embedded in the programs running on the platform.

This section summarizes a set of concrete network design points in important commercial and research parallel architectures. Using the framework established in this chapter, it systematically outlines the key design parameters.

10.9.1 CRAY T3D Network

The CRAY T3D network consists of a three-dimensional bidirectional torus of up to 1,024 switch nodes, each connected directly to a pair of processors,⁸ with a data rate

8. Special I/O gateway nodes attached to the boundary of the cube include two processors each attached to a two-dimensional switch node connected into the x and z dimensions.

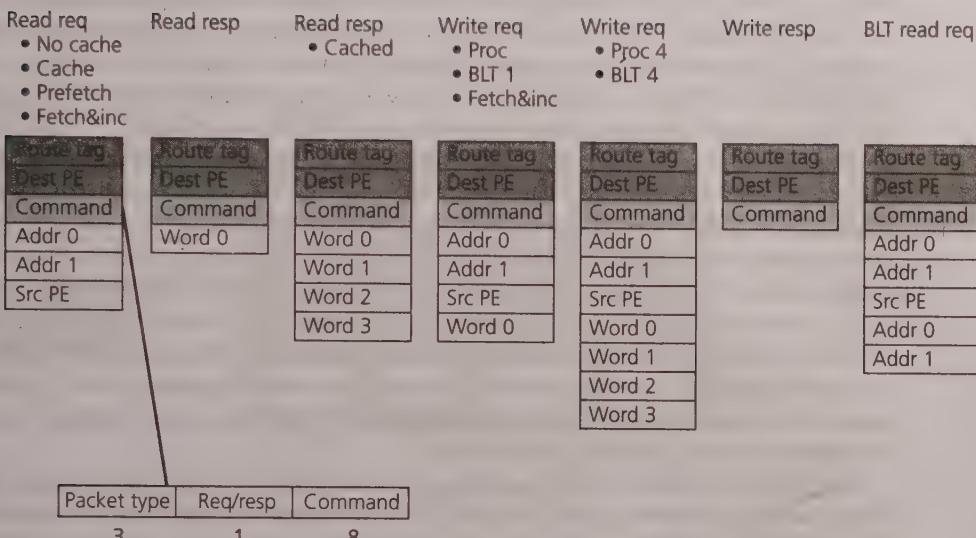


FIGURE 10.36 CRAY T3D packet formats. All packets consist of a series of 16-bit phits, with the first three being the route and tag, the destination processor number, and the command. Parsing and processing of the rest of the packet is determined by the tag and command.

of 300 MB/s per channel. Each node is on a single board, along with two processors and memory. There are up to 128 nodes (256 processors) per cabinet; larger configurations are constructed by cabling together cabinets. Dimension order, cut-through, packet-switched routing is used. The design of the network is very strongly influenced by the higher-level design of the system. Logically independent request and response networks are supported, with two virtual channels each to avoid deadlock, over a single set of physical links. Packets are variable length in multiples of 16 bits, as illustrated by Figure 10.36, and network transactions include various reads and writes plus the ability to start a remote block transfer engine (BLT), which is basically a DMA device. The first phit always contains the route, followed by the destination address, and the packet type or command code. The remainder of the payload depends on the packet type, consisting of relative addresses and data. All of the packet is parity protected, except the routing phit. If the route is corrupted, it will be misrouted, and the error will be detected at the destination because the destination address will not match the value in the packet.⁹

T3D links are short, wide, and synchronous. Each unidirectional link is 16 data and 8 control bits wide, operating under a single 150-MHz clock. Flits and phits are 16 bits. Two of the control bits identify the phit type (00 no info, 01 routing tag, 10

9. This approach to error detection reveals a subtle aspect of networks. If the route is corrupted, there is a very small probability that it will take a wrong turn at just the wrong time and collide with another packet in a manner that causes deadlock within the network; in this unlikely case the end node error detection will not be engaged.

packet, 11 last). The routing tag phit and the last-packet phit provide packet framing. Two additional control bits identify the virtual channel (req-hi, req-lo, resp-hi, resp-lo). The remaining four control lines are acknowledgments in the opposite direction, one per virtual channel. Thus, flow control per virtual channel and phits can be interleaved between virtual channels on a cycle-by-cycle basis.

The switch is constructed as three independent dimension routers, in six 10-K gate arrays. There is a modest amount of buffering in each switch (eight 16-bit parcels for each of four virtual channels in each of three dimensions), so packets compress into the switches when blocked. There is enough buffering in a switch to store small packets. The input port determines the desired output port by a simple arithmetic operation. The routing distance is decremented, and if the result is nonzero the packet continues in its current direction on the current dimension; otherwise, it is routed into the next dimension. Each output port uses a rotating priority among input ports requesting that output. For each input port, there is a rotating priority for virtual channels requesting that output.

The network interface contains eight packet buffers, two per virtual channel. The entire packet is buffered in the source NI before being transmitted into the network. It is buffered in the destination NI before being delivered to the processor or memory system. This store-and-forward delay effectively decouples the network and node operations. In addition to the main data communication network, separate treelike networks for logical-AND (Barrier) and logical-OR (Eureka) are provided.

The presence of bidirectional links provides two possible options in each dimension. A table lookup is performed in the source network interface to select the (deterministic) route consisting of the direction and distance in each of the three dimensions. An individual program occupies a partition of the machine consisting of a logically contiguous subarray of arbitrary shape (under operating system configuration). Shift and mask logic within the communication assist maps the partition-relative virtual node address into a machinewide logical $\langle X, Y, Z \rangle$ coordinate address. The machine can be configured with spare nodes, which can be brought in to replace a failed node. The $\langle X, Y, Z \rangle$ is used as an index for the route lookup, so the NI routing tables provide the final level of translation, identifying the physical node by its $\langle \pm \Delta x, \pm \Delta y, \pm \Delta z \rangle$ route from the source. This routing lookup also identifies which of the four virtual channels is used. To avoid deadlock within either the request or response (virtual) network, the high channel is used for packets that cross the wrap-around links and the low channel otherwise.

10.9.2 IBM SP-1, SP-2 Network

The network used in the IBM SP-1 and SP-2 parallel machines (Abali and Aykanat 1994; Stunkel et al. 1994) is in some ways more versatile than that in the CRAY T3D but of lower performance and without support for two-phase, request-response operations. It is packet switched, with cut-through, source-based routing and no virtual channels. The switch has eight bidirectional 40-MB/s ports and can support a wide variety of topologies. However, in the SP machines, a collection of switches are

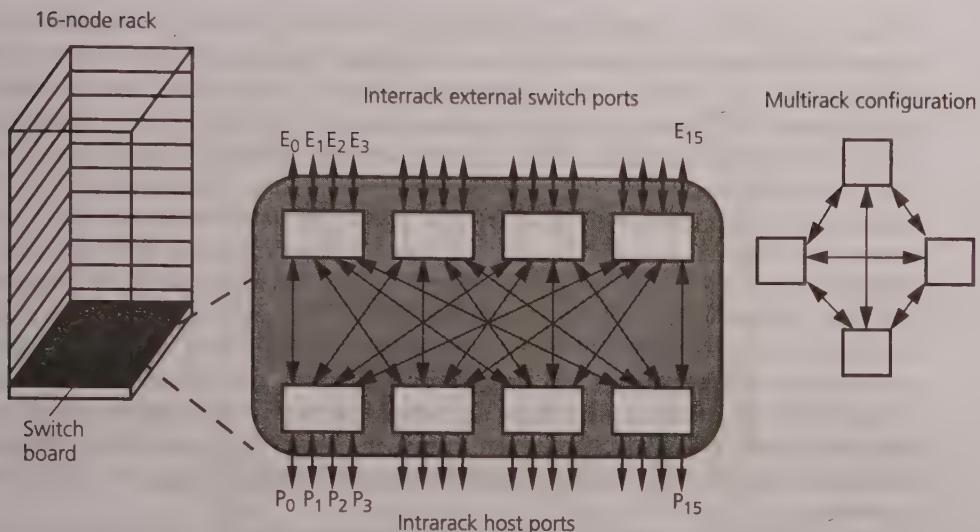


FIGURE 10.37 SP switch packaging. The collection of switches within a rack provides the bidirectional connection between 16 internal ports and 16 external ports.

packaged on a board as a 4-ary 2-dimensional butterfly with 16 internal connections to hosts in the same rack as the switch board and 16 external connections to other racks, as illustrated in Figure 10.37. The rack-level topology varies from machine to machine but is typically a variant of a butterfly. Figure 1.23 in Chapter 1 shows the large IBM SP-2 configuration at the Maui High-Performance Computing Center. Individual cabinets have the first level of routing at the bottom connecting the 16 internal nodes to 16 external links. Additional cabinets provide connectivity between collections of cabinets. The wiring for these additional levels is located beneath the machine room floor. Since the physical topology is not fixed in hardware, the network interface inserts the route for each outgoing message via a table lookup.

Packets consist of a sequence of up to 255 bytes; the first byte is the packet length, followed by one or more routing bytes and then the data bytes. Each routing byte contains two 3-bit output specifiers with an additional selector bit. The links are synchronous, wide, and long. A single 40-MHz clock is distributed to all the switches, with each link tuned so that its delay is an integral number of cycles. (Interboard signals are 100-K ECL differential pairs. Onboard clock trees are also 100-K ECL.) The links consist of 10 wires: 8 data bits, a framing “tag” control bit, and a reverse flow control bit. Thus, the phit is 1 byte. The tag bit identifies the length and routing phits. The flit is 2 bytes; two cycles are used to signal the availability of 2 bytes of storage in the receiver buffer. At any time, a stream of data/tag flits can be propagating down the link while a stream of credit tokens propagate in the other direction.

The switch provides 31 bytes of FIFO buffering on each input port, allowing links to be 16 phits long. In addition, there are 7 bytes of FIFO buffering on each

output and a shared central queue holding 128 8-byte chunks. As illustrated in Figure 10.38, the switch contains both an unbuffered byte-serial crossbar and a 128×64 -bit dual port RAM as the interconnect between the input and output ports. After 2 bytes of the packet have arrived in the input port, the input port control logic can request the desired output. If this output is available, the packet cuts through via the crossbar to the output port, with a minimum routing delay of 5 cycles per switch. If the output port is not available, the packet fills into the input FIFO. If the output port remains blocked, the packet is spooled into the central queue in 8-byte “chunks.” Since the central queue accepts one 8-byte input and one 8-byte output per cycle, its bandwidth matches that of the 8-byte serial input and output ports of the switch. Internally, the central queue is organized as eight FIFO linked lists, one per output port, using an auxiliary 128×7 -bit RAM for the links. One 8-byte chunk is reserved for each output port. Thus, the switch operates in byte-serial mode when the load is low, but when contention forms, it time-multiplexes 8-byte chunks through the central queue, with the inputs acting as a deserializer and the outputs as a serializer.

Each output port arbitrates among requests on an LRU basis, with chunks in the central queue having priority over bytes in input FIFOs. Output ports are served by the central queue in LRU order. The central queue gives priority to inputs with chunks destined for unblocked output ports.

The SP network has three unusual aspects. First, since the operation is globally synchronous, instead of including CRC information in the envelope of each packet, time is divided into 64-cycle “frames.” The last two phits of each frame carry the CRC. The input port checks the CRC and the output port generates it (after stripping it of used routing phits). Second, the switch is a single chip, and every switch chip is shadowed by an identical switch. The pins are bidirectional I/O pins, so one of the chips merely checks the operation of the other. (This will detect switch errors but not link errors.) Finally, the switch supports a circuit-switched “service mode” for various diagnostic purposes. The network is drained free of packets before changing modes.

10.9.3 Scalable Coherent Interface

The Scalable Coherent Interface provides a well-specified case study in high-performance interconnects because it emerged through a standards process rather than as a proprietary design or academic proposal. It was a standard long time in coming but has gained popularity as implementations have gotten under way. It has been adopted by several vendors, although in many cases only a portion of the specification is followed. Essentially, the full SCI specification is used in the interconnect of the HP/Convex Exemplar and in the Sequent NUMA-Q. The CRAY SCX I/O network is based heavily on SCI.

A key element of the SCI design is that it builds around the concept of unidirectional rings, called *ringlets*, rather than bidirectional links. Ringlets are connected by switches to form large networks. The specification defines three layers: a physical

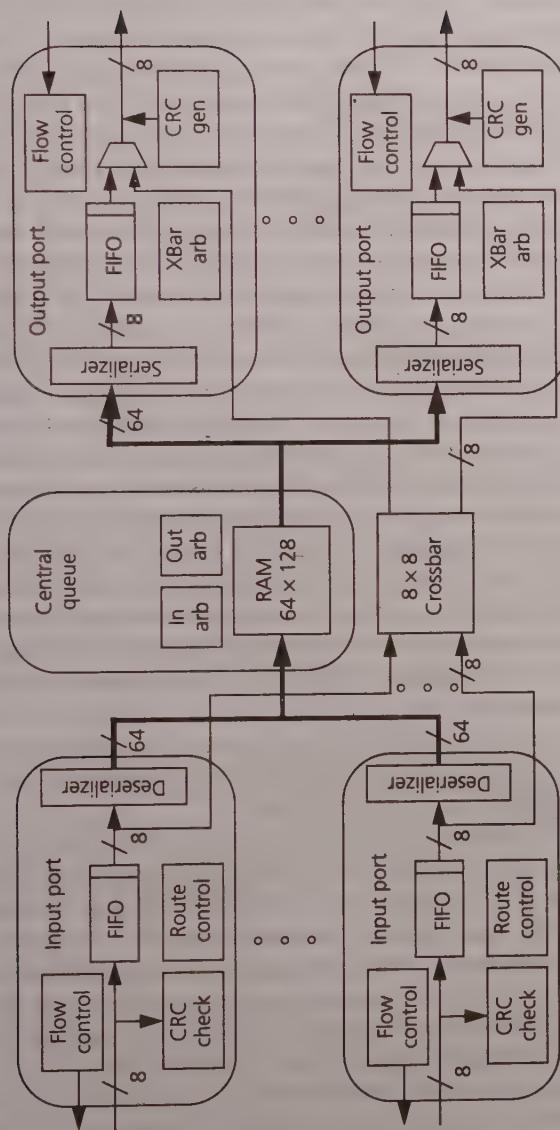


FIGURE 10.38 IBM SP (Vulcan) switch design. The IBM SP switch uses a crossbar for unbuffered packets passed directly from input port to output port, but all buffered packets are shunted into a common memory pool. Arbitration logic schedules flits into and out of the central queue RAM.

layer, a packet layer, and a transaction layer. The physical layer is specified in two 1-GB/s forms. Packets consist of a sequence of 16-bit units, much like the CRAY T3D packets. As illustrated in Figure 10.39, all packets consist of a TargetID, Command, and SourceID, followed by zero or more command-specific units and finally a 32-bit CRC. One unusual aspect is that source and target IDs are arbitrary 16-bit node addresses. The packet does not contain a route. In this sense, the design is like a bus: the source puts the target address on the interconnect, and the interconnect determines how to get the information to the right place. Within a ring, this is simple because the packet circulates the ring and the target extracts the packet. In the general case, switches use table-driven routing to move packets from ring to ring.

An SCI transaction, such as read or write, consists of two network transactions (request and response), each of which has two phases on each ringlet. Let's take this step by step. The source node issues a request packet for a target. The request packet circulates on the source ringlet. If the target is on the same ring as the source, the target extracts the request from the ring and replaces it with an echo packet, which continues around the ring back to the source whereupon it is removed from the ring. The echo packet serves to inform the source that the original packet was either accepted by the target, or rejected, in which case the echo contains a NACK. It may be rejected either because of a buffer-full condition or because the packet was corrupted. The source maintains a timer on outstanding packets so it can detect if the echo gets lost or corrupted. If the target is not on the source ring, a switch node on the ring serves as a proxy for the target. It accepts the packet and provides the echo once it has successfully buffered the packet. Thus, the echo only tells the source that the packet successfully left the ringlet. The switch will then initiate the packet onto another ring along the route to the target. Upon receiving a request, the target node will initiate a response transaction. It too will have a packet phase and an echo phase on each ringlet on the route back to the original source. The request echo packet informs the source of the Transaction ID assigned to the target; this is used to match the eventual response, much as on a split-phase bus.

Rather than have a clean envelope for each of the layers, in SCI they blur together a bit in the packet format where several fields control queue management and retry mechanisms. The control tpr (transaction priority), command mpr (maximum ring priority), and command spr (send priority) together determine one of four priority levels for the packet. The transaction priority is initially set by the requestor, but the actual send priority is established by nodes along the route based on what other blocked transactions they have in their queues. The phase and busy fields are used as part of the flow control negotiation between the source and target nodes.

In the Sequent NUMA-Q, the 18-bit-wide SCI ring is driven directly by the DataPump in a quad at 1-GB/s node-to-network bandwidth. The transport layer follows a strict request-reply protocol. When the DataPump puts a packet on the ring, it keeps a copy of the packet in its outgoing buffer until an *echo* is returned. When a DataPump removes an incoming packet from the ring, it replaces it by a *positive echo*. If a DataPump detects a packet destined for it but does not have space to remove that packet from the ring, it sends a *negative echo*, which causes the sender to retry its transmission (still holding the space in its outgoing buffer).

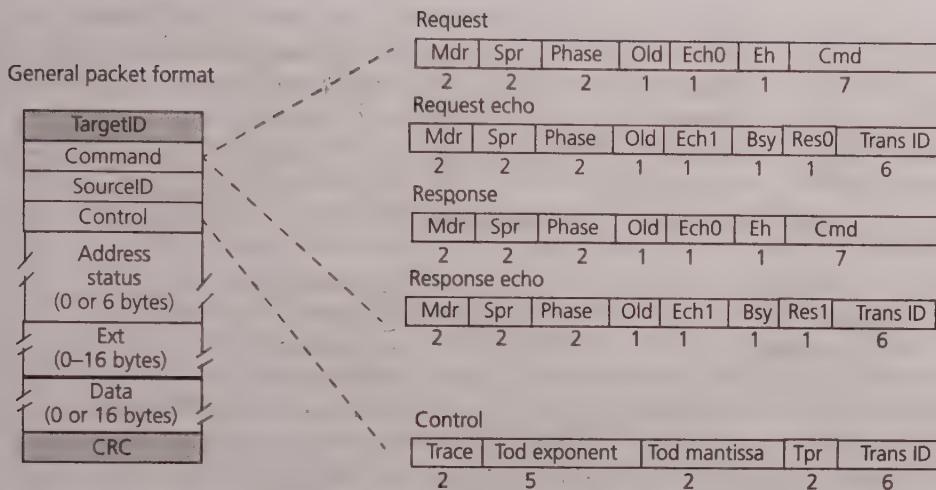


FIGURE 10.39 SCI packet formats. SCI operations involve a pair of transactions: a request and a response. Owing to its ring-based underpinnings, each transaction involves conveying a packet from the source to the target and an echo (going the rest of the way around the ring) from the target back to the source. All packets are a sequence of 16-bit units, with the first three being the destination node, command, and source node, and the final being the CRC. Requests contain a 6-byte address, optional extension, and optional data. Responses have a similar format, but the address bytes carry status information. Both kinds of echo packets contain only the minimal four units. The command unit identifies the packet type through the phase and ech fields. It also describes the operation to be performed (request cmd) or matched (trans id). The remaining fields and the control unit address lower-level issues of how packet queuing and retry are handled at the packet layer.

Since the latency of communication on a ring increases linearly with the number of nodes on it, large high-performance SCI systems are expected to be built out of smaller rings interconnected in arbitrary network topologies. For example, a performance study indicates that a single ring can effectively support four to eight high-performance processing nodes (Scott, Vernon, and Goodman 1992).

10.9.4 SGI Origin Network

The SGI Origin network is based on a flexible switch, called SPIDER, that supports six pairs of unidirectional links, each pair providing over 1.56 GB/s of total bandwidth in the two directions. Two nodes (four processors) are connected to each switch so there are four pairs of links to connect to other switches. This building block is configured in a family of topologies related to hypercubes, as illustrated in Figure 10.40. The links are flexible (long, wide) cables that can be up to 3 meters long. Messages are pipelined through the network, and the latency through the router itself is 41 ns from pin to pin. Routing is table driven, so as part of the network initialization the routing tables are set up in each switch. This allows routing to be programmable so that it is possible to support a range of configurations, to

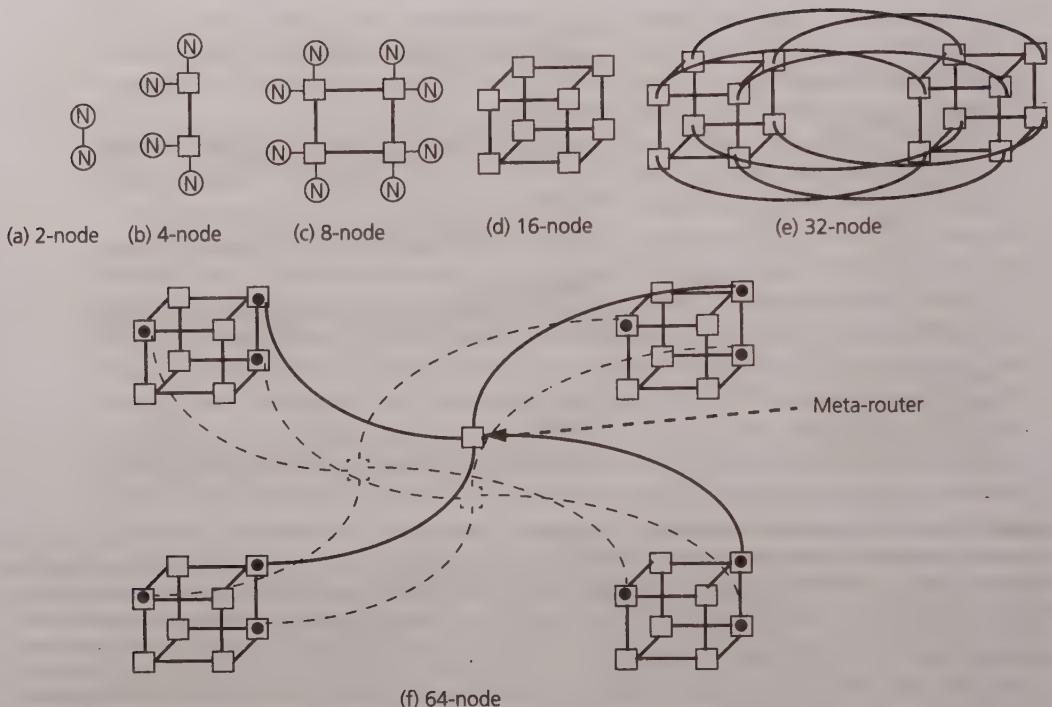


FIGURE 10.40 Network topology and router connections in the SGI Origin multiprocessor. Hypercube topologies are used for up to 32 nodes, as in (a)–(e), and beyond that a fat-tree variant is employed. Configurations (d)–(f) show only the routers and omit the two nodes connected to each for simplicity. Configuration (f) also shows only a few of the fat-cube connections across hypercubes; the routers that connect 16-node subcubes are called meta-routers. For a 512-node (1,024-processor) configuration, each meta-router itself would be replaced by a 5-d router hypercube.

have a partially populated network (i.e., not all ports are used), and to route around faulty components. Availability is also aided by the fact that the routers are separately powered and provide per-link error protection with hardware retry upon error. The switch provides separate virtual channels for requests and replies and supports 256 levels of message priority, with packet aging.

10.9.5 Myricom Network

As a final case study, we briefly examine the Myricom network (Boden et al. 1995) used in several cluster systems. The communication assist within the network interface card was described in Section 7.7. Here we are concerned with the switch that is used to construct a scalable interconnection network. Perhaps the most interesting aspect of its design is its simplicity. The basic building block is a switch with eight bidirectional ports of 160 MB/s each. The physical link is a 9-bit-wide long wire. It

can be up to 25 m, and 23 phits can be in transit simultaneously. The encoding allows control flow, framing symbols, and “gap” (or idle) symbols to be interleaved with data symbols. Symbols are constantly transmitted between switches across the links so that the switch can determine which of its ports are connected. A packet is simply a sequence of routing bytes followed by a sequence of payload bytes and a CRC byte. The end of a packet is delimited by the presence of a gap symbol. The start of a packet is indicated by the presence of a nongap symbol.

The operation of the switch is extremely simple: it removes the first byte from an incoming packet, computes the output port by adding the contents of the route byte to the input port number, and spools the rest of the packet to that port. The switch uses wormhole routing with a small amount of buffering for each input and each output. The routing delay through a switch is less than 500 ns. The switches can be wired together in an arbitrary topology. It is the responsibility of the host communication software to construct routes that are valid for the physical interconnection of the network. If a packet attempts to exit a switch on an invalid or unconnected port, it is discarded. When all the routing bytes are used up, the packet should have arrived at an NIC. The first byte of the message has a bit to indicate that it is not a routing byte, and the remaining bits indicate the packet type. All higher-level packet formatting and transactions are realized by the NIC and the host; the interconnection network only moves the bits.

10.10

CONCLUDING REMARKS

Parallel computer networks present a rich and diverse design space that brings together several levels of design. The physical link-level issues represent some of the most interesting electrical engineering aspects of computer design. In a constant effort to keep pace with the increasing rate of processors, link rates are ever improving. Currently, we are seeing multigigabit rates on copper pairs, and parallel fiber technologies offer the possibility of multigigabyte rates per link in the near future. One of the major issues at these rates is dealing with errors. If bit error rates of the physical medium are in the order of 10^{-19} per bit and data is being transmitted at a rate of 10^9 bytes per second, then an error is likely to occur on a link roughly every 10 minutes. With thousands of links in the machine, errors occur every second. These must be detected and corrected rapidly.

The switch-to-switch layer of design also offers a rich set of trade-offs, including how aspects of the problem are pushed down into the physical layer and how they are pushed up into the packet layer. For example, flow control can be built into the exchange of digital symbols across a link, or it can be addressed at the next layer in terms of packets that cross the link or even one layer higher, in terms of messages sent from end to end. There is a huge space of alternatives for the design of the switch itself, and these are again driven by engineering constraints from below and design requirements from above.

Even the basic topology, switching strategy, and routing algorithm reflect a compromise of engineering requirements and design requirements from below and above. We have seen, for example, how a basic question like the degree of the switch

depends heavily on the cost model associated with the target technology and the characteristics of the communication pattern in the target workload. Finally, as with many other aspects of architecture, there is significant room for debate as to how much of the higher-level semantics of the node-to-node communication abstraction should be embedded in the hardware design of the network itself. The entire area of network design for parallel computers is bound to be exciting for many years to come. In particular, there is a strong flow of ideas and technology between parallel computer networks and advancing, scalable networks for local area and system area communication.

10.11 EXERCISES

- 10.1 Consider a packet format with 10 bytes of routing and control information and 6 bytes of CRC and other trailer information. The payload contains a 64-byte cache block, along with 8 bytes of command and address information. If the raw link bandwidth is 500 MB/s, what is the effective data bandwidth on cache block transfers using this format? How would this change with 32-byte cache blocks? 128-byte blocks? 4-KB page transfers?
- 10.2 Suppose the links are 1 byte wide and operating at 300 MHz in a network where the average routing distance between nodes is $\log_4 P$ for P nodes. Compare the unloaded latency for 80-byte packets under store-and-forward and cut-through routing, assuming 4 cycles of delay per hop to make the routing decision and P ranging from 16 to 1,024 nodes.
- 10.3 Perform the comparison in Exercise 10.2 for 32-KB transfers fragmented into 1-KB packets.
- 10.4 Find an optimal fragmentation strategy for an 8-KB message, as is common for page-sized transfers under the following assumptions. There is a fixed overhead of o cycles per fragment, there is a store-and-forward delay through the source and the destination NI, packets cut through the network with a routing delay of R cycles, and the limiting data transfer rate is b bytes per cycle.
- 10.5 Suppose an $n \times n$ matrix of double-precision numbers is laid out over P nodes by rows. In order to compute on the columns you desire to transpose it to have a column layout. How much data will cross the bisection during this operation?
- 10.6 Consider a 2D torus direct network of N nodes with a link bandwidth of b bytes per second. Calculate the bisection bandwidth and the average routing distance. Compare the estimate of aggregate communication bandwidth using Equation 10.6 with the estimate based on bisection. In addition, suppose that every node communicates only with nodes 2 hops away. Then what bandwidth is available? What if each node communicates only with nodes in its row?
- 10.7 Show how an N -node torus is embedded in an N -node hypercube such that neighbors in the torus (i.e., nodes of distance one) are neighbors in the hypercube. [Hint: observe what happens when the addresses are ordered in a graycode sequence.] Generalize this embedding for higher-dimensional meshes.

- 10.8 Perform the analysis of Equation 10.6 for a hypercube network. In the last step, treat the row as being defined by the graycode mapping of the grid into the hypercube.
- 10.9 Calculate the average distance for a linear array of N nodes (assume N is even) and for a 2D mesh and 2D torus of N nodes.
- 10.10 A more accurate estimate of the communication time can be obtained by determining the average number of channels traversed per packet, h_{ave} , for the workload of interest and the particular network topology. The effective aggregate bandwidth is at most

$$\frac{\|C\|}{h_{ave}} B$$

Suppose the orchestration of a program treats the processors as a $\sqrt{p} \times \sqrt{p}$ logical mesh where each node communicates n bytes of data with its eight neighbors in the four directions and on the diagonal. Give an estimate of the communication time on a grid and a hypercube.

- 10.11 Show how an N -node tree can be embedded in an N -node hypercube by stretching one of the tree edges across two hypercube edges.
- 10.12 Use a spreadsheet to construct a comparison of Figure 10.12 with a design point of 10 cycle routing delays and 16-bit-wide links. Extend this comparison for 1,024-byte messages. What conclusions can you draw?
- 10.13 Verify that the minimum latency is achieved under equal pin scaling in Figure 10.14 when the routing delay is equal to the channel time.
- 10.14 Derive a formula for dimension that achieves the minimum latency under equal bisection scaling based on Equation 10.7.
- 10.15 Under the equal bisection scaling rule, how wide are the links of a 2D mesh with 1 million nodes?
- 10.16 Compare the behavior under load of 2D and 3D cubes of roughly equal size, as in Figure 10.17, for equal pin and equal bisection scaling.
- 10.17 Specify the Boolean logic used to compute the routing function in the switch with $\Delta x, \Delta y$ routing on a 2D mesh.
- 10.18 If $\Delta x, \Delta y$ routing is used, what effect does decrementing the count in the header have on the CRC in the trailer? How can this issue be addressed in the switch design?
- 10.19 Specify the Boolean logic used to compute the routing function in the switch dimension order routing on a hypercube.
- 10.20 Prove that the e -cube routing in a hypercube is deadlock-free.
- 10.21 Construct the table-based equivalent of $\Delta x, \Delta y$ routing in a 2D mesh.
- 10.22 Show that permitting the pair of turns not allowed in Figure 10.24 leaves complex cycles. [Hint: they look like a figure eight.]
- 10.23 Show that with two virtual channels, arbitrary routing in a bidirectional network can be made deadlock-free.

- 10.24 Pick any flow control scheme in the chapter and compute the fraction of bandwidth used by flow control symbols.
- 10.25 Revise latency estimates of Figure 10.14 for stacked dimension switches.
- 10.26 Table 10.1 provides the topologies and an estimate of the basic communication performance characteristics for several important designs. For each, calculate the network latency for 40-byte and 160-byte messages on 16- and 1,024-node configurations. What fraction of this is routing delay and what fraction is occupancy?

Latency Tolerance

In Chapter 1, we saw that while the speed of microprocessors increases by more than a factor of ten per decade, the access time of commodity memories (DRAMs) is only halved. Thus, the latency of memory access in terms of processor clock cycles grows by more than a factor of five in ten years! Multiprocessors greatly exacerbate the problem. In bus-based systems, latency is increased by snooping. In distributed-memory systems, the latency of the network, network interface, and endpoint processing is added to that of accessing the local memory on the node. Caches help reduce the frequency of high-latency accesses, but they are not a panacea: they do not reduce inherent communication, and programs have significant miss rates from other sources as well. Latency usually grows with the size of the machine since more nodes implies more communication relative to computation, more hops in the network for general communication, and likely more contention.

The goal of the protocols developed in the previous chapters has been to reduce the frequency of long-latency events and the bandwidth demands imposed on the communication media while providing a convenient programming model. The goal of the underlying hardware design has been to reduce the latency of data access while maintaining high, scalable bandwidth. Usually, we can improve bandwidth by throwing hardware at the problem (for example, using wider links or richer topologies), but latency is a more fundamental limitation.

So far, we have seen three ways to reduce the effective latency of data access in a multiprocessor system—the first two the responsibility of the system and the third the responsibility of the application.

1. *Reduce the access time to each level of the extended memory hierarchy.* This requires careful attention to detail in making each step in the access path efficient. The processor-to-cache interface can be made very tight. The cache controller needs to act quickly on a miss to reduce the penalty in going to the next level. The network interface can be closely coupled with the node and designed to format, deliver, and handle network transactions quickly. The network itself can be designed to reduce routing delays, transfer time, and congestion delays. With careful design, we can try not to exceed the inherent latency of the technology too much. Nonetheless, the costs add up, and data access takes time.

2. *Structure the system to reduce the frequency of high-latency accesses.* This is the basic job of automatic replication, such as in caches that take advantage of spatial and temporal locality in the program access pattern to keep the most important data close to the processor that accesses it. We can make replication more effective by tailoring the machine structure; for example, by providing a substantial amount of storage in each node.
3. *Structure the application to reduce the frequency of high-latency accesses.* This involves decomposing and assigning computation to processors to reduce inherent communication and structuring access patterns to increase spatial and temporal locality.

In addition to data access and communication, there are other potentially high-latency events, such as synchronization, for which similar efforts can be made. These system and application efforts to reduce latency help greatly, but often they do not suffice. This chapter discusses another approach to dealing with the latency that remains. This approach is to *tolerate the remaining latency*; that is, hide the latency from the processor's critical path by overlapping it with computation or other high-latency events. The processor is allowed to perform other useful work or even data access and communication while the high-latency event is in progress. The key to latency tolerance is in fact parallelism since the overlapped activities must be independent of one another. The basic idea is very simple, and we use it all the time in our daily lives. If you are waiting for one thing (e.g., a load of clothes to finish in the washing machine), you do something else (e.g., run an errand) while you wait.

Latency tolerance cuts across all of the issues discussed in the book and has implications for hardware as well as software, so it serves a useful role in “putting it all together.” As we shall see, the success of latency tolerance techniques depends on both the characteristics of the application as well as on the efficiency of the mechanisms provided by the machine.

Latency tolerance is familiar to us from multiprogramming in uniprocessors. On a disk access, which is a truly long-latency event, the processor does not stall waiting for the access to complete. Rather, the operating system blocks the process that made the disk access and switches in another one, typically from another application, thus overlapping the latency of the access with useful work. The blocked process is resumed later, hopefully after the disk access completes. This is latency tolerance: although the disk access itself does not complete any more quickly, the underlying resource (e.g., the processor) is not stalled and accomplishes other useful work in the meantime. Switching from one process to another via the operating system takes many instructions, but the latency of disk access is high enough that this is worthwhile. In this multiprogramming example, we do not succeed in reducing the execution time of any one process—in fact, we might increase it—but we improve the system’s throughput and utilization. More overall work gets done and more processes complete per unit time.

The latency tolerance in this chapter is different from the preceding example in two important ways. First, we focus primarily on trying to overlap the latency with work from the same application; that is, our goal is to use latency tolerance to reduce the execution time of a given application. Second, we are trying to tolerate

the latencies of the memory and communication systems, not disks. These latencies are much smaller, and the events that cause them are usually not visible to the operating system. Thus, the time-consuming switching of applications or processes via the operating system is not a viable solution.

Before we go further, it may be useful to define a few terms that will be used throughout the chapter. The *latency* of a memory access or communication operation includes all components of the time that elapses from issue by the processor until completion. For communication, this includes the processor overhead, assist occupancy, transit delay, bandwidth-related costs, and contention. The latency may be for one-way transfer of communication or round-trip transfers, which will usually be clear from the context, and it may include the cost of protocol transactions like invalidations and acknowledgments in addition to the cost of data transfer. *Synchronization latency* is the duration that begins when a processor issues a synchronization operation (e.g., lock or barrier) and continues until it gets past that operation; this includes accessing the synchronization variable as well as the time spent waiting for an event that it depends on to occur. *Instruction latency* is the duration that begins when an instruction is issued and ends when it completes in the processor pipeline, assuming no memory, communication, or synchronization latency. Much of instruction latency is already hidden from the processor by pipelining, but some may remain due to long instructions (e.g., floating-point divides) or bubbles in the pipeline. Different techniques are capable of tolerating some subset of these different types of latencies. Our primary focus will be on tolerating communication latencies, whether for explicit or implicit communication, but some of the techniques discussed are applicable to local memory, synchronization, and instruction latencies and hence to uniprocessors as well.

Communication from one node to another that is triggered by a single user operation is called a *message*, regardless of its size. For example, a send in the explicit message-passing abstraction constitutes a message, as does each network transaction triggered by a cache miss that is not satisfied locally in a shared address space (if the miss is satisfied locally, its latency is called *local memory latency* or simply *memory latency*; if satisfied remotely, it is called *communication latency*). Finally, an important aspect of communication is whether it is initiated by the sender (source) or receiver (destination) of the data. Communication is said to be *sender initiated* if the operation that causes the data to be transferred is initiated by the process that has produced or currently holds the data without solicitation from the receiver, for example, an unsolicited send operation in message passing. It is said to be *receiver initiated* if the data transfer is caused or solicited by an operation issued by the process that obtains the data, for example, a read miss to nonlocal data in a shared address space. Sender- and receiver-initiated communication is discussed in more detail later in the context of specific programming models.

The discussion of latency tolerance in this chapter proceeds as follows. Section 11.1 examines the problems that result from memory and communication latency and introduces four approaches to latency tolerance: block data transfer, precommunication, proceeding past an outstanding communication event in the same thread, and multithreading or finding independent work to overlap in other threads of

execution. It also discusses the basic system and application requirements that apply to any latency tolerance technique and the potential benefits and fundamental limitations of exploiting latency tolerance in real systems.

The rest of the chapter examines how the four approaches are applied in the two major communication abstractions. Section 11.2 discusses how the approaches may be used with explicit message passing. The discussion for a shared address space follows and is more detailed since latency is likely to be a more significant bottleneck when communication is performed through individual loads and stores than through flexibly sized transfers. In addition, latency tolerance exposes interesting interactions with the architectural support already provided for a shared address space, and many of the techniques used in this abstraction are applicable to uniprocessors as well. Section 11.3 provides an overview of latency tolerance in a shared address space, and each of the next four sections focuses on one of the approaches, describing the implementation requirements, the performance benefits, the trade-offs and synergies among techniques, and the implications for hardware and software. One of the requirements across all the techniques is that caches be nonblocking or lockup-free, so Section 11.8 discusses techniques for implementing lockup-free caches.

11.1

OVERVIEW OF LATENCY TOLERANCE

To begin our discussion of tolerating communication latency, let us look at a very simple producer-consumer example that will be used throughout the chapter.

A process P_A computes and writes n elements of an array A , and another process P_B reads them. Each process performs some unrelated computation during the loop in which it writes or reads the data in A , and A is allocated in the local memory of the processor on which P_A runs. With no latency tolerance, the process generating the communication would simply perform it a word at a time—explicitly or implicitly as per the communication abstraction—and would wait until each word-length message completes before doing anything else. This will be referred to as the *baseline communication structure*. Figure 11.1(a) shows how the computation might look with explicit message passing, and Figure 11.1(b) shows how it might look with implicit, read-write communication in a shared address space. In the former, it is typically the send operation issued by process P_A that generates the actual communication of the data whereas in the latter it is typically the read of $A[i]$ by P_B .¹ We assume that the read stalls the processor until it completes and that a synchronous send is used (as was described in Section 2.3.6). The resulting timelines for the processes that initiate the communication are shown in Figure 11.2. A process spends most of its time stalled waiting for communication.

1. The examples are in fact not exactly symmetric in their baseline communication structure since in the message-passing version the communication of data happens after each array entry is produced whereas in the shared address space version it happens after the entire array has been produced. However, implementing the fine-grained synchronization necessary for the exactly analogous shared address space version would require synchronization for each array entry, and the communication needed for this synchronization would complicate the discussion. The asymmetry does not affect the discussion of latency tolerance.

P_A	P_B	P_A	P_B
<pre> for i←0 to n-1 do compute A[i]; write A[i]; send (A[i] to P_B); compute f(C[i]); /*unrelated*/ end for </pre>	<pre> for i←0 to n-1 do compute A[i]; write A[i]; compute f(C[i]); end for </pre>	<pre> for i←0 to n-1 do compute A[i]; write A[i]; compute f(C[i]); end for </pre>	<pre> while flag = 0(); for i←1 to n-1 do receive (myA[i]); use myA[i]; compute g(B[i]); /*unrelated*/ end for </pre>
(a) Message passing	(b) Shared address space		

FIGURE 11.1 Pseudocode for the example computation. Pseudocode is shown for explicit message passing (a) and for implicit read-write communication in a shared address space (b), with no latency hiding in either case. The boxes highlight the operations that generate the data transfer.

Time in processor's critical path

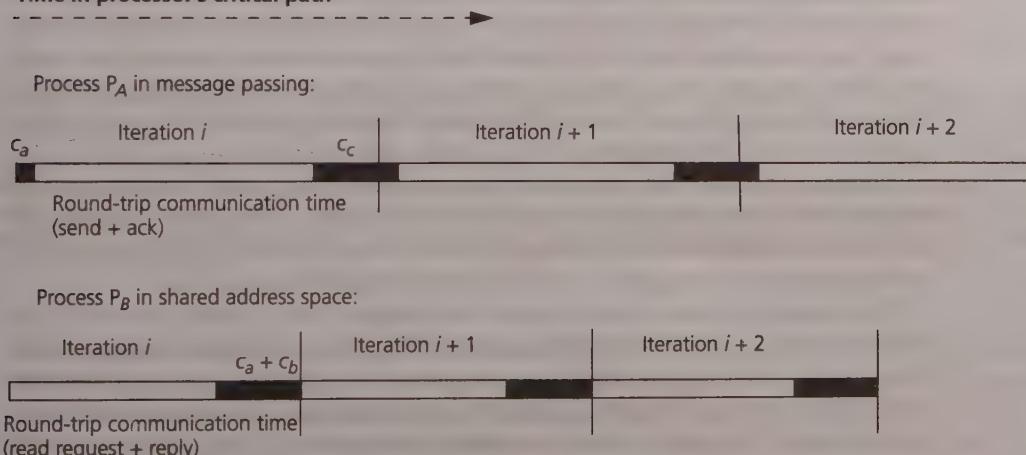


FIGURE 11.2 Timelines for the processes that initiate communication, with no latency hiding. The black segments of the timeline are local processing time (which in fact includes the time spent stalled to access local data), and the white segments are time spent stalled on communication. c_a , c_b , c_c are the durations to compute an array entry $A[i]$ and to perform the unrelated computations $f(B[i])$ and $g(C[i])$, respectively.

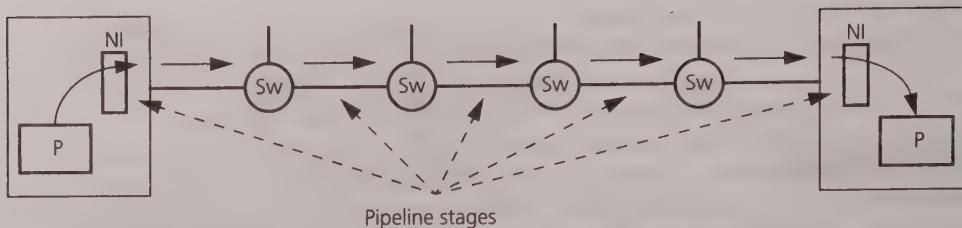


FIGURE 11.3 The network viewed as a pipeline for a communication sent from one processor to another. The stages of the pipeline include the network interfaces (NI) and the hops between successive switches (Sw) on the route between the two processors (P).

11.1.1 Latency Tolerance and the Communication Pipeline

Approaches to latency tolerance are best understood by looking at how the resources in the machine are utilized. From the viewpoint of a processor, the communication architecture from one node to another can be viewed as a pipeline. The stages of the pipeline clearly include the network interfaces at the source and destination, as well as the network links and switches along the way (see Figure 11.3). There may also be stages in the communication assist, the local memory/cache system, and even the main processor, depending on how the architecture manages communication. It is important to recall that the endpoint overhead of instructions incurred on the processor (not necessarily memory accesses) itself cannot be hidden from the processor, though all the other components potentially can. Systems with high endpoint overhead per message therefore have a difficult time tolerating latency. Unless otherwise mentioned, this overview ignores the processor overhead; we assume that most of the endpoint processing of messages is performed on the communication assist and focus on assist occupancy at the endpoints since this has a chance of being hidden too. We also assume that this message initiation and reception cost is incurred as a fixed cost once per message (i.e., it is not proportional to message size).

Not tolerating latency leads to poor utilization of the pipeline and other resources. Figure 11.4 shows the utilization problem for the baseline communication structure described earlier: either the processor or the communication architecture is busy at a given time, and, if the latter, then only one stage of the communication is busy at a time.² The goal in latency tolerance is to overlap the use of these resources as much as possible. From a processor's perspective, there are three major types of overlap that can be exploited. The first is within the *communication pipeline* between two nodes, which allows us to transmit multiple words at a time through the network resources, just like instruction pipelining overlaps the use of different resources in the processor (instruction fetch unit, register files, execute unit, etc.). These words may be from the same message, if messages are larger than a single word, or

2. For simplicity, we ignore the fact that the width of the network link is often less than a word, so even a single word may occupy multiple stages of the network part of the pipeline.

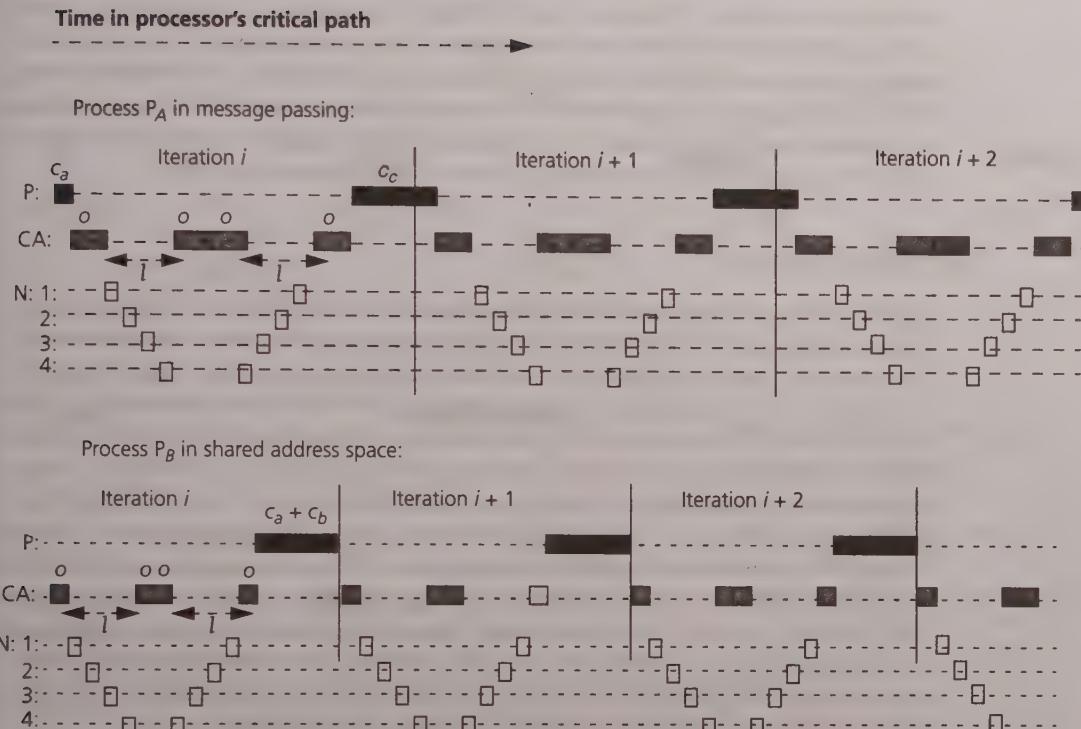


FIGURE 11.4 Timelines for the message-passing and shared address space programs with no latency hiding. Time for a process is divided into that spent on the processor (P) (including the local memory system), on the communication assist (CA), and in the network (N). The numbers under the network category indicate different hops or links in the network on the path of the message. The endpoint overhead \circ in the message-passing case is shown to be larger than in the shared address space, which we assume to be hardware supported. l is the network latency, which is the time spent that does not involve the processing node itself.

from different messages. The second exploits the overlap across different point-to-point communication pipelines, in different portions of the network, by having communication outstanding with different nodes at once. In both cases, from a processor's perspective the communication of one word is overlapped with that of other words, so we call this overlapping *communication with communication*. The third type of overlap is that of *computation with communication*; that is, a processor continues to do useful local work while a communication operation it has initiated is in progress.

11.1.2 Approaches

There are four key approaches to exploiting this overlap of hardware resources and thus tolerating latency. The first, which is called *block data transfer*, is to make individual messages larger so they communicate more than a word and can be pipelined

through the network. The other three overlap a message with computation or with other messages and are thus complementary to block data transfer. They are *precommunication*, *proceeding past communication in the same thread*, and *multithreading*. Each approach can be used in both the shared address space and message-passing abstractions, although the specific techniques and the support required are different in the two cases. A brief introduction to each of the approaches follows.

Block Data Transfer

Making messages larger has several advantages. First, it exploits the communication pipeline between the two nodes to overlap communication with computation. The processor sees the latency of the first word in the message, but subsequent words arrive every network cycle or so, limited only by the rate or bandwidth of the pipeline. Second, it amortizes the per-message endpoint overhead over the large amount of data being sent. Third, depending on how packets are structured, it may also amortize the per-packet routing and header information. Finally, a large message may require only a single acknowledgment rather than one per word, which can reduce latency as well as traffic and contention. These advantages are similar but on a different scale to those obtained by using long cache blocks to transfer data from memory to cache in a uniprocessor. Figure 11.5 shows the effect of employing a single large message in the explicit message-passing case while still using synchronous messages and without overlapping computation. For simplicity of illustration, the network stages have been collapsed into one (pipelined) stage.

Although making messages larger helps keep the pipeline between two nodes busy, it does not in itself keep the processor or communication assist busy while a message is in progress or keep other paths through the network busy. The other approaches address these opportunities as well. They are complementary to block data transfer in that they are applicable whether messages are large or small. Let us examine them one by one.

Precommunication

Generating the communication before the point where the operation naturally appears in the program so that it is partially or entirely completed before data is actually needed can be done either in software, by inserting a precommunication operation earlier in the code, or in hardware, by detecting the opportunity and issuing the communication operation early.³ The operations that actually use the data typically remain where they are in the program. Of course, the precommunication transaction itself should not stall the processor until it completes, or overlap will not

3. Recall that several of the techniques, including this one, are applicable to hiding local memory access latency as well, even though we are speaking in terms of communication since we can think of local access as communication with the local memory system.

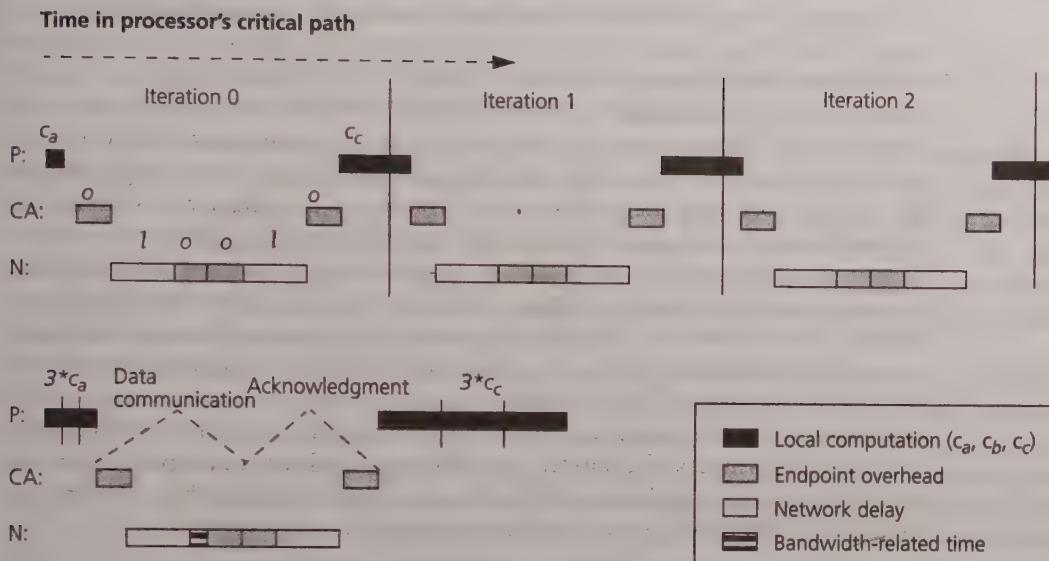


FIGURE 11.5 Effect of making messages larger on the timeline for the example message-passing program. The figure assumes that three iterations of the loop are executed. The sender process (P_A) first computes the three values $A[0..2]$ to be sent (the computation c_a), then sends them in one message, and then performs all three pieces of unrelated computation $f(C[0..2])$, that is, the computation c_c . The overhead of the data communication is amortized, and only a single acknowledgment is needed. A bandwidth-related cost is added to the data communication for the time taken to get the remaining words into the destination node after the first word arrives, but this is tiny compared to the savings in latency and overhead. This additional bandwidth cost is not required for the acknowledgment, which is also small.

be achieved. Many forms of precommunication require that long-latency events be predictable so that hardware or software can anticipate them and issue them early.

Sender-initiated communication is naturally initiated soon after the sender produces the data, so the data may reach the receiver before it is actually needed, resulting in a form of precommunication for free for the receiver. On the other hand, it may be difficult for the sender to move the communication any earlier to hide the latency that the sender sees. Actual precommunication, by generating the communication operation early, is therefore more common in receiver-initiated communication where communication is naturally initiated when the data is needed, which may be long after it has been produced.

Proceeding Past Communication in the Same Thread

The communication operation may be generated where it naturally occurs in the program, but the processor is allowed to proceed past it and find other independent computation or communication that would come later in the same process or thread

of execution. Thus, while precommunication causes the communication to be overlapped with other instructions from that thread that are before the point where the communication-generating operation appears in the original program, this technique causes communication to be overlapped with instructions from later in the thread. These instructions may themselves cause communication or memory accesses, or there may be overlap with those activities as well. As we might imagine, this latency tolerance method is generally easier to use effectively with sender-initiated communication since the instructions that immediately follow the communication operation on the sender do not depend on that communication operation and can therefore be easily overlapped. In receiver-initiated communication, a receiver naturally tries to access data only just before it is actually needed, so there is not much independent work to be found in between the communication and the use. It is, of course, possible to delay the actual use of the data by trying to push it further down in the instruction stream, or equivalently to find independent work even beyond the use, and compilers and processor hardware can exploit some overlap in this way. In either case, either hardware or software must check that the data transfer has completed before executing an instruction that depends on it.

Multithreading

This approach is similar to the previous case, except that the independent work is found by switching to another thread that has been mapped to run on the same processor. This makes the latency of receiver-initiated communication easier to hide than the previous case, and so this method lends itself easily to hiding latency from either a sender or a receiver. In fact, since the overlap here is with other threads, multithreading is the approach that is least concerned with the type and structure of the latency to be tolerated and is in this sense the most general technique. However, multithreading implies that a given processor must have multiple threads that are concurrently executable so that it can switch from one thread to another when a long-latency event is encountered. The multiple threads may be from the same parallel program or from completely different programs, as in our earlier multiprogramming example. A multithreaded program is usually no different from an ordinary parallel program; it is simply decomposed and assigned among P processes, where P is larger than the actual number of physical processors p , and P/p threads are mapped to the same physical processor. Thus, multithreading requires that the additional parallelism needed for latency tolerance be explicit in the form of additional threads.

11.1.3 Fundamental Requirements, Benefits, and Limitations

Before we apply these approaches to specific programming models and systems, it is useful to understand the fundamental requirements, benefits, and limitations of latency tolerance, regardless of the technique used. This basic analysis bounds the extent of the performance improvement we can expect. It is based solely on overlap in the use of resources and on the occupancies of these resources.

Requirements

Tolerating latency requires extra parallelism, increased bandwidth, and, in many cases, more sophisticated hardware and protocols.

- *Extra parallelism*, or *slackness*. Since the overlapped activities (computation or communication) must be independent of one another, the parallelism in the application must be greater than the number of processors used. The additional parallelism may be explicit in the form of additional threads, as in multithreading, or it may exist within a thread. Even communicating two words from the same process in parallel implies a lack of total serialization between them and hence extra parallelism.
- *Increased bandwidth*. Whereas tolerating latency may reduce the execution time, it does not reduce the amount of communication performed. The same communication performed in less time means a higher rate of communication per unit time and hence a larger bandwidth requirement imposed on the communication architecture. In fact, if the bandwidth requirements increase much beyond the bandwidth afforded by the machine, the resulting resource contention may slow down other unrelated transactions, and latency tolerance may hurt performance rather than help it.
- *More sophisticated hardware and protocols*. Except for the case of making messages larger, the processor must be allowed to proceed past a long-latency operation before the operation completes, and it must be allowed to have multiple outstanding long-latency operations if they are to be overlapped with one another and not just with computation.

These requirements imply that all latency tolerance techniques have significant costs. We should therefore try to use algorithmic techniques to reduce the frequency of high-latency events before relying on techniques to tolerate latency. The fewer the long-latency events, the less aggressively we need to hide latency.

Potential Benefits

A simple analysis can give us bounds on the performance benefits we might expect from latency tolerance, thus establishing realistic expectations. Let us focus on tolerating the latency of communication and assume that the latency of local memory references is not hidden. Suppose the execution time as seen by a processor has the following profile when no latency tolerance is used: T_c cycles are spent computing locally, T_{ov} cycles processing message overhead on the processor, T_{occ} (occupancy) cycles on the communication assist, and T_l cycles waiting for message transmission in the network. If we can assume that other resources can be perfectly overlapped with the activity on the main processor, then the potential speedup can be determined by a simple application of Amdahl's Law. The processor must be occupied for $T_c + T_{ov}$ cycles; the maximum latency that we can hide from the processor is $T_l + T_{occ}$, so the maximum speedup due to latency hiding is

$$\frac{T_c + T_{ov} + T_{occ} + T_l}{T_c + T_{ov}}$$

or

$$(1 + \frac{T_{occ} + T_l}{T_c + T_{ov}}).$$

This limit is an upper bound since it assumes perfect overlap of resources and no extra cost imposed by latency tolerance. However, it gives us a useful perspective. For example, if the process originally spends at least as much time computing locally or in processor overhead as stalled on the communication system, then the maximum speedup that can be obtained from tolerating communication latency is a factor of two. If the original communication stall time is overlapped only with processor activity, not with other communication, then the maximum speedup is a factor of two, regardless of how much communication latency there is to hide. This is illustrated in Figure 11.6.

How much latency can actually be hidden depends on many factors involving the application and the architecture. Relevant application characteristics include the structure of the communication and how much other work can be overlapped with it. Architectural issues are: how much of the endpoint processing is performed on the main processor versus on the assist; can communication be overlapped with computation, other communication, or both; how many messages involving a given processor may be outstanding at a time; to what extent can endpoint overhead processing be overlapped with data transmission in the network for the same message; and what are the occupancies of the assist and the stages of the network pipeline.

Figure 11.7 illustrates the effects on the timeline for a few different kinds of message structuring and overlap. The figure is merely illustrative. For example, it assumes that overhead per message is quite high relative to transit latency and does not consider contention anywhere. Under these assumptions, larger messages are often more attractive than many small overlapped messages because they amortize overhead. Further, with small messages the pipeline rate might be limited by the endpoint processing of each message (which determines the gap between messages) rather than by the network link speed. Other assumptions may lead to different results. Exercise 11.2 looks more quantitatively at an example of communication that only overlaps with other communication.

Clearly, some components of communication latency are easier to hide than others. For example, instruction overhead incurred on the processor cannot be hidden from that processor, and latency incurred off node—either in the network or in the other node being communicated with—is generally easier to hide by overlapping with other messages than occupancy incurred on the assist or elsewhere within the node. Let us examine some of the key limitations that may prevent us from achieving the upper bounds on latency tolerance.

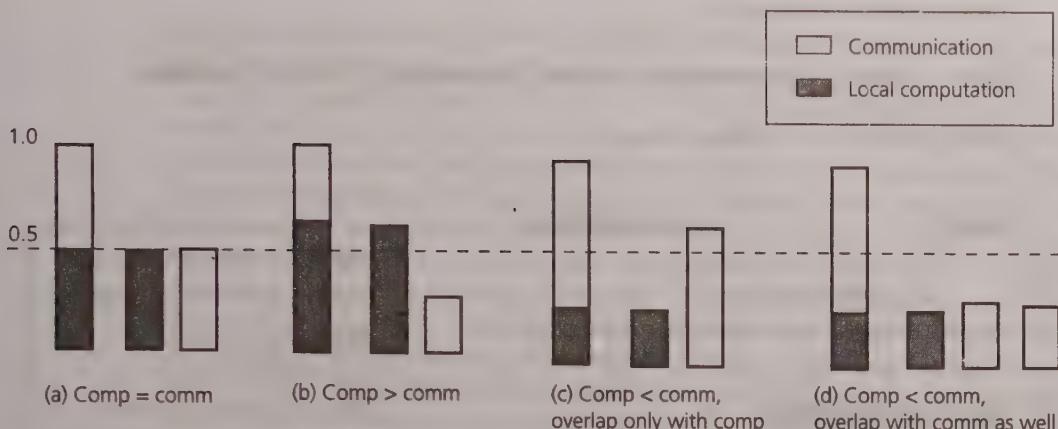


FIGURE 11.6 Bounds on the benefits from latency tolerance. Each figure shows a different scenario. Gray identifies computation time, and white identifies communication time in each case. The bar on the left shows the time breakdown without latency hiding, normalized to 1.0 units, whereas the bars on the right show the situation with latency hiding. When computation time (Comp) equals communication time (Comm), shown in (a), the upper bound on speedup is 2. When computation exceeds communication, the upper bound is less than 2 since we are limited by computation time (b). The same is true of the case in which communication exceeds computation but can be overlapped only with computation (c). The way to obtain a better speedup than a factor of two is to have communication time originally exceed computation time but to let communication be overlapped with computation as well (d).

Limitations

The major limitations can be divided into three classes: application limitations, limitations of the communication architecture, and processor limitations.

Application Limitations The amount of independent computation time that is available to overlap with the latency may be limited, so all the latency may not be hidden and the processor will have to stall for some of the time. Even if the program has enough work and extra parallelism, the structure of the program may make it difficult for the system or programmer to identify the concurrent operations and orchestrate the overlap, as we shall see when we discuss specific latency tolerance mechanisms.

Communication Architecture Limitations The communication architecture may restrict the number of messages or the number of words that can be outstanding from a node at a time, and the performance parameters of the communication architecture may limit the latency that can be hidden (Culler 1994).

With only one message outstanding, independent computation can be overlapped with both assist processing and network transmission. However, assist processing

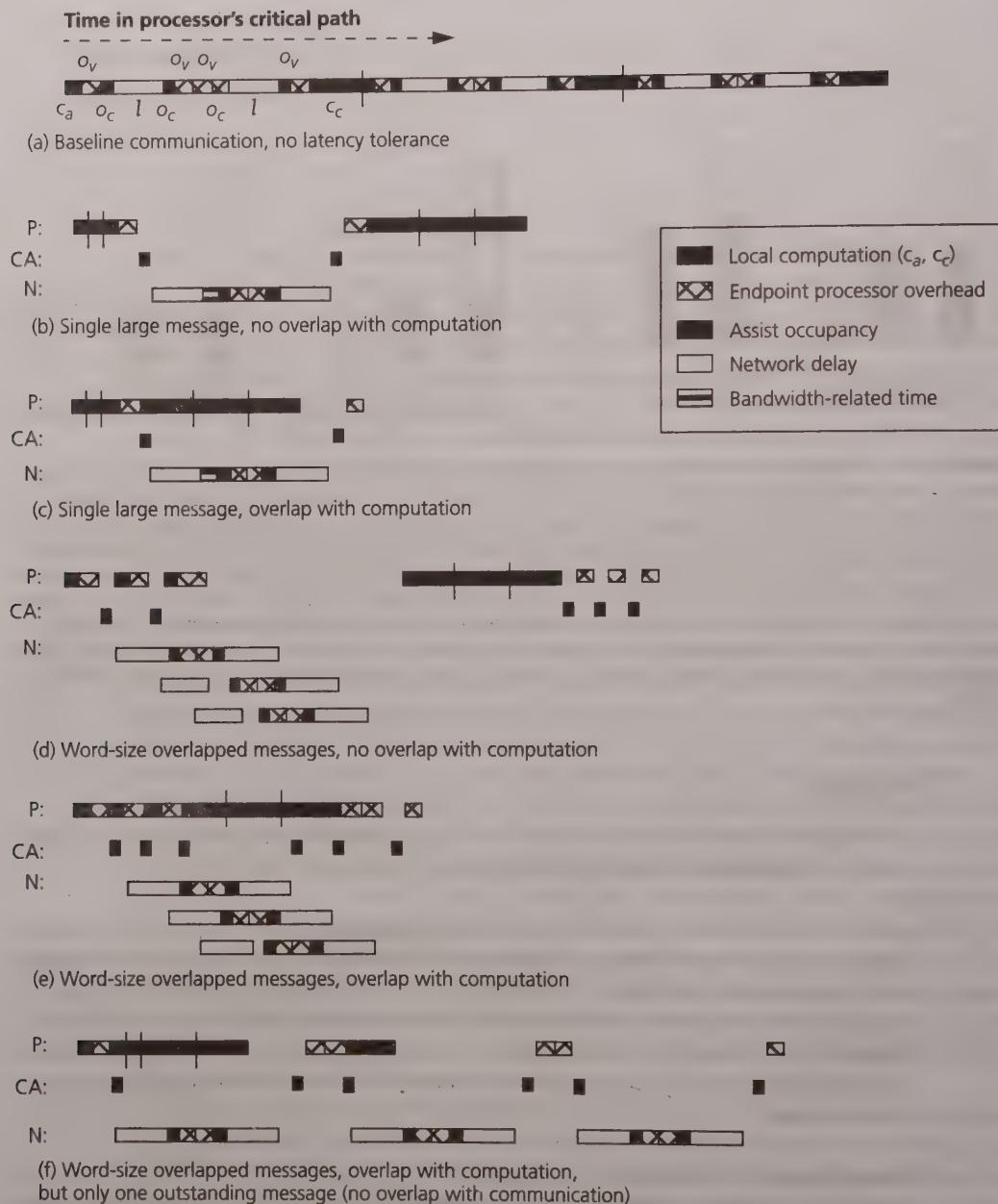


FIGURE 11.7 Timelines for different forms of latency tolerance. P indicates time spent on the main processor, CA on the local communication assist, and N nonlocal (in the network or on other nodes). The timelines are shown from the perspective of process P_A in the message-passing example in Figure 11.1.

may or may not be overlapped with network transmission for that message, and the network pipeline itself can be kept busy only if the message is large. With multiple messages outstanding, processor time, assist occupancy, and network transmission can all be overlapped. The network pipeline itself can be kept well utilized even when messages are very small since several may be in progress simultaneously. Thus, for messages of a given size, more latency can be tolerated if each node allows multiple messages to be outstanding at a time. If L is the latency per message we can possibly hide, and r cycles of independent computation are available to be overlapped with each message, we need $\lceil L/r \rceil$ messages to be outstanding for maximal latency hiding. More outstanding messages do not help since whatever latency could be hidden already has been. Similarly, the number of words outstanding is important. If k words can be outstanding from a processor at a time, from one or multiple messages, then in the best case the network delay as seen by the processor can be reduced by almost a factor of k . More precisely, for one-way messages to the same destination, it is reduced from k^*l cycles to $l + k/B$, where l is the network transit time for a bit and B is the bandwidth or rate of the communication pipeline (network and perhaps assist) in words per cycle.

Assuming that enough messages and words are allowed to be outstanding, the performance parameters of the communication architecture can become limitations as well. Let us examine some of these parameters, assuming that the message sizes are predetermined and the per-message latency of communication we want to hide is L cycles. For simplicity, we shall consider one-way data messages without acknowledgments.

- *Overhead.* The message-processing overhead incurred on the processor cannot be hidden.
- *Assist occupancy.* The occupancy of the assist can be hidden by overlapping with computation. Whether it can be overlapped with other communication (in the same message or in different messages) depends on whether the assist is internally pipelined and whether assist processing for a message can be overlapped with transmission of the message data through the network.

An endpoint message processing time (overhead or occupancy) of o cycles per message establishes an upper bound on the frequency with which we can issue messages to the network (at best, every o cycles). If we assume that on average a processor receives as many messages as it sends and that the overhead of sending and receiving is the same, then the time spent in endpoint processing per message sent is $2o$; so the largest number of messages that we can have outstanding from a processor in a period of L cycles is $L/2o$. If $2o$ is greater than the r cycles of computation we can overlap with each message, then we cannot have the L/r messages outstanding that we would need to hide all the latency. The impact of per-message overhead and occupancy limitations is especially severe when messages are small.

- *Point-to-point bandwidth.* Even if overhead is not an issue, for example, if messages are large, the rate of injecting words into the network may be limited by the slowest link in the entire network pipeline from source to destination; that is, the stages of the network itself, the node-to-network interface, or even any

assist occupancy or processor overhead incurred per word. Just as an endpoint overhead of o cycles between messages limits the number of outstanding messages in an L cycle period to L/o (ignoring incoming messages), a pipeline stage time of s cycles per word in the network limits the number of outstanding words to L/s .

- *Network capacity.* The number of messages a processor can have outstanding is also limited by the total bandwidth or data carrying capacity of the network since different processors compete for this finite capacity. If each link is one word wide, a message of M words traveling h hops in the network may require the equivalent of up to $M*h$ links in the network at a given time. If each processor has k such messages outstanding, then each processor may require up to $M*h*k$ links at a given time. However, if there are D links in the network in all, and all processors are transmitting in this way, then each processor on average can only occupy D/p links at a given time. Thus, the number of outstanding messages per processor k is limited by the relation

$$M \times h \times k < D/p, \text{ or } k < \frac{D}{p \times M \times h}$$

Processor Limitations In a cache-coherent shared address space, spatial locality through long cache blocks already allows more than one word of communication to be outstanding at a time. To hide latency beyond this, a processor and its cache subsystem must allow multiple cache misses to be outstanding simultaneously. This is costly, as we shall see, so processor-cache systems have relatively small limits on this number of outstanding misses (and these include local misses that don't cause communication). Explicit communication requires less hardware tracking and has more flexibility in this regard, though the system may limit the number of messages that can be outstanding at a time.

It is clear from the preceding discussion that making the communication architecture efficient (high bandwidth, low endpoint overhead or occupancy) is very important for tolerating latency effectively. We have seen the properties of some real machines in terms of network bandwidth, node-to-network bandwidth, and endpoint overhead, as well as where the endpoint overhead is incurred, in the last few chapters. From the data for overhead, communication latency, and gap between messages, we can compute two useful numbers in understanding the potential for latency hiding. The first is the ratio L/o for the limit imposed by communication performance on the number of messages that can be outstanding at a time in a period of L cycles, assuming that other processors are not sending messages to this one at the same time. The second is the inverse of the gap, which is the rate at which messages can be pipelined into the network and is also influenced by o . If there is not enough computation to overlap with the latency of L/o overlapped messages, then the only way to hide the remaining latency is to make messages larger. The values of L/o for remote reads in a shared address space and for explicit message passing using message-passing libraries can be computed from Figures 7.31 and 7.32 and from the microbenchmark data for the Origin2000 in Chapter 8, as shown in Table 11.1.

Table 11.1 Number of One-Word Microbenchmark Messages Outstanding at a Time as Limited by L/o Ratio and Rate at Which Messages Can Be Issued to the Network

Machine	One-Way Network Transaction		Remote Read		Machine	One-Way Network Transaction		Remote Read	
	$L/2o$	Msgs/ms	L/o	Msgs/ms		$L/2o$	Msgs/ms	L/o	Msgs/ms
TMC CM-5	2.12	250	1.75	161	NOW-Ultra	1.79	172	1.77	127
Intel Paragon	2.72	131	5.63	133	CRAY T3D	1.00	345	1.13	2,500
Meiko CS-2	3.28	74	8.35	63.3	SGI Origin				5,000

Here is the total latency of a message, including overhead. Two types of operations are considered: a one-word, round-trip remote read in a shared address space, and a one-way network transaction including both send and receive operations. For a remote read, o is the overhead on the initiating processor, which cannot be hidden. For one-way message-passing network transactions, we assume symmetry in messages and therefore count $2o$ to be the sum of the send and receive overhead. For machines with full hardware support for a shared address space, the gap and hence rate of message injection is limited not by processor overhead but by assist occupancy, which is small. The main limitation in these cases is the number of outstanding misses allowed by the processor or assist, but these are ignored in the table.

For the message-passing systems (CM-5, Paragon, CS-2, NOW-Ultra), it is clear that the number of messages that can be outstanding at a time is quite small and that hiding latency with small messages is clearly limited by the endpoint processing overheads. Making messages larger is therefore likely to be more successful at hiding latency than trying to overlap multiple small messages. The hardware-supported shared address space machines (T3D and Origin) are limited much less by performance parameters of the communication architecture in their ability to hide the latency of small messages. Here, the major limitations tend to be the number of outstanding requests supported by the processor or cache system and the fact that a given memory operation may generate several protocol transactions (messages), which stress assist occupancy.

With an understanding of the basics, we are now ready to look at individual techniques in the context of the two major communication abstractions. Since the treatment of latency tolerance for explicit message-passing communication is simpler, the next section discusses all four general techniques in its context.

11.2

LATENCY TOLERANCE IN EXPLICIT MESSAGE PASSING

To understand which latency tolerance techniques are most effective and how they might be employed, it is useful to consider the structure of communication in an abstraction; in particular, how sender- and receiver-initiated communication is orchestrated and whether messages are of fixed or variable size.

11.2.1 Structure of Communication

The actual transfer of data in explicit message passing is typically sender initiated; a receive operation does not in itself cause data to be communicated across the network but rather copies data from an incoming buffer into the application address space. Receiver-initiated communication is performed by sending a request message to the process that is the source of the data, which, in turn, sends the data back.⁴ The baseline communication structure in the example in Figure 11.1 uses synchronous sends and receives, with sender-initiated communication. A synchronous send operation has a communication latency equal to the time it takes to communicate all the data in the message to the destination, plus the time for receive processing, plus the time to return an acknowledgment. The latency of a synchronous receive operation is its processing overhead, including copying the data into the application, plus the additional wait time if the data has not yet arrived. We would like to hide these latencies at both ends. Let us continue to assume that the overhead at each end is incurred on a communication assist and not on the main processor, so that it can be hidden, and see how the four classes of latency tolerance techniques might be applied in classical sender-initiated message passing.

11.2.2 Block Data Transfer

Making messages large is important for amortizing overhead and for ensuring that the rate of the communication pipeline is not limited by the endpoint overhead of message processing. These benefits can be obtained even with synchronous messages. However, if we also want to overlap the communication with computation or with other messages, then we must use one or more of the other three approaches to latency tolerance. Although they can be used with either small or large (block transfer) messages and are in this sense complementary to block transfer, we shall illustrate them with the small messages that are in our baseline communication structure.

11.2.3 Precommunication

Figure 11.8 shows a precommunicating version of the message-passing program introduced in Figure 11.1. The loop on the sender, P_A , is split in two. All the sends are pulled up before the computations of the function $f(B[])$, which are postponed to a separate loop. The sends are made asynchronous, so the process does not stall waiting for them to complete before proceeding.

Is this a good idea? The advantage is that messages are sent to the receiver as early as possible; the disadvantages are that less work is overlapped on the sender between messages, and the pressure on the buffers at the receiver is higher since

4. In other abstractions built on message-passing machines and discussed in Chapter 7, like remote procedure call or active messages, the receiver sends a request message, and a handler that processes the request at the data source sends the data back without involving the application process there. However, we shall focus on classical send/receive message passing, which dominates at the programming model layer.

P_A	P_B
<pre> for i←0 to n-1 do compute A[i]; write A[i]; a_send (A[i] to proc.P_B); end for for i←0 to n-1 do compute f(C[i]); </pre>	<pre> a_receive (myA[0] from P_A); for i←0 to n-2 do a_receive (myA[i+1] from P_A); while (!recv_probe(myA[i])) {}; use myA[i]; compute g(B[i]); end for while (!received(myA[n-1])) {}; use myA[n-1]; compute g(B[n-1]) </pre>

FIGURE 11.8 Hiding latency through precommunication in the message-passing example. In the pseudocode, `a_send` and `a_receive` are asynchronous send and receive operations, respectively.

messages may arrive well before they are needed. Whether the net result is beneficial depends on the overhead incurred per message, the number of messages that a process is allowed to have outstanding at a time, and how much later the receives would have occurred than the sends anyway. For example, if only one or two messages may be outstanding, then we are better off interspersing computation (of $f(B[])$) between asynchronous sends, thus building a *software pipeline* of communication and computation instead of waiting for those two messages to complete before doing anything else. The same is true if the overhead incurred on the assist is high so that we can keep the processor busy while the assist is incurring this high per-message overhead. The ideal case would be to build a balanced software pipeline in which we pull sends up but do just the right amount of computation between message sends.

Now consider the receiver P_B in Figure 11.8. To hide the latency of the receives through precommunication, we try to pull them up earlier in the code and we use asynchronous receives. The `a_receive` call simply posts the specification of the receive to the message layer and allows the processor to proceed. When the data comes in, the assist is notified and it moves the data to the application data structures, transparently to the processor. The application must check that the data has arrived (using the `recv_probe` call) before it can use the data reliably. If the data has already arrived when the `a_receive` is posted (preissued), then what we can hope to hide by preissuing is the cost of the receive processing incurred on the assist; otherwise, we might hide both transit time and receive processing.

Receive overhead is usually larger than send overhead, as discussed in Chapter 7, so if many asynchronous receives are to be issued (or preissued), it is usually beneficial to intersperse computation between them rather than issue them back to back. Otherwise, the assist cannot process them as fast as the processor issues them, so the processor will stall when the buffer between it and the assist fills. One way to do this

is to build a software pipeline of communication and computation, as shown in Figure 11.8. Each iteration of the loop issues an `a_receive` for the data needed for the next iteration and then does the processing for the current iteration. Hopefully, by the time the next iteration is reached, the message for which the `a_receive` was posted in the current iteration will have arrived and will have been processed, and the data will be ready to use. If the receive overhead is much higher than the computation per iteration, the `a_receive` can be issued several iterations ahead instead of just one iteration ahead.

The software pipeline has three parts. In addition to the *steady-state* loop just described, some work must be done to start the pipeline and some to wind it down. In this example, the prologue must post a receive for the data needed in the first iteration of the steady-state loop, and the epilogue must process the code for the last iteration of the original loop (this is left out of the steady-state loop since we do not want to post an asynchronous receive for a nonexistent next iteration). A similar software pipeline strategy could be used if communication were truly receiver initiated, for example, if an `a_receive` or `get` operation sent a request to the node running P_A , and a handler there reacted to the request by supplying the data without needing an explicit send operation in the program. The precommunication strategy is known as *prefetching* the data since the receive (or get) truly causes the data to be fetched across the network. We see prefetching in more detail in the context of a shared address space in Section 11.6 since there the communication is frequently truly receiver initiated.

11.2.4 Proceeding Past Communication in the Same Thread

Now suppose we do not want to pull communication operations up in the code but leave them where they are. One way to hide latency in this case is to simply make the communication messages asynchronous and proceed past them to either computation or other asynchronous communication messages in the same thread. We can continue doing this until we come to a point where we depend on a communication message completing or run into a limit on the number of messages we may have outstanding at a time. Of course, as with prefetching, the use of asynchronous receives means that we must add probes or synchronization to ensure that the data is available before we try to use it (and, on the sender side, that the source data has been copied or sent out before we reuse the corresponding storage).

11.2.5 Multithreading

To exploit multithreading, when a process issues a send or receive operation, it may suspend itself and allow another ready-to-run process or thread from the application to run. If this thread issues a send or receive, then it too is suspended and another thread is switched in. The hope is that by the time we run out of threads and the first thread is rescheduled, the communication operation that thread issued will have completed. The switching and management of threads can be managed by the message-passing library, rather than the application, using calls to the operating sys-

tem to change the program counter and perform other protected thread management functions. For example, the implementation of the send primitive might automatically cause a thread switch after initiating the send (of course, if there is no other ready thread on that processing node, then the same thread will be switched back in). Multithreading allows latency tolerance even with a synchronous message-passing programming model. This approach was the basis of the Occam language on the Transputer.

Switching a thread requires that we save the processor state needed to restart it, including the processor registers, the program counter, the stack pointer, and various processor status words. The state must be restored when the thread is switched back in. Saving and restoring state in software is expensive and can undermine the benefits obtained from multithreading. Some message-passing architectures have therefore provided hardware support for multithreading, for example, by providing multiple sets of registers and program counters in hardware. Note that systems that support asynchronous message handlers that are separate from the application process but that run on the main processor, are essentially multithreaded between the application process and the handler, even if applications themselves do not use multithreading. This form of multithreading has also been supported in hardware by some research architectures (e.g., the message-driven processor of the J-machine [Dally et al. 1992; Noakes, Wallach, and Dally 1993]). We discuss these hardware support issues in more detail in Section 11.7 where we examine multithreading in a shared address space.

11.3

LATENCY TOLERANCE IN A SHARED ADDRESS SPACE

The rest of this chapter focuses on latency tolerance in the context of a hardware-supported shared address space. This discussion is more detailed than the message-passing discussion for several reasons. First, the existing hardware support for communication brings the techniques and requirements much closer to the architecture and the hardware/software interface. Second, the implicit nature of long-latency events (such as communication) makes it more likely that much of the latency tolerance will be addressed by the system rather than by the user program. Third, the granularity of communication and efficiency of the underlying communication mechanisms require that the latency tolerance techniques be hardware supported to be effective. And finally, since much of the latency is that of reads, writes, and perhaps instructions—not explicit communication operations like sends and receives—most of the techniques are applicable to uniprocessors as well. In fact, since we don't know ahead of time which read and write operations will generate communication, latency hiding is treated in much the same way for local accesses as for communication in shared address space multiprocessors. The difference is in the magnitude of the latencies and, in cache-coherent systems, in the interactions with the cache coherence protocol.

Much of our discussion of latency tolerance in a shared address space applies to cache-coherent systems as well as those that do not cache shared data. For the most part, we assume that the shared address space (and cache coherence) is supported in

hardware and that the default communication and coherence are at the granularity of individual words or cache blocks. The experimental results presented in the following sections are taken from the literature. These results use several of the same applications that we have used in previous chapters, but they typically use different versions with somewhat different communication-to-computation ratios and other behavioral characteristics, and they do not always follow the methodology outlined in Chapter 4. The system parameters used also vary across studies, so the results cannot be compared across techniques and are presented purely for illustrative purposes. As with message passing, let us begin by briefly examining the structure of communication in this abstraction.

11.3.1 Structure of Communication

The baseline communication in a shared address space is through reads and writes and is called *read-write communication* for convenience. Receiver-initiated communication is typically performed with memory operations that result in data from another processor's memory or cache being accessed. It is thus a natural extension of data access in the uniprocessor programming model: accessing words of memory when you need to use them.

If there is no caching of shared data, sender-initiated communication may be performed through writes to data that are allocated in remote memories.⁵ With cache coherence, the effect of writes is more complex. Whether writes lead to sender- or receiver-initiated communication depends on the cache coherence protocol. For example, suppose processor P_A writes a word that is allocated in P_B 's memory. In the most common case of an invalidation-based cache coherence protocol with write-back caches, the write will only generate a read-exclusive or upgrade request and perhaps some invalidations, and it may bring data to itself. It will not actually cause the newly written data to be transferred to P_B . While requests and invalidations involve network transactions too, and it is important to hide their latency, the actual communication of the new value from P_A to P_B will be generated by the later read or write of the data by P_B . In this sense, it is receiver initiated. Alternatively, the data transfer may be caused by an asynchronous replacement of the data from P_A 's cache, which will cause it to go back to its home in P_B 's memory. In an update protocol, on the other hand, the write itself will communicate the data from P_A to P_B if P_B had a cached copy.

Whether receiver initiated or sender initiated, the communication in a hardware-supported read-write shared address space is naturally fine grained, which makes latency tolerance particularly important. The different approaches to latency tolerance are better suited to different types and magnitudes of latency and have achieved different levels of acceptance in commercial products. We examine these approaches in some detail in the next sections.

5. An interesting case is the one in which a processor writes a word that is allocated in a different processor's memory and a third processor reads the word. In this case, we have two data communication events to transfer data from producer to consumer—one “sender initiated” and one “receiver initiated.”

P_A	P_B
<pre> for i←0 to n-1 do A[i]←... end for put (A[0..n-1] to tmp[0..n-1]); flag ← 1; for i←0 to n-1 do compute f(C[i]); end for </pre>	<pre> while (!flag) {} /*spin-wait*/ for i←1 to n-1 do use tmp[i]; compute g(B[i]); end for </pre>

FIGURE 11.9 Using block transfer in a shared address space for the example of Figure 11.1. The array A is allocated in processor P_A 's local memory and the array tmp in processor P_B 's local memory.

11.4 BLOCK DATA TRANSFER IN A SHARED ADDRESS SPACE

In a shared address space, coalescing data to make messages larger (called block data transfer) and initiating the block transfers can be done either explicitly in the user program or transparently by the system. For example, the prevalent use of long cache blocks on modern machines is a means of transparent block transfers in cache-block-sized messages. Relaxed memory consistency models further allow us to buffer words or cache blocks and send them in coalesced messages only at synchronization points, a fact utilized particularly by software shared address space systems, noted in Chapter 9. However, let us focus here on explicit initiation of block transfers.

11.4.1 Techniques and Mechanisms

Explicit block transfers are initiated by explicitly issuing a put command, similar to a send but with both source and destination addresses specified by the sender, in the user program, as shown in the simple example in Figure 11.9. The put command is interpreted by the communication assist, which transfers the data in a pipelined manner from the source node to a destination node. At the destination, the communication assist transfers data from the network and into the specified locations. The path is shown in Figure 11.10. The major differences with send/receive message passing arise from the ability of the sending process to directly specify the program data structures (virtual addresses) where the data is to be placed at the destination, since these locations are in the shared address space. Receive operations are not needed in the programming model since the incoming message specifies where the data should be put in the program address space. System buffering or copying is also

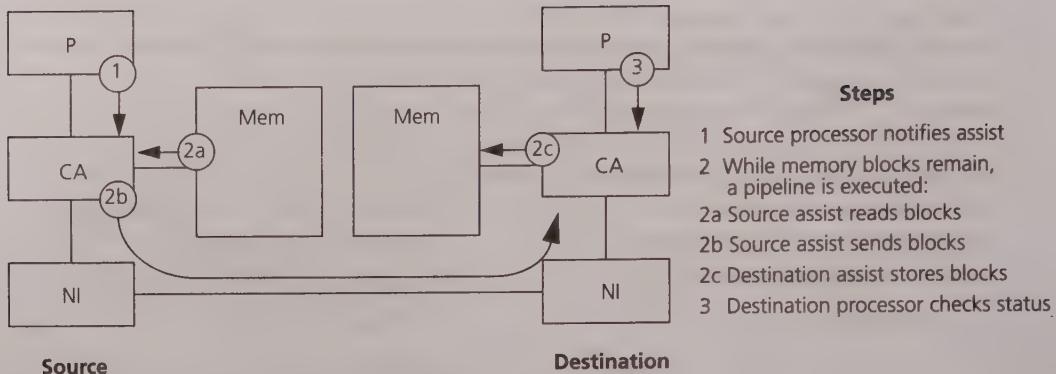


FIGURE 11.10 Path of a block transfer from a source node to a destination node in a shared address space machine. The transfer is done in terms of cache blocks since that is the granularity of communication for which the machine is optimized.

not needed in main memory at the destination, if the destination assist is available to put the data directly from the network interface into the user data structures in memory. However, some form of synchronization (like spinning on a flag or blocking) must be used to determine that the data has arrived before it is used by the destination process as well as to ensure that the destination region is ready to be overwritten before the data arrives. It is also possible to use receiver-initiated block transfer, in which case the request to transfer the data is issued by the receiver and handled by the source communication assist.

The communication assist on a node that performs the block transfer could be the same one that processes coherence protocol transactions or a separate DMA engine dedicated to block transfer. It can be designed with varying degrees of aggressiveness in its functionality; for example, it may allow contiguous data transfers only, uniform strides, or more general scatter-gather operations. The block transfer may leverage the support provided for efficient transfer of cache blocks, or it may be a completely separate mechanism. Since block transfers may need to interact with the coherence protocol in a coherent shared address space, we shall assume that the block transfer is built on top of pipelined transfers of entire cache blocks; that is, the block transfer engine is a part of the source assist that reads cache blocks out of memory and transfers them into the network in a pipelined manner. Figure 11.10 shows the steps in a possible implementation of block transfer in a cache-coherent shared address space.

11.4.2 Policy Issues and Trade-Offs

Two policy issues are of particular interest in block transfer: how interactions with the underlying shared address space and cache coherence protocol are handled, and where the block-transferred data is placed in the destination node.

Interactions with the Cache-Coherent Shared Address Space

The first interesting interaction is with the shared address space itself, regardless of whether it includes automatic coherent caching or not. The data that a processor “puts” in a block transfer may not be allocated in its local memory but in another processing node’s memory or may even be scattered among other node’s memories. The options here are to disallow such transfers, have the initiating node retrieve the data from the other memories and forward it in a pipelined manner, or have the initiating node send messages to the owning nodes’ assists asking them to perform the relevant transfers.

The second interaction is specific to a cache-coherent shared address space since now the same data structures may be communicated using two different protocols: block transfer and cache coherence. Regardless of which main memory the data is allocated in, it may be cached in the sender’s cache in dirty state, the receiver’s cache in shared state, or another processor’s cache (including the receiver’s) in dirty state. The first two cases create what is called the *local coherence* problem; that is, ensuring that the data sent constitutes the most up-to-date values on the sending node and that copies of that data on the receiving node are left in coherent state after the transfer. The third case is called the *global coherence* problem; that is, ensuring that the values transferred are the latest values for those addresses anywhere in the system (according to the consistency model) and that the data involved in the transfer is left in coherent state in the whole system. Once again, the options are to provide no guarantees in any of these three cases, to provide only local but not global coherence, or to provide full global coherence. Each successive case makes the programmer’s job easier but imposes more requirements on the communication assist. To provide coherence, the assist must check the state of every block being transferred, retrieve data from the appropriate caches, and invalidate data in the appropriate caches. The data being transferred may not be properly aligned with cache blocks, which makes interactions with the block-level coherence protocol even more complicated, as does the possibility that the directory information for the blocks being transferred may not be present at the sending node. The explicit message-passing programming model is simpler in this regard: it requires local coherence but not global coherence since any data that a process can access—and hence transfer—is allocated in its private address space and cannot be cached by any other processors.

For block transfer to maintain even local coherence, the assist has some work to do for every cache block and becomes an integral part of the transfer pipeline. It can therefore easily become the bottleneck in transfer bandwidth, affecting even those blocks for which no interaction with caches is required. Global coherence may require the assist to send network transactions around before sending each block, reducing bandwidth dramatically. It is therefore important that a block transfer system have a “pay for use” property; that is, providing coherent transfers should not significantly hurt the performance of transfers that do not need coherence, particularly if the latter are a common case. For example, if block transfer is used in the

system only to support explicit message-passing programs, and not to accelerate coarse-grained communication in an otherwise read-write shared address space program, then it may make sense to provide only local coherence but not global coherence. However, as shown in Example 11.1, global coherence is essential if we want to provide true integration of block transfer with a coherent read-write shared address space and not make the programming model restrictive.

EXAMPLE 11.1 Give a simple example of a situation where global coherence may be needed for block transfer in a cache-coherent shared address space.

Answer Consider false sharing. The data that a processor P_1 wants to send P_2 may be allocated in P_1 's local memory and produced by P_1 , but another processor P_3 may have written other words on those cache blocks more recently. The cache blocks will be in invalid state in P_1 's cache, and the latest copies must be retrieved from P_3 in order to be sent to P_2 . False sharing is only an example: it is not difficult to see that in a true shared address space program the words that P_1 sends to P_2 might themselves have been written most recently by another processor. ■

More details on implementing block transfer in cache-coherent machines can be found in the literature (Kubiatowicz and Agarwal 1993; Heinlein et al. 1994; Woo, Singh, and Hennessy 1994; Heinlein et al. 1997).

Where to Place the Transferred Data

The other interesting policy issue is whether the data that is block transferred should be placed in the main memory of the destination, in the cache, or in both. Since the destination processor will read the data, having the data come into its cache is useful. However, it has some disadvantages as well. First, it requires intervening in the processor cache, which is expensive on modern systems and may prevent the processor from accessing the cache at the same time. Second, unless the block-transferred data is used soon it may be replaced before it is used, so we should transfer data into main memory as well to keep the resulting cache misses local. And third and most dangerous, the transferred data might replace other useful data from the cache, perhaps even data that is currently in the processor's active working set. For these reasons, large block transfers into a small first-level cache are not likely to be a good idea, whereas transfers into a large or second-level cache may be more useful.

11.4.3 Performance Benefits

Using large explicit transfers in a shared address space has several advantages compared to communicating implicitly in cache-block-sized messages. However, some of the advantages discussed earlier for message passing are compromised and there are disadvantages as well. Let us discuss the trade-offs qualitatively and then look at some performance data.

Potential Advantages and Disadvantages

The following are the major performance advantages of using block transfer. (The first two items were discussed in the context of message passing, so we only point out the differences here.)

- *Amortized per-message overhead.* This advantage may be less important in a hardware-supported shared address space since the endpoint overhead for cache block communication is already quite small. In fact, explicit, flexibly sized transfers tend to have higher endpoint overhead than small, fixed-size transfers, since buffering for the latter can easily be done in hardware while the former may require software management and copying. As a result, the per-communication overhead is likely to be substantially larger for block transfer than for read-write communication. Block transfer engines on many systems are like DMA devices, operating on physical addresses, so they run in kernel mode and require a system call, greatly increasing overhead. This increase turns out to be a major stumbling block for several commercial hardware-coherent machines, which currently use the facility more for operating system operations like page migration than for application activity. However, in less efficient communication architectures, such as those that use commodity parts, the overhead per message can be large even for cache block communication and the amortization due to block transfer can be very important.
- *Pipelined transfer of large chunks of data.*
- *Less wasted bandwidth.* In general, the larger the message, the less the relative overhead of headers and routing information compared to the payload (the data transferred that is useful to the end application). This advantage may be lost when the block transfer is built on top of the existing cache line transfer mechanism, in which the header information is sent once per cache block anyway. When used properly, block transfer can reduce the number of protocol messages (e.g., invalidations, acknowledgments) as well.
- *Replication of transferred data in the destination main memory.* Since block transfer is usually done into main memory, subsequent capacity misses at the destination node will be satisfied locally. This reduces the number of capacity misses at the destination that have to be satisfied remotely, as in COMA machines. However, without a COMA architecture, it implies that the user must manage the coherence and replacement of the replicated data in main memory.
- *Bundling of synchronization with data transfer.* Synchronization notifications can be piggybacked on the same message that transfers data rather than having to communicate separately for data and synchronization. This reduces the number of messages needed, though the absence of an explicit blocking receive operation implies that, functionally, synchronization still has to be managed separately from data transfer at the endpoints, just as in asynchronous message passing.

The potential performance disadvantages of using block transfer are

- *Higher overhead per transfer.*
- *Increased contention.* Long messages tend to incur higher contention, both at the endpoints and in the network, because they occupy more resources in the network at a time: the latency tolerance they provide places greater bandwidth demands on the communication architecture. On the other hand, bandwidth demand due to protocol messages is reduced compared to a cache coherence protocol, as discussed earlier.
- *Extra work.* Programs may have to do extra work to organize themselves so communication can be performed in large transfers (if this can be done effectively at all). This extra work may turn out to have a higher cost than the benefits achieved from block transfer (see the discussion of the Barnes-Hut application in Section 3.6).

Example 11.2 illustrates the performance improvements that can be obtained by using block transfers rather than reads and writes, particularly from amortized overhead and pipelined data transfer.

EXAMPLE 11.2 Suppose we want to communicate 4 KB of data from a source node to a destination node in a cache-coherent shared address space machine with a cache block size of 64 bytes. Assume that the data is contiguous in memory so that spatial locality is exploited perfectly (Exercise 11.6 discusses the issues that arise when spatial locality is not good). Suppose it takes the source assist 40 processor cycles to read a cache block out of memory, 50 cycles to push a cache block out through the network interface, and the same numbers of cycles for the complementary operations at the receiver. Assume that the local read-miss latency is 60 cycles, the remote read-miss latency is 180 cycles, and the time to start up a block transfer is 200 cycles. What is the potential performance advantage of using block transfer rather than communication through cache misses, assuming that the processor blocks on memory operations until they complete?

Answer The cost of getting the data through read misses, as seen by the processor, is $180 * (4,096 / 64) = 11,520$ cycles. With block transfer, the rate of the transfer pipeline is limited by $\max(40, 50, 50, 40)$ or 50 cycles per block. This brings the data to the local memory at the destination, from where it will be read by the processor through local misses, each of which costs 60 cycles. Thus, the cost of getting the data to the destination processor with block transfer is $200 + (4,096 / 64) * (50 + 60)$ or 7,240 cycles. Using block transfer therefore gives us a speedup of $11,520 / 7,240$, or 1.6. ■

Another way to achieve block transfers is with vector operations, for example, a vector read from remote memory. In this case, a single instruction causes the data to appear in the (vector) registers; individual load and store instructions are not required even locally, and a savings in instruction bandwidth and perhaps local cache misses results. Vector registers are typically managed by software, which has the disadvantages of aliasing and tying up register names but the advantage of not suffering from cache conflicts. However, many high-performance systems today do not include vector operations, so we shall focus on block transfers that still need individual local read and write operations to access and use the data.

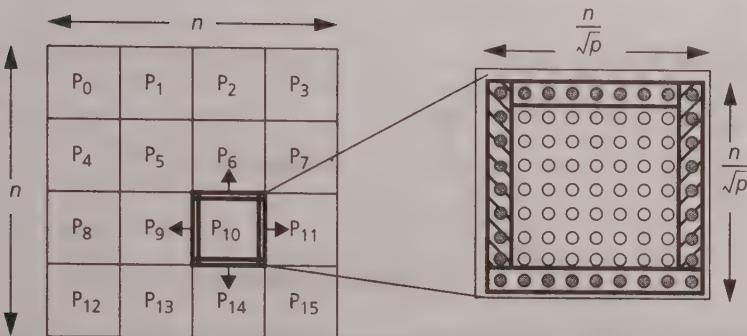


FIGURE 11.11 The use of block transfer in a near-neighbor equation solver. A process can send an entire boundary subrow or subcolumn of its partition to the process that owns the neighboring partition in a single message. The size of each message is n/\sqrt{p} elements.

Performance Benefits and Limitations in Real Programs

Whether block transfer can be used effectively in real programs depends on both program and system characteristics. Consider a simple example amenable to block transfer, a near-neighbor equation solver on a grid, to see how the performance issues play out. Let us ignore the coherence complications discussed previously by assuming that the transfers are done from the source main memory to the destination main memory and that the data is not cached anywhere. We assume a four-dimensional array representation of the two-dimensional grid and a processor's partition of the grid allocated in its local memory.

Instead of communicating elements at partition boundaries through individual cache blocks, a process can simply send a single message containing all the appropriate boundary elements to its neighbor, as shown in Figure 11.11. The communication-to-computation ratio is proportional to \sqrt{p}/n . Since block transfer is intended to improve communication performance, this ratio in itself would tend to increase the relative effectiveness of block transfer as the number of processors increases. However, the size of an individual block transfer is n/\sqrt{p} elements and therefore decreases with p . Smaller transfers make the additional overhead of initiating a block transfer relatively more significant. The trade-offs between these two factors combine to yield a sweet spot in the number of processors for which block transfer is most effective for a given grid size, as suggested by Figure 11.12. The sweet spot moves to larger numbers of processors as the grid size increases.

Figure 11.13 illustrates this sweet spot effect for a Fast Fourier Transform (FFT) on a simulated architecture that models the Stanford FLASH multiprocessor and is quite close to the SGI Origin2000 (though much more aggressive in its block transfer capabilities and performance). We can see that the relative benefit of block transfer over ordinary cache-coherent communication diminishes with increasing cache block size since the excellent spatial locality in this program causes long cache

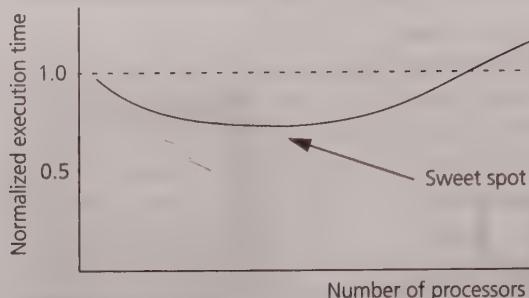


FIGURE 11.12 Relative performance improvement with block transfer. The figure illustrates the execution time using block transfer normalized to that using loads and stores. The sweet spot occurs when communication is high enough that block transfer matters, and the transfers are also large enough that overhead doesn't dominate.

blocks to themselves behave like small block transfers. For the largest cache block size of 128 bytes (the actual block size of the SGI Origin2000), the benefits of using block transfer are small, even with a very efficient block transfer engine. Figure 11.14 shows results for Ocean, a more complete, regular application whose communication is a variant of the nearest-neighbor communication in Figure 11.11. Block transfers at row-oriented partition boundaries transfer contiguous data, but this communication also has very good spatial locality; at column-oriented partition boundaries, spatial locality in communicated data is poor, but it is also difficult to exploit block transfer well. When block transfer is implemented using pipelined transfers of whole cache blocks (as assumed here), each word at a column boundary will be transferred on a separate cache block unless the boundary column is first copied to a contiguous data structure, so the lack of spatial locality hurts block transfer as well. Overall, the communication-to-computation ratio in Ocean is much smaller than in FFT. Although the ratio increases at higher levels of the grid hierarchy in the multigrid solver—as processor partitions become smaller—block transfers also become smaller at these levels and are less effective at amortizing overhead. The result is that block transfer is a lot less helpful for this application, and the relative benefits of block transfer do not depend so greatly on cache block size. Quantitative data for other applications can be found in (Woo, Singh, and Hennessy 1994).

Block transfer may be useful for some aspects of parallelism management. For example, in a task-stealing scenario, if the task descriptors for a stolen task are large, they can be sent using a block transfer, as can the data associated with the task, and the necessary synchronization for task queues can be piggybacked on these transfers as well.

One situation in which block transfer is clearly beneficial is when the overhead to initiate remote read-write accesses is very high; for example, when the shared address space is implemented in software. To see the effects of other parameters of the communication architecture, such as network delay and point-to-point bandwidth, let us look at the benefits of block transfer more analytically. Let us continue

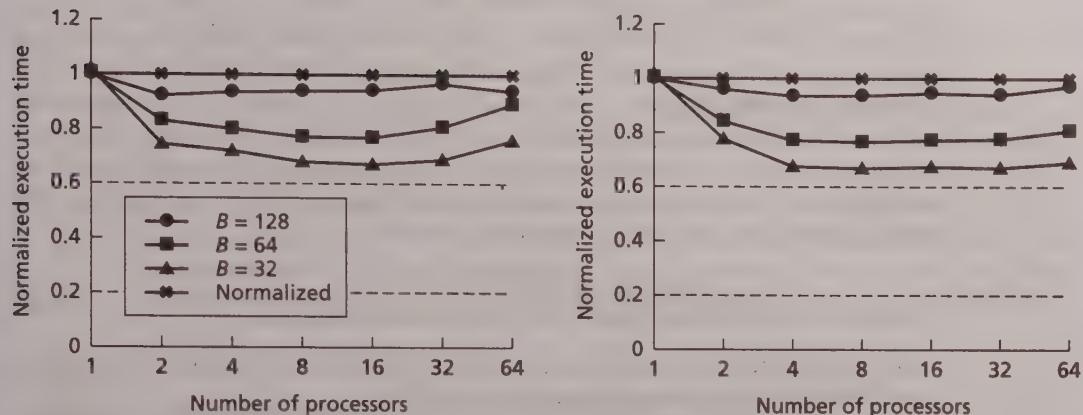


FIGURE 11.13 The benefits of block data transfer in a Fast Fourier Transform program. The two graphs are for two problem sizes. The architectural model and parameters used resemble those of the Stanford FLASH multiprocessor (Kuskin et al. 1994) and are similar to those of the SGI Origin2000. Each curve is for a different cache block size (B) in the second-level cache. For each curve, the y-axis shows the execution time normalized to that of an execution without block transfer using the same cache block size. Thus, the different curves are normalized differently (self-normalized) and different points for the same x-axis value are not normalized to the same number. The greater the y-axis value of a point, the less the improvement obtained by using block transfer over regular read-write communication for that case.

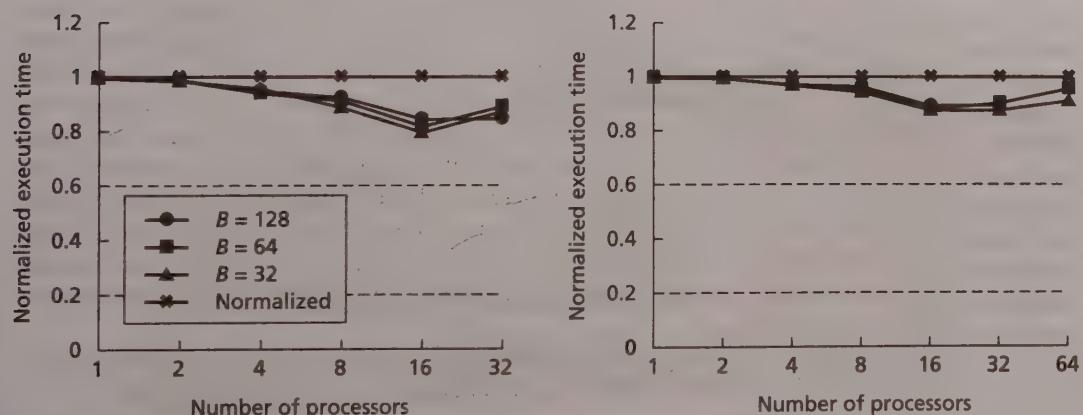


FIGURE 11.14 The benefits of block transfer in the Ocean application. The platform and data interpretation are exactly the same as in Figure 11.13. B is the second-level cache block size of the machine. The benefits and their dependence on cache block size are not as great as in the Fast Fourier Transform program.

to assume that read misses stall the processor and do not have their latency hidden. In general, the time to communicate a large block of data through remote read misses is ($\# \text{ of read misses} * \text{remote read-miss time}$), and the time to get the data to the destination processor through block transfer is ($\text{start-up overhead} + \# \text{ of cache blocks transferred} * \text{pipeline stage time per cache block} + \# \text{ of read misses} * \text{local read-miss time}$). The simplest case to analyze is a very large contiguous chunk of data for which we can ignore the start-up cost and assume perfect spatial locality. In this case, the speedup due to block transfer is limited by the ratio

$$\frac{\text{Remote Read-Miss Time}}{\text{Block Transfer Pipeline Stage Time} + \text{Local Read-Miss Time}} \quad (11.1)$$

where the block transfer pipeline stage time is the maximum of the time spent in any stage of the pipeline that was shown in Figure 11.10.

A longer network delay implies that the remote read-miss time is increased. If the point-to-point network bandwidth does not decrease with increased network delay, then the other terms are unaffected and longer delays favor block transfer. Alternatively, network bandwidth may decrease proportionally to the increase in delay, for example, if delay is increased by decreasing the clock speed of the network. In this case, as delay increases, at some point the rate at which data can be pushed into the network becomes smaller than the rate at which the assist can move the data to or from main memory. Up to this point, the memory access time dominates the block transfer pipeline stage time, so increases in network delay do not change block transfer performance and hence favor block transfer. After this point, the numerator and denominator in the ratio increase at the same rate with network delay, so the relative advantage of block transfer is unchanged.

Finally, suppose delay and overhead stay fixed but bandwidth changes (e.g., more wires are added to each network link). We might think that communication based on stalling remote reads is latency bound whereas block transfer is bandwidth bound, so increasing bandwidth should favor block transfer. This is true up to the point where network bandwidth limits the block transfer pipeline stage time. However, increasing network bandwidth past that point means that the memory access time may become the bottleneck, so increasing bandwidth further does not improve the performance of block transfer. Reducing bandwidth has the inverse effect. Thus, if the other variables are kept constant in each case, block transfer is more effective with increased per-message overhead, increased network delay, and increased bandwidth, up to a point.

In summary, the performance improvement obtained from block transfer over read-write cache-coherent communication depends on the following factors:

- the fraction of the execution time spent in communication that is amenable to block transfer
- the extra work that must be done to structure this communication as block transfers
- the problem size and number of processors, which affect the communication-to-computation ratio as well as the sizes of the transfers

- the overheads, delays, and bandwidths in the system
- the spatial locality in the program and how it interacts with the granularity of data transfer

If we can really make all messages large enough, then the delay components of communication may not be the problem; bandwidth becomes a more significant constraint. But if not, we still need to hide the latency of data accesses by overlapping them with computation or with other accesses. The other three latency tolerance approaches do this; however, to be successful at cache block granularity, they require support in the microprocessor as well. Instead of precommunication, let us begin with techniques to move past long-latency accesses in the same thread of computation.

11.5 PROCEEDING PAST LONG-LATENCY EVENTS

A processor can proceed past a memory operation to other instructions if the memory operation is made nonblocking. For writes, this is usually straightforward: the write is put in a write buffer, and the processor goes on while the buffer takes care of issuing the write to the memory system and tracking its completion as necessary. Many processors also support nonblocking reads in which the processor performs instructions while the read is outstanding. Without additional support, the processor stalls when it encounters an instruction that attempt to use the results of the read. The problem is that in most programs such a dependent instruction is likely to follow soon after the read. If the read misses in the cache, very little of its miss latency is likely to be hidden in this manner. Hiding read latency effectively requires that we look ahead in the instruction stream past the dependent instruction to find other instructions that are independent of the read. This requires support in the compiler instruction scheduling, the hardware, or both. Hiding write latency does not usually require instruction lookahead: we can allow the processor to stall when it encounters a dependent instruction since it is not likely to encounter one soon.

Proceeding past operations before they complete can benefit from both buffering and pipelining. Buffering of memory operations allows the processor to proceed with other activity, to delay the propagation and application of requests like invalidations, and to merge operations to the same word or cache block in the buffer. (The page-based, all-software, shared virtual memory protocols discussed in Chapter 9 are an extreme example of buffering, delaying propagation until synchronization points.) Pipelining multiple memory operations into the memory hierarchy allows their latencies to be overlapped. These advantages hold for uniprocessors as well as multiprocessors.

In multiprocessors, proceeding past memory operations before they complete or commit violates the sufficient conditions for sequential consistency stated in Chapter 5. Whether or not it actually violates SC itself depends on whether the operations are allowed to become visible out of program order. By requiring that memory operations from the same thread should not appear to perform out of program order with respect to other processors, SC restricts—but by no means eliminates—the amount

of buffering and overlap that can be exploited. Relaxed consistency models allow greater overlap, including by allowing some types of memory operations to complete out of order. The extent of overlap possible is thus determined by both the machine mechanisms and the consistency model: relaxed models may not be useful if the mechanisms needed to support their reorderings (e.g., write buffers, compiler support, or dynamic scheduling) are not provided, and the success of some types of aggressive implementations for overlap may be restricted by the consistency model.

Our discussion of proceeding past operations is organized around increasingly aggressive mechanisms needed for overlapping the latency of read and write operations. Each of these mechanisms is naturally associated with a particular class of consistency models—the class that allows those operations to complete out of order—but can also be exploited in more limited ways with stricter consistency models by allowing overlap but not out-of-order completion. In each case, we examine the performance gains and the implementation complexity, assuming the most aggressive consistency model needed to fully exploit that overlap, as well as the extent to which sequential consistency can exploit the overlap mechanisms. The focus is on hardware cache-coherent systems, though many of the issues apply quite naturally to systems that don't cache shared data or are software coherent. To simplify the discussion, let us begin by examining mechanisms to hide only write latency since hiding read latency requires more elaborate support (albeit provided in many modern microprocessors). Reads are initially assumed to be blocking, so the processor cannot proceed past them; later, hiding read latency is examined as well.

11.5.1 Proceeding Past Writes

Let us start with a simple, statically scheduled processor with blocking reads. To proceed past write misses, the only support we need in the processor is a write buffer. Writes that miss in the first-level cache are simply placed in the buffer, and the processor proceeds to other work in the same thread. The processor (or the first-level cache) stalls upon a write only if the write buffer is full. The write buffer may also be placed before the first-level cache, in which case all writes are placed in it.

The write buffer is responsible for controlling the visibility of writes to the rest of the extended memory hierarchy, and hence to other processors, and for the completion of writes relative to other operations from that processor. This frees the processor's internal execution unit and the extended memory hierarchy from worrying about these orders. Consider overlap with reads. For correct uniprocessor operation, a read may be allowed to bypass the write buffer and may be issued to the memory system as long as a write to the same location is not pending in the write buffer. If it is, the value from the write buffer may be forwarded to the read even before the write completes, or the write buffer may be flushed and the read issued to the memory system thereafter. Forwarding allows reads to complete out of order with respect to earlier writes in program order, thus violating SC in multiprocessors, while flushing does not. Reads bypassing the write buffer will also violate SC unless the read is not allowed to complete (bind its value) before those writes. Thus, SC can take advantage of write buffers, but the benefits are limited. We know from Chapter 9

that processor consistency (PC) and total store ordering (TSO) are the consistency models that allow only reads to complete before previous writes.

Now consider the overlap among writes themselves. Multiple writes may be placed in the write buffer, which determines the order in which they are made visible or in which they complete. If writes are allowed to complete out of order, a great deal of flexibility and overlap among writes can be exploited by the buffer. First, a write to a location or cache block that is currently in the buffer can be merged (coalesced) into that cache block and only a single ownership request sent out for the block by the buffer, thus reducing traffic as well. Especially if the merging is not into the last entry of the buffer, this leads to writes becoming visible out of program order. Second, buffered writes can be retired from the buffer when they issue to the memory system, making it possible for other writes behind them to get through before they complete. This allows the ownership requests of the writes to be pipelined through the extended memory hierarchy, but it can make the writes visible to other processors out of program order and can violate write atomicity.

Partial store order (PSO) or more relaxed consistency models allow writes to complete out of program order. Stricter models like SC, TSO, and PC essentially restrict merging into the write buffer to the last block in the buffer, and even then in restricted circumstances since other processors must not be able to see the writes to different words in the block in a different order. Retiring writes early to let others pass through is possible under the stricter models only if the order of visibility and completion among these writes is preserved in the extended memory hierarchy. This is relatively easy in a bus-based machine but very difficult in a distributed-memory machine with different home memories and independent network paths. On the latter systems, guaranteeing program order among writes, as needed for SC, TSO, and PC, essentially requires that a write not be retired from the head of the FIFO buffer until it has committed with respect to all processors.

Overall, a strict model like SC can utilize write buffers to overlap write latency with reads and other writes but to a limited extent. Greater latency tolerance requires relaxing the consistency model. Under relaxed models, exactly when write operations are sent out to the extended memory hierarchy and made visible to other processors depends on implementation and performance considerations as well. For example, invalidations may be sent out as soon as the writes are able to get through the write buffer, or they may be delayed in the buffer until the next synchronization point. The latter option allows greater merging of writes as well as reduction of invalidations and misses due to false sharing. However, it also implies that invalidation-related traffic will be bursty at synchronization points rather than pipelined throughout the computation.

Performance Impact

Simulation studies have shown the benefits of allowing the processor to proceed past writes on the parallel applications in the original SPLASH application suite (Singh, Weber, and Gupta 1992), assuming blocking reads but without maintaining SC (Gharachorloo, Gupta, and Hennessy 1991a; Gharachorloo 1995). The resulting

techniques are separated into those that allow the program order between a write and a following read (write → read order) to be violated, satisfying TSO or PC, and those that also allow the order between two writes (write → write order) to be violated, satisfying PSO. The latter is essentially the best that more relaxed models like relaxed memory order (RMO), weak ordering (WO), or release consistency (RC) can accomplish given that reads are blocking.

Figure 11.15 shows with single-issue, statically scheduled processors the reduction in execution time, divided into different components, for two representative parallel programs when the write → read and write → write orders are allowed to be violated. The programs are older versions of the Ocean and Barnes-Hut applications, running with small problem sizes on 16-processor, simulated, directory-based cache-coherent systems that are considerably less aggressive in instruction scheduling than current microprocessors. The baseline for comparison is the straightforward implementation of sequential consistency that satisfies the sufficient conditions: the processor issuing a read or write stalls until that reference completes. (Using the write buffer but preserving SC by stalling a read until the write buffer is empty shows only a very small performance improvement over stalling the processor on all writes themselves.)

The second bars in the figure show that, with a deep write buffer and these system assumptions, simply allowing write → read reordering is usually enough to hide most of the write latency from the processor. It is less successful in Ocean, where there is a greater frequency of write misses. The full study also showed some interesting secondary effects. In some cases, the read stall time increases slightly from the base SC case. This is because the additional bandwidth demands of hiding write latency contend with read misses, making them more costly. This is a relatively small effect with simple processors, though it may be more significant with modern, dynamically scheduled processors that also hide read latency. Another beneficial effect is that synchronization wait time is also sometimes reduced. As memory stall time is reduced, imbalances in memory stall time across processors are reduced, and hence load imbalance is diminished. Also, if the latency write performed inside a critical section is hidden, then the critical section completes more quickly. The lock protecting it can be passed more quickly to another processor, which therefore incurs less wait time for that lock.

The third bars in the figure show the results for the case in which the write → write order is allowed to be violated as well. Write merging is now enabled to any block in the same 16-entry write buffer and a write is allowed to retire from the write buffer as soon as it reaches the head, even before it is committed. Thus, pipelining of writes through the memory and interconnect is given priority over buffering them for more time (which would enable merging and delayed invalidations)⁸⁷⁶. Of course, having multiple writes outstanding in the memory system requires that the caches allow multiple outstanding misses.

The write-write overlap hides whatever write latency remained with write-read overlap, even in Ocean. Since writes retire from the write buffer at a faster rate, the write buffer does not fill up and stall the processor as easily. Synchronization wait time is reduced further in some cases since a sequence of writes before a release

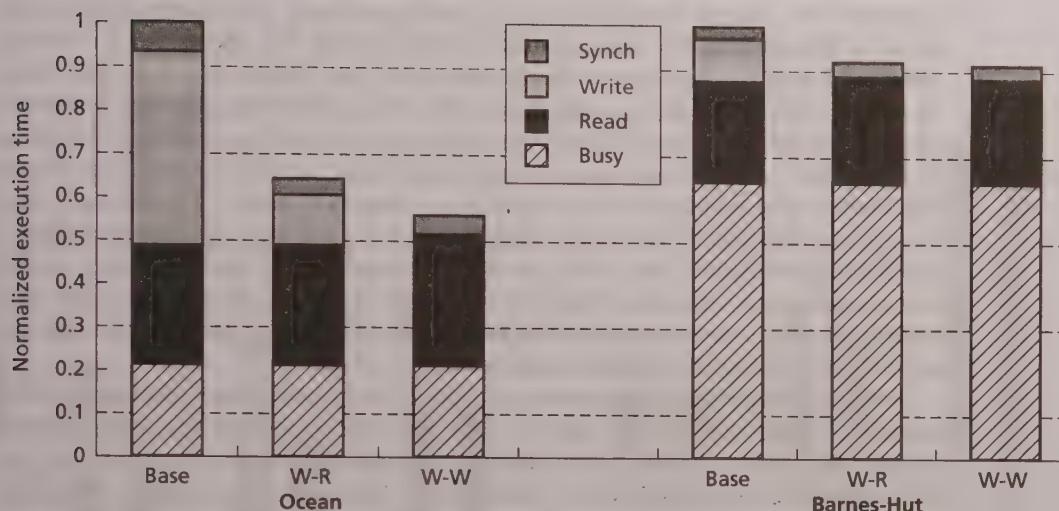


FIGURE 11.15 Performance benefits from proceeding past writes by taking advantage of relaxed consistency models. The results assume a statically scheduled processor model. For each application, Base is the case with no latency hiding, W-R indicates that reads can bypass writes (i.e., the write → read order can be violated), and W-W indicates that both writes and reads can bypass writes. The execution time for the Base case is normalized to 100 units. The system simulated is a cache-coherent multiprocessor using a flat, memory-based directory protocol much like that of the Stanford DASH multiprocessor (Lenoski et al. 1993). The simulator assumes write-through first-level and write-back second-level caches, with deep, 16-entry write buffers between the two. The processor is single issue and statically scheduled, with no special scheduling in the compiler targeted toward latency tolerance. The write buffer is aggressive, with read bypassing of it and read forwarding from it enabled. To preserve the write → write order (in the case where this reordering is not permitted), writes are not merged in the write buffer and the write at the head of the FIFO write buffer is not retired until it has completed. The data access parameters assumed for a read miss are 1 cycle for an L₁ hit, 15 cycles for an L₂ hit, 29 cycles if a miss is satisfied in local memory, 101 cycles for a two-message remote miss, and 132 cycles for a three-message remote miss. Write latencies are a little smaller in each case. The system thus assumes a much slower processor relative to a memory system than modern systems.

completes more quickly. In most applications, however, write-write overlap provides little benefit since most of the write latency is already hidden by allowing reads to bypass previous writes. Another factor limiting the effectiveness of write-write overlap is that the processor model assumes that reads are blocking, so write-write overlap cannot be exploited past read operations. The differences between models like weak ordering and release consistency as well as subtle differences among models (such as TSO, which preserves write atomicity, and PC, which does not) do not seem to affect performance substantially.

Since performance is highly dependent on implementation and on problem and cache sizes, it is useful to examine less aggressive write buffers and second-level cache architectures as well as cache sizes where an important working set does not fit in the cache. A lockup-free L₂ cache is very important for obtaining good performance improvements, as we might expect. Bypassable write buffers are very

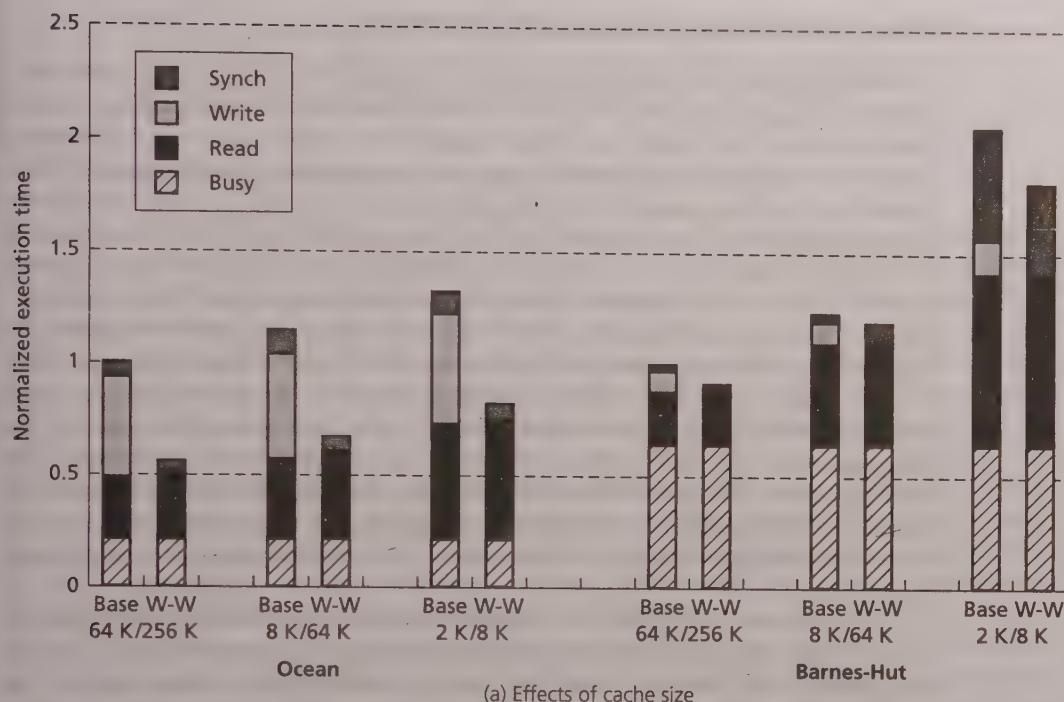
important for a system that allows write → read reordering, particularly when the L₂ cache is lockup-free, but less important for the system that allows write → write reordering as well. The reason is that in the former case writes retire from the buffer only after completion, so it is quite likely that when a read misses in the first-level cache it will find a write still in the write buffer, and stalling the read until the write buffer empties will hurt performance. In the latter case, writes retire much faster, so the likelihood of finding a write in the buffer to bypass is smaller. For the same reason, write buffer size is less critical when write → write reordering is allowed. Without lockup-free caches, the ability for reads to bypass writes in the buffer is much less advantageous whether or not write → write reordering is allowed, since the stalling of the L₂ cache becomes the bottleneck. All parts of the system must be appropriately designed to obtain the benefits of overlap.

Results from the study with smaller L₁ and L₂ caches are shown in Figure 11.16, as are results for varying the cache block size. Consider cache size first. With smaller caches, write latency is still hidden effectively by allowing write → read and write → write reordering. (As discussed in Chapter 4, for Barnes-Hut the smallest caches do not represent a realistic scenario.) Interestingly, the impact of hiding write latency on overall performance is often larger with larger caches even though the total write latency to be hidden is smaller. This is because larger caches are more effective in reducing read latency than write latency, so the latter becomes relatively more important to hide. All of these results assume a cache block size of only 16 bytes, which is quite unrealistic today. Larger cache blocks tend to reduce the miss rates for these applications and hence somewhat reduce the impact of these reorderings on execution time, as seen for Ocean in Figure 11.16(b).

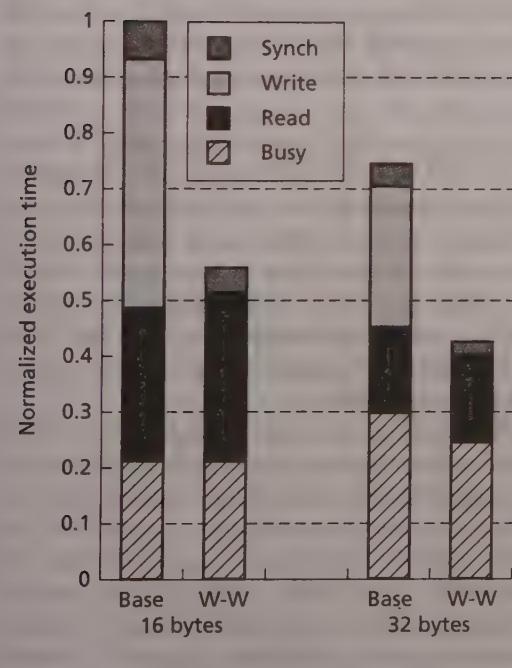
11.5.2 Proceeding Past Reads

To hide read latency effectively, we need both nonblocking reads and a mechanism to look ahead beyond dependent instructions. Both compiler and hardware mechanisms are complicated by the fact that the dependent instructions may be followed soon after by branches. Predicting future paths through the code to find independent instructions requires effective branch prediction as well as speculative execution past predicted branches. Speculatively executed instructions in turn demand hardware support to cancel their effects upon detecting misprediction.

The trend in the microprocessor industry today is toward increasingly sophisticated processors that provide all these features in hardware. For example, they are included by processor families such as the Intel Pentium Pro (Intel Corporation 1996), the Silicon Graphics R10000 (MIPS Technologies 1996), the Sun UltraSparc (Lee, Kwok, and Briggs 1991), and the Hewlett-Packard PA8000 (Hunt 1996) because latency hiding and overlapped operation of the memory system and functional units are very important even in uniprocessors. The mechanisms have a high design and hardware cost, but since they are already present in the microprocessors they can be used to hide latency in multiprocessors as well. Once present, they can hide write latency as well, so a separate write buffer may not be needed.



(a) Effects of cache size



(b) Effects of cache block size for Ocean

FIGURE 11.16 Effects of cache size and cache block size on the benefits of proceeding past writes. In (a), which varies cache size, the cache block size assumed is the (small) 16-byte default size used in the study. The cache sizes specified on the x-axis are the L₁ and L₂ cache sizes, separated by a slash. In (b), which varies cache block size, the cache sizes are assumed to be the default 64-KB L₁ cache and 256-KB L₂ cache used in the study. The y-axis is the normalized execution time so that the leftmost bar in each graph is one unit.

Dynamic Scheduling and Speculative Execution

To understand how to hide memory latency by using dynamic scheduling and speculative execution and especially how the techniques interact with memory consistency models, let us briefly recall from uniprocessor architecture how the methods work. More details can be found in texts, such as (Hennessy and Patterson 1996). *Dynamic scheduling* means that instructions are fetched and decoded in program order as presented by the compiler, but they are executed by the functional units in the order in which their operands become available at run time. One way to orchestrate this out-of-order execution is through reservation stations and Tomasulo's algorithm (Hennessy and Patterson 1996). Dynamic scheduling does not necessarily imply that memory operations will become visible or complete out of program order, as we will see. *Speculative execution* means allowing the processor to look at and schedule for execution instructions that are not necessarily going to be useful to the program's execution, for example, instructions past a future branch instruction. The functional units can execute these instructions, assuming that they will indeed be useful, but the results are not committed to registers or made visible to the rest of the system until the validity of the speculation has been determined (e.g., the branch has been resolved).

The key mechanism used for speculative execution is an instruction *lookahead* or *reorder* buffer. As instructions are decoded by the decode unit, whether in-line or speculatively past predicted branches, they are placed in the reorder buffer. The reorder buffer therefore holds instructions in program order. Among these, instructions that are not dependent on an incomplete instruction can be chosen from the reorder buffer for execution. It is quite possible that independent long-latency instructions (e.g., reads) further ahead in the buffer have not yet been executed or completed, so in this way the processor proceeds past incomplete memory operations and other instructions. Instructions in the reorder buffer become ready for execution as soon as their operands are produced by other instructions, not necessarily waiting until the operands are available in the register file. An instruction keeps its results with it in the reorder buffer without committing them to the register file and is retired from the buffer only when it reaches the head, that is, in program order. It is only at this retirement point that the result of a read or other instruction may be put in the destination register and that the value produced by a write is free to be made visible to the memory system. Thus, even with aggressive out-of-order execution, memory operations can complete in program order.

Retiring instructions in program order simplifies speculation: if a branch is found to be mispredicted, which is determined before the branch retires from the reorder buffer, then no instruction after it (in program order) could have retired from the reorder buffer and committed its effects. No read has updated a register, and no write has become visible to the memory system. Upon detecting misprediction, all instructions after the branch are invalidated in the reorder buffer and the reservations stations, and decoding is resumed from the correct branch target. In-order retirement also makes it easy to implement precise exceptions. However, in-order retirement does mean that if a read miss reaches the head of the buffer before its data

value has come back, the reorder buffer (not the processor) stalls and later instructions cannot be retired. This FIFO nature and stalling implies that the extent of latency tolerance may be limited by the size of the reorder buffer.

Hiding Latency under Sequential and Release Consistency

The difference between an operation retiring from the reorder buffer and completing is important. A read completes when its value is bound, which may be before it reaches the head of the reorder buffer and can retire (modifying the processor register). On the other hand, a write is not complete even when it reaches the head of the buffer and retires to the memory system; it completes only when it has actually become visible to all processors. Understanding this difference helps us understand how out-of-order, speculative processors can hide latency under both SC and more relaxed models like release consistency (RC).

Under RC, a write may indeed be retired from the buffer before it completes, allowing operations behind it to retire more quickly. Under SC, a write is retired from the buffer only when it has completed (or at least committed with respect to all processors), so it may hold up the reorder buffer longer once it reaches the head and is sent out to the memory system. Under RC, a read may be issued to the memory system and complete anytime after it is inserted in the reorder buffer, unless an acquire operation before it has not completed. Under SC, although a read may still be issued to the memory system and complete before it reaches the head of the buffer, it is not issued to the memory system before all previous memory operations in the reorder buffer have completed (not necessarily retired). Thus, SC exploits less overlap than RC, but the difference is not as great as with in-order execution.

Additional Techniques to Enhance Latency Tolerance

Additional techniques—including a form of hardware prefetching, speculative reads, and write buffers—can be used with dynamically scheduled, speculative processors to hide latency further under SC as well as RC (Gharachorloo, Gupta, and Hennessy 1991b). The hardware may issue prefetch operations for memory operations that are in the reorder buffer but are not yet permitted by the consistency model to actually issue to the memory system. For example, the processor might prefetch a read that is preceded in the buffer by another incomplete memory operation in SC or by an incomplete acquire operation in RC, thus overlapping them. For writes, prefetching allows the data and ownership to be brought to the cache before the write actually gets to the head of the buffer and can be issued to the memory system. Basically, the read-exclusive transaction is issued early. These prefetches are *nonbinding*, which means that the data is brought into the cache, but not into a register or the reorder buffer, and is still visible to the coherence protocol so the prefetches do not violate the consistency model. If the block is invalidated before the read commits, the only harm is a little extra memory traffic. (Prefetching in a more general context will be discussed in Section 11.6.)

Hardware prefetching is ineffective when the address to be prefetched is not known and determining the address itself requires the completion of a long-latency operation. For example, if a read miss to an array entry $A[I]$ is preceded by a read miss to the array index I , then the two operations cannot be overlapped because the processor does not know the address of $A[I]$ until the read of I completes. It can prefetch I , but to obtain the address of $A[I]$ requires that the read of I complete. To increase the overlap, a processor can use speculative read operations. A *speculative read* is a read that completes speculatively even before its completion is permitted by the consistency model, that is, before it reaches the head of the reorder buffer, in the case of SC. Its value can then be used as an address for future memory operations, but the use will be guarded as for instructions after a predicted branch. Essentially, the processor speculates that the (prefetched) value will not be changed between the time of the speculative read and the time that the real read is allowed to be performed according to the memory consistency model. If the value is indeed changed during this time, then the effects of the speculative read and all operations that have been issued after it must be undone.

In the current example, I is not only prefetched but also speculatively read before it has reached the head of the buffer, so the read of $A[I]$ can be prefetched early as well. We need to be sure that we use the correct value for I and hence read the correct location for $A[I]$. For this reason, speculative reads are loaded not into the registers themselves but into a buffer called the speculative read buffer where they stay until the read retires from the reorder buffer. The speculative read buffer “watches” the cache and is apprised of cache actions to those blocks. If an invalidation, update, or even cache replacement occurs on a block whose address is in the speculative read buffer, then the speculative read and all instructions following it in the reorder buffer must be cancelled and the execution rolled back, just as on a mispredicted branch. Such hardware prefetching and speculative read support are present in processors like the Pentium Pro (Intel Corporation 1996), the MIPS R10000 (MIPS Technologies 1996), and the HP PA-8000 (Hunt 1996). Note that under SC every read issued before previous memory operations have completed is a speculative read and goes into the speculative read buffer, whereas under RC reads are speculative (and have to watch the cache) only if they are issued before a previous acquire has completed.

A final optimization used to increase overlap, even in a dynamically scheduled processor, is a separate write buffer. Instead of writes waiting at the head of the reorder buffer until they complete, holding up the reorder buffer, they are removed from the reorder buffer and placed in the write buffer when they reach the head. The write buffer allows them to become visible to the extended memory hierarchy and keeps track of their completion as required by the consistency model. Write buffers are clearly useful with relaxed models like RC and PC in which reads are allowed to bypass writes. The reads will now reach the head of the reorder buffer and retire more quickly. Under SC, we can put writes in the write buffer, but a read stalls when it reaches the head of the reorder buffer anyway until the write completes. Thus, much of the latency that the write would have seen may instead be seen by the next read to reach the head of the reorder buffer.

Performance Impact

Simulation studies have examined the extent to which hardware prefetching, speculative read, and write buffering techniques can hide latency under different consistency models. One study assuming an RC model finds that a substantial portion of read latency can indeed be hidden using a dynamically scheduled processor with speculative execution and that the amount of read latency that can be hidden increases with the size of the reorder buffer even up to buffers as large as 64 to 128 entries (Gharachorloo, Gupta, and Hennessy 1992). A detailed study compares the performance of the SC and RC consistency models with aggressive, multiple-issue, dynamically scheduled processors (Pai et al. 1996). It also examines the benefits obtained individually from hardware prefetching and speculative reads with each model. Because a write takes longer to complete even on an L_1 hit when the L_1 cache is write through rather than write back, the study examines both types of L_1 caches, always assuming a write-back L_2 cache. The studies are preliminary since they use very small problem sizes and scaled-down caches. They also do not use sophisticated compiler technology to schedule instructions in a way that can obtain increased benefits from dynamic scheduling and speculative execution. (For example, placing more independent instructions close after a memory operation or miss may allow smaller reorder buffers to suffice.) However, the results shed light on the interactions between mechanisms and models.

The most interesting question is whether, with aggressive, dynamically scheduled processors, RC still buys substantial performance gains over SC at the hardware/software interface. If not, the programming burden of relaxed consistency models may not be justified with these processors. (Relaxed models may still be important at the programmer's interface to allow compiler optimizations, but the programming burden may be lighter if it is only the compiler that may reorder operations.) The results of the second study, shown for two programs in Figures 11.17 and 11.18, indicate that RC is still beneficial, even though the gap has closed substantially compared to the case of processors with blocking reads. The figures show the results for SC without any of the more sophisticated optimizations (hardware prefetching, speculative reads, and write buffering) and then with those optimizations applied cumulatively one after the other. Results are also shown for processor consistency (PC) and for RC. The PC and RC cases always assume write buffering and are shown first without the other two optimizations and then with those cumulatively.

When hardware prefetching and speculative reads are not used, RC has substantial advantages over SC even with a dynamically scheduled processor. This is primarily because RC is able to hide write latency more successfully than SC, as was the case with simple processors. It allows writes to be retired faster and allows later accesses to be issued to the memory system and to complete before previous writes. The improvement due to RC is even greater with write-through caches since under SC a write that reaches the head of the buffer issues to the memory system but has to wait until the write performs in the second-level cache even if it hits in the first-level cache. While read latency is hidden with some success even under SC, RC allows for much earlier issue, completion, and hence retirement of reads.

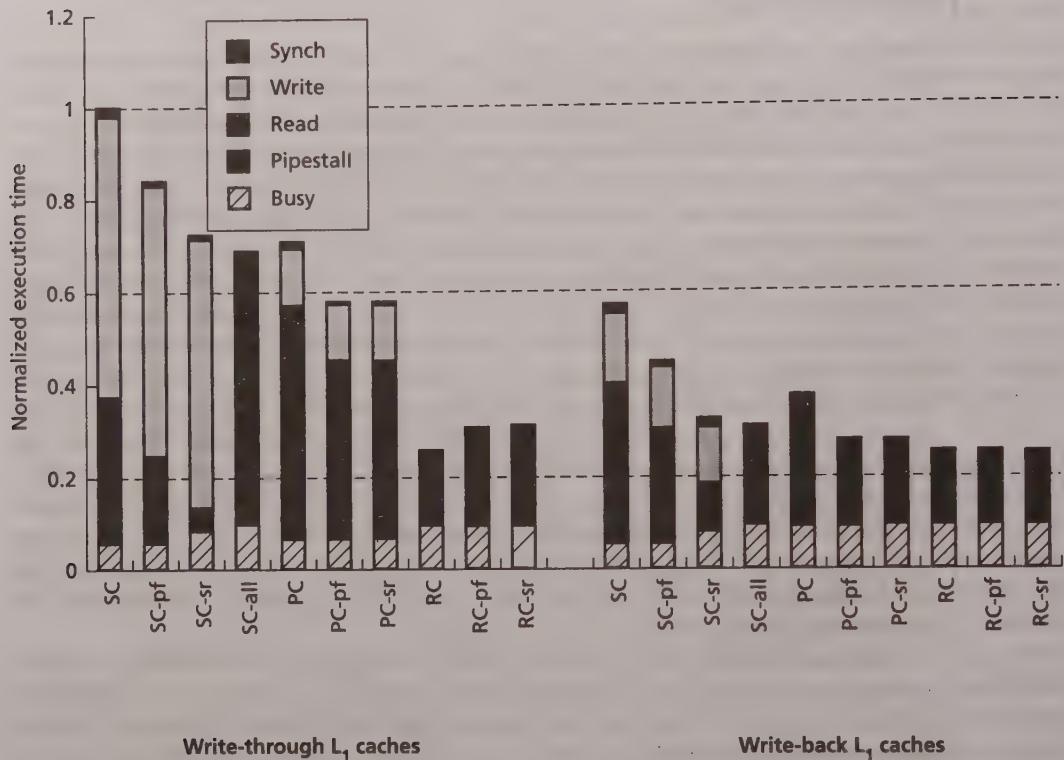


FIGURE 11.17 Performance of an FFT kernel with different consistency models, assuming a dynamically scheduled processor with speculative execution. The set of ten bars on the left assumes write-through first-level caches, while the ten bars on the right assume write-back first-level caches. Second-level caches are always write back. SC, PC, and RC are sequential, processor, and release consistency, respectively. For SC there are four bars. The first bar excludes hardware prefetching, speculative reads, and write buffers; the second bar (pf) includes the use of hardware prefetching, the third bar (sr) includes the use of speculative reads as well, and the fourth bar (all) includes all three optimizations. For PC and RC, the use of write buffers is always assumed, so there are only three sets of bars each (pf now means hardware prefetching and write buffering, and sr includes all three optimizations). The processor model assumed resembles the MIPS R10000 (MIPS Technologies 1996). The processor is clocked at 300 MHz and is capable of issuing 4 instructions per cycle. It uses a reorder buffer of size 64 entries, a merging write buffer with 8 entries, a 4-KB direct-mapped first-level cache, and a 64-KB, 4-way set-associative second-level cache. Small caches are chosen since the data sets are small, but they may exaggerate the effect of latency hiding. More detailed parameters can be found in (Pai et al. 1996).

The effects of hardware prefetching and speculative reads are much greater for SC than for RC since in RC, memory operations are allowed to issue and complete out of order anyway. However, even with these optimizations a significant gap still remains compared to RC, especially in write latency. Write buffering is not very useful under SC for the reason discussed earlier.

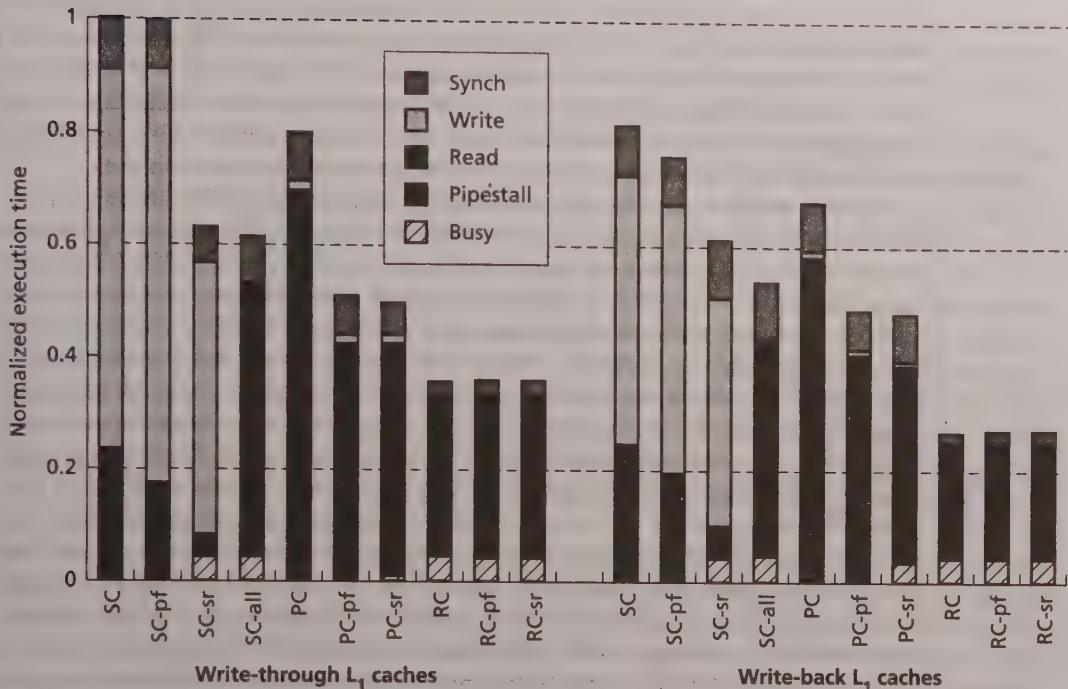


FIGURE 11.18 Performance of Radix with different consistency models, assuming a dynamically scheduled processor with speculative execution. The system assumptions and the organization of the figure are the same as in Figure 11.17

The figures also confirm that write latency is a more significant problem with write-through first-level caches under SC than with write-back caches but is hidden equally well under RC with both types of caches. The difference in write latency between write-through and write-back caches under SC is larger for FFT than for Radix because in FFT the writes are to locally allocated data and hit in the write-back cache for this problem size whereas in Radix they are to nonlocal data and miss in both cases. Overall, PC rates between SC and RC, with hardware prefetching helping but speculative reads not helping as much as in SC.

Finally, the graphs appear to reveal an anomaly: the processor busy time is different in different schemes, even for the same application, problem size, and memory system configuration. The reason is an interesting methodological point for superscalar processors. Since several instructions can be issued every cycle, how do we decide whether to attribute a cycle to busy time or to a particular type of stall time?

There isn't a very good answer. The decision made in most studies in the literature, and in these results, is to attribute a cycle to busy time only if all instruction slots issue in that cycle. If not, then the cycle is attributed to whatever form of stall is seen by the first instruction (starting from the head of the reorder buffer) that should have issued in that cycle but didn't. Since this pattern changes with consistency models and implementations, the busy time is not the same across schemes.

It is interesting to examine the interactions with hardware prefetching and speculative reads in a little more detail, particularly in how they interact with application characteristics. Prefetching for operations that are in the reorder buffer works most successfully when a number of operations that will otherwise miss are close together in the code so that they will appear together in the reorder buffer. This happens in the matrix transposition phase of an FFT, so the gains from prefetching in Figure 11.17 are substantial. It can be aided in other programs by appropriate scheduling of operations by the compiler. The situation that motivated speculative reads in the first place—the address to be prefetched is not known until a read completes—is encountered in the Radix sorting application, shown in Figure 11.18. Here, the relevant misses are to array entries indexed by histogram values that have to be read as well. An interesting effect in both programs is that the processor busy time is reduced as well by speculative reads. This is because the ability to consume the values of reads speculatively makes many more otherwise dependent instructions available to execute, greatly increasing the utilization of a superscalar processor. Interestingly, speculative reads help reduce read latency in FFT even though it does not have indirect array accesses. This is because conflict misses in the L₂ cache are reduced due to greater combining of accesses to an outstanding cache block: accesses that would otherwise have caused conflict misses in the L₂ cache are overlapped by using speculative reads and are therefore combined by the mechanism used to keep track of outstanding L₂ misses. This illustrates that important observed effects sometimes are not directly due to the feature being studied. Although speculative reads, and speculative execution in general, are hurt by misspeculation and consequent rollbacks, this occurs rarely in the programs studied, which take quite predictable and straightforward paths through the code. The results may be different for programs with highly unpredictable control flow and access patterns.

11.5.3 Summary

The extent to which latency can be tolerated by proceeding past reads and writes in a multiprocessor depends on both the aggressiveness of the implementation and the memory consistency model. Tolerating write latency is relatively easy in cache-coherent multiprocessors, even with simple blocking-read processors, when the memory consistency model is relaxed. Modern dynamically scheduled processors can hide both read and write latency, but only partially. The instruction lookahead window (reorder buffer) size needed to hide read latency can be substantial and grows with the latency to be hidden. Fortunately, latency is hidden increasingly as window size increases rather than needing a very large threshold size before any significant latency hiding takes place. In fact, with the mechanisms that are widely

available in modern processors, read latency can be hidden reasonably well even under sequential consistency, at least on moderate-scale systems. Compiler scheduling of instructions can help processors hide latency even better.

In general, conservative design choices—such as blocking reads or blocking caches—make preserving orders easier, but at the cost of performance. For example, in dynamically scheduled processors, delaying the retirement of writes from the reorder buffer until all previous instructions are complete to avoid rolling back on writes (e.g., for precise exceptions) makes it easier to preserve SC, but it makes write latency more difficult to hide under SC. Both read and especially write latency are hidden better by using relaxed consistency models. The implementation requirements for relaxed consistency models in hardware-coherent systems are not very demanding beyond what is provided to hide write and read latency in modern uniprocessors and what is needed even for sequential consistency (see Exercise 11.9). Most of the support for preserving a given consistency model is in the buffers and caches close to the processor; the rest of the memory hierarchy can then reorder transactions as it pleases.

The approach of hiding latency by proceeding past operations has two significant drawbacks. The first is that it may very well require relaxing the consistency model to be very effective, especially with simple statically scheduled processors but also with dynamically scheduled ones. This places a greater burden on the programmer to label synchronization (competing) operations or insert memory barriers, although relaxing the consistency model is to a large extent needed to allow compilers to perform many of their optimizations anyway. The second drawback is the difficulty of hiding read latency effectively with processors that are not dynamically scheduled and the resource requirements of hiding multiprocessor read latencies even with processors that are dynamically scheduled, especially as latencies become larger. In these situations, other methods like precommunication and multithreading might be more successful at hiding read latency and other forms of latency and may be used in conjunction with proceeding past memory operations in the same thread.

11.6 PRECOMMUNICATION IN A SHARED ADDRESS SPACE

Precommunication support, especially prefetching, has also been widely adopted in commercial microprocessors, and its importance is likely to increase in the future. To understand the techniques for precommunication, let us first consider a shared address space with no caching of shared data, whereby all data accesses go to the relevant main memory. After the introduction of some basic concepts, we examine prefetching in a cache-coherent shared address space, including performance benefits and implementation issues.

11.6.1 Shared Address Space without Caching of Shared Data

In a shared address space without caching of shared data, receiver-initiated communication is triggered by reads of nonlocally allocated data, and sender-initiated communication is triggered by writes to nonlocally allocated data. In the baseline

communication structure of our example code (Figure 11.1), the communication is sender initiated if we assume that the array A is allocated in the local memory of process P_B , not P_A . As with the sends in the message-passing case, the most precommunication we can do is to perform all the writes to A before we compute any of the $f(B[])$ by splitting into two loops (see the first column of Figure 11.19). Making the writes nonblocking allows some of the write latency to be overlapped with the $f(B[])$ computations; however, this would have happened anyway if writes were made nonblocking and left where they were. The more important effect of the precommunication is to have the writes on P_A complete earlier, hence allowing the reader P_B to emerge from its while loop more quickly.

If the array A is allocated in the local memory of P_A , the communication is receiver initiated. The writes by the producer P_A are now local, and the reads by the consumer P_B are remote. Precommunication in this case means *prefetching* the elements of A before they are actually needed, just as we issued `a_receives` before they were needed in the message-passing case. The difference is that the prefetch is not just posted locally, like the receive, but rather causes data transfer across the network: a prefetch request is sent across the network to the remote node (P_A) where the communication assist responds to the request by actually transferring the data back. In the meantime, P_B proceeds with other work. There are many ways to implement prefetching, as we shall see. One is to issue a special *prefetch* instruction and build a software pipeline as in the message-passing case. The shared address space code with prefetching is shown in Figure 11.19. The software pipeline has a prologue that issues a prefetch for the first iteration, a steady-state period of $n - 1$ iterations in which a prefetch is issued for the next iteration and the current iteration is executed, and an epilogue consisting of the work for the last iteration. Note that a prefetch instruction does not replace the actual read of the data item (the load instruction), which happens in its original place in the program. Further, the prefetch instruction itself must be nonblocking (must not stall the processor) if it is to achieve its goal of hiding latency through overlap.

Since shared data is not cached in this case, the prefetched data is brought into a special hardware structure called a prefetch buffer. When the word is actually loaded into a register in the next iteration, it is read from the head of the prefetch buffer rather than from memory. If the latency to hide were much larger than the time to compute a single loop iteration, we would prefetch several iterations ahead and the prefetch buffer would potentially hold several words at a time. The CRAY T3D multiprocessor provides such a prefetch buffer and ensures that data becomes available in the buffer in the order that the prefetches issue, so the processor can read from it in the same order. The CRAY T3E uses a set of external registers as a prefetch buffer.

Even if data cannot be prefetched early enough to have arrived by the time the actual reference is made, prefetching is beneficial. If the actual reference finds that the address it is accessing has an outstanding prefetch associated with it, then it can simply wait for the remaining time until the prefetched data returns, thus hiding part of the latency. In addition, depending on the number of prefetches allowed to be outstanding at a time, prefetches can be issued back to back to overlap their laten-

P_A	P_B
<pre> for i←0 to n-1 do compute A[i]; write A[i]; end for flag ← 1; for i←0 to n-1 do compute f(B[i]); </pre>	<pre> while flag = 0 {}; prefetch(A[0]); for i←0 to n-2 do prefetch(A[i+1]); read A[i] from prefetch_buffer; use A[i]; compute g(C[i]); end for read A[n-1] from prefetch_buffer; use A[n-1]; compute g(C[n-1]) </pre>

FIGURE 11.19 Prefetching in the shared address space example. The example assumes that shared data is not cached, so prefetched data is read from the prefetch buffer rather than the cache.

cies. This provides pipelined data movement, although the pipeline rate may be limited by the overhead of issuing prefetches.

11.6.2 Cache-Coherent Shared Address Space

Precommunication is much more interesting in a cache-coherent shared address space since shared nonlocal data may be precommunicated directly into a processor's cache rather than a special buffer and since precommunication interacts with the cache coherence protocol. We therefore discuss the techniques in more detail in this context.

Consider an invalidation-based coherence protocol. A read miss fetches the data from wherever it is. A write that generates a read exclusive fetches both data and ownership (by informing the home, perhaps invalidating other caches, and receiving acknowledgments), and a write that generates an upgrade fetches only ownership. All of these "fetches" have latency associated with them, so all are candidates for prefetching. We can prefetch data or ownership or both.

Update-based coherence protocols generate sender-initiated communication, and like other sender-initiated communication techniques they provide a form of precommunication from the viewpoint of the destinations of the updates. Although update protocols are not very prevalent, techniques to selectively update copies can be used for sender-initiated precommunication even with an underlying invalidation-based protocol. One possibility is to selectively insert software instructions that generate updates; another is to use hybrid update-invalidate methods. Some of these techniques are discussed later in this section.

Prefetching Concepts

Two broad categories of prefetching are applicable to multiprocessors and uniprocessors: hardware-controlled and software-controlled prefetching. In hardware-controlled prefetching, no special instructions are added to the code; rather, special hardware is used to predict future accesses from observed behavior and to prefetch data based on these predictions. In software-controlled prefetching, the decisions of what and when to prefetch are made by the programmer or compiler (hopefully the compiler!) by analyzing the code, and appropriate prefetch instructions are inserted in the code. Trade-offs between hardware- and software-controlled prefetching are discussed later in this section. Prefetching can also be combined with block data transfer in *block prefetch* (receiver-initiated prefetch of a large block of data) and *block put* (sender-initiated) techniques.

In a multiprocessor, a key issue that dictates how early a prefetch can be issued in both software- and hardware-controlled prefetching is whether prefetches are binding or nonbinding. A *binding* prefetch means that the value of the prefetched data is bound at the time of the prefetch; that is, when the process later reads the variable through a regular read (load instruction), it will see the value that the variable had when it was prefetched even if the value has been modified by another processor and the new value has become visible to the reader's cache in between the prefetch and the actual read. The prefetch we discussed in the non-cache-coherent case (prefetching into a prefetch buffer, see Figure 11.19) is typically a binding prefetch, as is prefetching directly into processor registers. A *nonbinding* prefetch means that the value brought by a prefetch instruction remains subject to modification or invalidation until the actual operation that needs the data is executed, as discussed in Section 11.5.2. For example, in a cache-coherent system with nonbinding prefetch, the prefetched data is brought into the cache rather than into a register or prefetch buffer (neither of which is typically under control of the coherence protocol), and a modification by another processor that occurs between the time of the prefetch and the time of the use will update or invalidate the prefetched block according to the protocol. This means that nonbinding prefetches can be issued at any time without affecting the semantics of a parallel program or the results it may produce. Binding prefetches, on the other hand, affect program semantics, so we have to be careful about issuing them too early. For example, if processes increment a shared counter in a critical section, it is unsafe to issue a binding prefetch for the counter before (outside) the critical section since another process may obtain the lock and modify the counter between the prefetch and the lock acquisition; however, issuing a non-binding prefetch before the critical section is safe. Since nonbinding prefetches can be generated as early as desired, they have performance advantages as well.

The other important issues concern determining what data to prefetch (*analysis*) and when to initiate prefetches (*scheduling*). Prefetching a given reference is considered *possible* only if the address of the reference can be determined ahead of time. For example, if the address can be computed only just before the word is referenced, then it may not be possible to prefetch. This is an important consideration for applications with irregular and dynamic data structures implemented using pointers, such as linked lists and trees.

The *coverage* of a prefetching technique is the percentage of the original cache misses (without any prefetching) that the technique is able to prefetch earlier than just before the actual reference. Achieving high coverage is not the only goal, however. We should not issue prefetches for data that will not be accessed by the processor or for data that is already in the cache that is the target of a prefetch. These prefetches will consume overhead and precious cache access bandwidth, interfering with regular accesses without doing anything useful. More is not necessarily better for prefetching. These are called *unnecessary* prefetches. Avoiding them requires that we analyze the data locality in the application's access patterns and how it interacts with the cache size and organization so that prefetches are issued only for data that is not likely to be present in the cache.

Finally, timing and luck play important roles in prefetching. A prefetch may be possible and not unnecessary, but it may be initiated too late to hide most of the latency from the actual reference. Or it may be initiated too early, so it arrives in the cache but is then either replaced or invalidated before the actual reference. Thus, a prefetch should be *effective*: early enough to hide the latency and late enough so the chances of replacement or invalidation are small.

The goal of prefetching analysis is to maximize coverage while minimizing unnecessary prefetches, and the goal of scheduling is to maximize effectiveness. Let us now consider hardware-controlled and software-controlled prefetching in some detail and see how successfully they address these important aspects.

Hardware-Controlled Prefetching

The goal in hardware-controlled prefetching is to provide hardware that can detect patterns in data accesses at run time. Hardware-controlled prefetching assumes that accesses in the near future will follow past patterns. Under this assumption, the cache blocks containing this data can be prefetched and brought into the processor cache so the later accesses may hit in the cache. The following discussion assumes nonbinding prefetches.

Both analysis and scheduling are the responsibility of hardware, with no special software support, and both are performed dynamically as the program executes. Analysis and scheduling are very closely coupled since the prefetch for a cache block is initiated as soon as it is determined that the block should be prefetched: it is difficult for hardware to make separate decisions about these issues.

The hardware should be simple and inexpensive, and it should not be in the critical path of the processor cycle time.

Many simple hardware prefetching schemes have been proposed. At the most basic level, the use of long cache blocks itself is a form of hardware prefetching, exploited well by programs with good spatial locality. No analysis is used to restrict unnecessary prefetches, the coverage depends on the degree of spatial locality in the program, and the effectiveness depends on how much time elapses between when the processor accesses the first word and when it accesses the other words on the block. For example, if a process simply traverses a large array with unit stride, the coverage of the prefetching with long cache blocks will be quite good (75% for a

cache block of four words) and there will be no unnecessary prefetches—but the effectiveness is not likely to be great since the prefetches are issued too late. Extending the idea of long cache blocks are *one-block lookahead* (OBL) schemes, in which a reference to cache block i may trigger a prefetch of block $i + 1$. Several variants for analysis and scheduling may be used in this technique; for example, block $i + 1$ can be prefetched whenever a reference to block i is detected, or only if a cache miss to i is detected, or when i is referenced for the first time after it is prefetched. Extensions may include prefetching several blocks ahead (e.g., blocks $i + 1$, $i + 2$, and $i + 3$) instead of just one (Dahlgren, Dubois, and Stenstrom 1995), an adaptation of the idea of *stream buffers* in uniprocessors where several subsequent blocks are prefetched into a separate buffer, rather than the cache, when a miss occurs (Jouppi 1990). Such techniques are useful when accesses are mostly unit stride.

A simple way to detect and prefetch accesses with nonunit or large stride is to keep the address of the previously accessed data item for a given instruction (i.e., for a given program counter value) in a history table indexed by program counter (PC). When the same instruction is issued again (e.g., in the next iteration of a loop), if the PC is found in the table, the stride is computed as the difference between the current data address and the one in the history table for that instruction or PC. A prefetch is issued for the data address computed as the current data address plus the stride (Fu and Patel 1991; Fu, Patel, and Janssens 1992). The history table, managed much like a branch history table, essentially detects regular strides in data accesses by an instruction and predicts that future accesses by that instruction will follow the same stride. This scheme is likely to work well when the stride is constant; however, most other prefetching schemes that we shall discuss, hardware or software, are likely to work well in this case too.

While the schemes so far find ways to detect simple regular patterns, they do not guard against unnecessary prefetches when references do not follow these patterns. For example, if the same stride is not maintained between three successive accesses by the same instruction, then the previous scheme will prefetch useless data. The traffic generated by these unnecessary prefetches can be detrimental to performance by competing for resources with useful regular accesses.

In more sophisticated hardware prefetching schemes, the history table stores not only the data address accessed the last time by an instruction but also the stride between the previous two addresses accessed by it (Baer and Chen 1991; Chen and Baer 1992). If the current data address accessed by that instruction is separated from the previous address by the same stride, then a regular stride pattern is detected and a prefetch may be issued. If not, then a break in the pattern is detected and a prefetch is not issued, thus reducing unnecessary prefetches. In addition to the address and stride, the table entry also contains some state bits that keep track of whether the accesses by this instruction have recently been in a regular stride pattern, have been in an irregular pattern, or appear to be transitioning into or out of a regular stride pattern. A set of simple rules is used to determine, based on the stride match and the current state, whether to potentially issue a prefetch or not. If the result is to potentially prefetch, the cache is looked up to see if the block is already there, and if not the prefetch is issued.

While this scheme improves the analysis, it does not yet do a great job of scheduling prefetches. In particular, in a loop it will prefetch only one loop iteration ahead. If the amount of work to do in a loop iteration is small compared to the latency that needs to be hidden, this will not be sufficient to tolerate the latency. The goal of scheduling is to achieve just-in-time prefetching, that is, to have a prefetch issued about l cycles before the instruction that needs the data, where l is the latency we want to hide, so that prefetched data arrives just before the actual instruction that needs it is executed and the prefetch is likely to be effective. This means that the prefetch should be issued $\lceil l/b \rceil$ loop iterations ahead of the instruction that needs the data, where b is the predicted execution time of an iteration of the loop body.

One way to implement such scheduling in hardware is to use a *lookahead program counter* (LA-PC) that tries to remain l cycles ahead of the actual current PC. The LA-PC is used to access the history table and generate prefetches instead of (but in conjunction with) the actual PC. The LA-PC starts out a single instruction ahead of the regular PC but is incremented every cycle even when the processor (and PC) stalls on a cache miss, thus letting it get ahead of the PC. The LA-PC also looks up the branch history table, just like the PC, so the branch prediction mechanism can be used to modify it when necessary and to try to keep it on the right track. When a mispredicted branch for the LA-PC is detected, the LA-PC is set back to being equal to PC + 1. A limit register controls how far the LA-PC can get ahead of the PC. The LA-PC stalls when this limit is exceeded or when the buffer of outstanding references is full (i.e., it cannot issue any prefetches).

Both the LA-PC and the PC look up the prefetch history table every cycle (of course, they are likely to access different entries). The lookup by the PC updates the “previous address” and the state fields for the entry that it hits, in accordance with the rules, but does not generate prefetches. A new “times” field is added to each history table entry, which keeps track of the number of iterations (encounters of that instruction) that the LA-PC is ahead of the PC. The lookup by the LA-PC increments this field for the entry that it hits (if any) and generates an address for potential prefetching according to the rules. The address generated is the times field multiplied by the stride stored in the entry, plus the previous address field. The times field is decremented when the PC encounters that instruction in its lookup of the prefetch history table. More details can be found in (Chen and Baer 1992).

Prefetches in these hardware schemes are treated as hints since they are nonbinding, so actual cache misses get priority over them in the extended hierarchy. If a prefetch raises an exception (for example, a page fault or other violation), the prefetch is simply dropped rather than handling the exception. More elaborate hardware-controlled prefetching schemes have been proposed to try to prefetch references with irregular stride (Zhang and Torrellas 1995). However, even the simpler techniques have not found their way into microprocessors for multiprocessors. Instead, the trend is to provide prefetch instructions for use by software-controlled prefetching schemes. Let us examine software-controlled prefetching before we discuss its relative advantages and disadvantages with respect to hardware-controlled schemes.

Software-Controlled Prefetching

In software-controlled prefetching, the analysis of what to prefetch and the scheduling of when to issue prefetches are typically done statically, by software. The compiler (or programmer) inserts special prefetch instructions into the code at points that it deems appropriate. As we saw in Figure 11.19, this may require restructuring the loops in the program to some extent as well. The hardware support needed is the provision of these nonblocking instructions, a cache that allows multiple outstanding accesses, and some mechanism for keeping track of outstanding accesses. The latter two mechanisms are in fact required for all forms of latency tolerance in a system based on reads and writes.

Let us first consider software-controlled prefetching from the viewpoint of a processor trying to hide latency in its reference stream without complications due to interactions with other processors. This problem is equivalent to prefetching in uniprocessors, except with a wider range of latencies. Then we discuss the complications introduced by multiprocessing.

Prefetching with a Single Processor

Consider a simple loop, such as our example from Figure 11.1. A naive approach would be to always issue a prefetch instruction one iteration ahead on array references within loops. This would lead to a software pipeline like the one in Figure 11.19, with two differences: the data is brought into the cache rather than into a prefetch buffer, and the later load of the data will be from the address of the data rather than from the prefetch buffer (i.e., `read(A[i])` and `use(A[i])`). This can easily be extended to approximate just-in-time prefetching by issuing prefetches multiple iterations ahead as discussed earlier (see Exercise 11.15).

To minimize unnecessary prefetches, it is important to analyze and predict the temporal and spatial locality in the program as well as the addresses of future references. For example, blocked matrix factorization reuses the data in the current block many times in the cache, so it does not make sense to prefetch all references to the block. In the software case, unnecessary prefetches have an additional disadvantage beyond cache lookup bandwidth and potentially useless traffic: they introduce useless prefetch instructions in the code, which add execution overhead. The prefetch instructions are often placed within conditional expressions, and with irregular data structures extra instructions are often needed to compute the address to be prefetched, both of which further increase the instruction overhead.

How easy is it to identify which references to prefetch in software? In particular, can a compiler do it, or must it be left to the programmer? The answer depends on the program's reference patterns. References are most predictable when they traverse an array in some regular way. For example, a simple case to predict is when the elements of an array are referenced inside a loop nest, and the array index is an affine function of the loop indices in the loop nest (i.e., a linear combination of loop index values). The following code shows an example:

```

for i ← 1 to n
    for j ← 1 to n
        sum = sum + A[3i+5j+7];
    end for
end for

```

Given the amount of latency we wish to hide, we can try to issue the prefetch that many iterations ahead in the relevant loop. The amount of array data accessed and the spatial locality in the traversal are easy to predict in this example, which makes locality analysis for minimizing unnecessary prefetches easy. The major complication in analyzing data locality is predicting cache misses due to mapping conflicts.

A more difficult class of references to analyze is indirect array references; for example,

```

for i ← 1 to n
    sum = sum + A[index[i]];
end for

```

Whereas we can easily predict the values of i and hence the elements of the index array that will be accessed, we cannot predict the value in $\text{index}[i]$ and hence the elements of A that we shall access. To predict the accesses to A we must first prefetch $\text{index}[i]$ well enough in advance and then use the prefetched value to determine the element of array A to prefetch. The latter requires additional instructions to be inserted. For scheduling, if the number of iterations that we would normally prefetch ahead is k , we should prefetch $\text{index}[i]$ $2k$ iterations ahead so it returns k iterations before we need $A[\text{index}[i]]$, at which point we can use the value of $\text{index}[i]$ to prefetch $A[\text{index}[i]]$. Analyzing temporal and spatial locality in these cases is more difficult than predicting addresses. It is impossible to perform accurate analysis statically since we do not know ahead of time what the spatial relationships among the references to A are nor even how many different locations of A will be accessed (different entries in the index array may have the same value). Our choices are therefore to prefetch all references to $A[\text{index}[i]]$, to prefetch none at all, to obtain profile information about access patterns gathered at run time and use it to make decisions, or to use higher-level programmer knowledge.

Compiler technology has advanced to the point where it can handle the preceding types of array references in loops quite well, within the constraints described. Locality analysis (Wolf and Lam 1991) is used first to predict when array references are expected to miss in a given cache (typically the first-level cache). This results in a prefetch predicate, which can be thought of as a conditional expression inside which a prefetch should be issued for a given iteration. Scheduling based on latency is then used to decide how many iterations ahead to issue the prefetches. Since the compiler may not be able to determine which level of the extended memory hierarchy the miss will be satisfied in, it may be conservative and assume the worst-case latency.

Predicting conflict misses is particularly difficult. Locality analysis, based on fully associative caches, may tell us that a block should still be in the cache so a prefetch

should not be issued, but the block may have been replaced because of conflict misses and therefore would benefit from a prefetch. A possible approach is to assume that a small associativity cache of C bytes effectively behaves like a fully associative cache that is smaller by some factor, but this is not reliable. Multiprogramming also throws off the predictability of misses across process switches since one process might pollute the cache state of another, although the time scales are such that locality analysis often doesn't assume blocks to stay in the cache that long anyway. Despite these problems, limited experiments have shown potential for success with compiler-generated prefetching when most of the cache misses are to affine or indirect array references in loops (Mowry 1994). These include programs on regular grids or dense matrices as well as sparse matrix computations (in which indirect array references are used, but most of the data is often stored in a packed, dense form anyway for efficiency). These experiments are performed through simulation since real machines are only just beginning to provide the underlying hardware support needed for effective software-controlled prefetching.

Accesses that are truly difficult for a compiler to predict are those that involve pointers or linked data structures (such as linked lists and trees). Unlike array indexing, traversing these data structures requires dereferencing pointers along the way; the address contained in the pointer within a list or tree element is not known until that element is reached in the traversal, so it cannot be easily prefetched. Predicting locality for such data structures is also very difficult. Currently, prefetching in these cases must be done by the programmer, exploiting higher-level semantic knowledge of the program and its data structures, as shown in Example 11.3. Compiler analysis for prefetching pointer-based, linked data structures is the subject of research (Luk and Mowry 1996) and will be helped by progress in alias analysis for pointers. In general, limitations of a compiler in other areas (e.g., interprocedural analysis) may limit the effectiveness of its prefetching analysis.

EXAMPLE 11.3 Consider the tree traversal to compute the force on a particle in the Barnes-Hut application described in Chapter 3. The traversal is repeated for each particle assigned to the process, and consecutive particles reuse much of the tree data, which is likely to stay in the cache across particles. How should prefetches be inserted in this tree traversal code? Discuss some possibilities and their trade-offs.

Answer The traversal of the oct-tree proceeds in depth-first manner. However, if it is determined that a tree cell needs to be opened, then all its eight children will be examined as long they are present in the tree. Thus, we can insert prefetches for all the children of a cell as soon as we determine that the cell will be opened (or we can speculatively issue prefetches as soon as we touch a cell and are hence able to dereference the pointers to its children). Since we expect the working set to at least fit in the L₂ cache (which a compiler is highly unlikely to be able to determine), we should prefetch a cell only the first time that we access it (i.e., for the first particle that accesses it), not for subsequent particles. Cache conflicts may occur, which cause unpredictable misses, but there is likely little we can do about that statically. Note that we need to do some work to generate prefetches (determine if this is the first time we are visiting the cell, access and dereference the child pointers, etc.), so the overhead of a prefetch is likely to be several instructions. If the overhead is incurred a lot more often than successful prefetches are generated, it may overcome the benefits of prefetching. Another problem with this scheme is that we

may not be prefetching early enough when memory latencies are high. Since we prefetch all the children at one time, in most cases the depth-first work done for the first child (or two) should be enough to hide the latency of the rest of the children, but this may not be the case. The only way to improve this is to speculatively prefetch multiple levels down the tree when we encounter a cell, dereferencing speculatively prefetched pointers to determine prefetch addresses, hoping that we will indeed touch all the cells we prefetch and they will still be in the cache when we reach them. Since prefetches are nonbinding, correctness is not violated. Other applications that use linked lists in unstructured ways may be even more difficult for a compiler or even a programmer to prefetch successfully. ■

Interactions with a Multiprocessor Coherence Protocol

Two additional issues we must consider when prefetching in parallel programs are prefetching communication misses and prefetching with ownership. Both arise from the fact that other processors might also be accessing and modifying the data that a process references.

In an invalidation-based cache-coherent multiprocessor, data may be removed from a processor's cache—and misses therefore incurred—not only because of replacements but also because of sharing. We should not prefetch data so early that it might be invalidated in the cache before it is used, and we should ideally recognize when data might have been invalidated so that we can prefetch it again before actually using it. Fortunately, nonbinding prefetching makes these performance issues rather than correctness issues.

It is difficult for a compiler to predict incoming invalidations and perform the necessary analysis because the communication in the application cannot be easily deduced from an explicitly parallel program in a shared address space. The one case where the compiler has a good chance is when the compiler itself parallelizes the program. But even then, dynamic task assignment and false sharing of data compromise the success of the analysis.

A programmer has the semantic information about interprocess communication, so it is easier for the programmer to insert and schedule prefetches as necessary in the presence of invalidations. The one kind of information that a compiler does have is that conveyed by explicit synchronization statements in the parallel program. Since synchronization usually implies that data is being shared (for example, in a "properly labeled" program, the modification of data by one process and its use by another process is separated by a labeled synchronization operation), the compiler analysis can assume that communication is taking place and that all the shared data in the cache has been invalidated whenever it sees a synchronization event. Of course, this is conservative and it may lead to unnecessary prefetches, especially when synchronization is frequent and little data is actually invalidated between synchronization events. It would be nice if a synchronization event conveyed some information about which data might be modified, or if this could be efficiently determined, but this is usually not the case (Wood et al. 1993).

As a second enhancement, since a processor often wants to fetch a cache block with exclusive ownership (or simply fetch ownership) in preparation for a write, it makes sense to prefetch in exclusive mode before a write. This can have two benefits

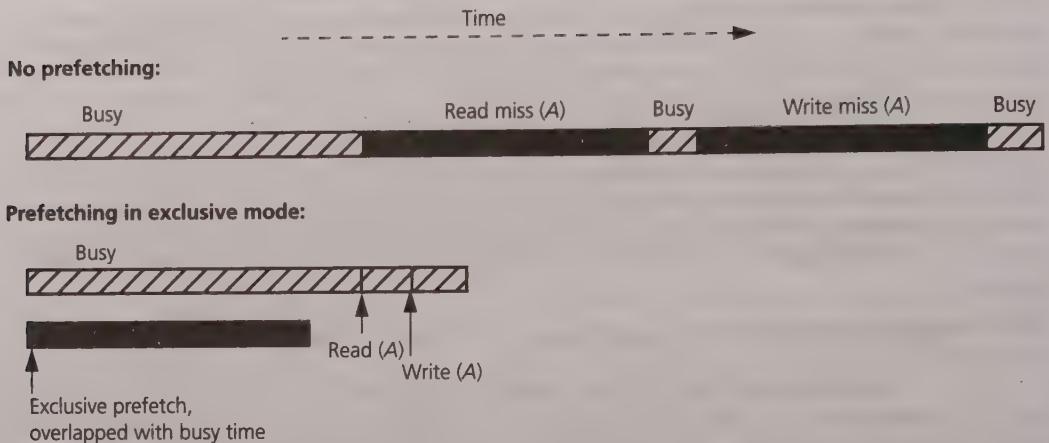


FIGURE 11.20 Benefits from prefetching with ownership. Suppose the latest copy of A is not available locally to begin with but is present in other caches and that a read and then a write are performed. Normal hardware cache coherence would fetch the corresponding block in shared state for the read and then communicate again to obtain ownership upon the write. With prefetching, if we recognize this read-write pattern, we can issue a single prefetch with exclusivity before the read itself and not have to incur any further communication at the write. By the time the write occurs, the block is already present in exclusive state. Prefetching in shared mode before the read hides the read latency but not the write latency since the write will still miss.

when used judiciously. First, it reduces the latency of the actual write operations that follow since the write does not have to invalidate other blocks and wait to obtain exclusive ownership (that was already done by the prefetch). Whether or not this has an impact on performance depends on whether write latency is already hidden by other methods such as by using a relaxed consistency model. The second advantage is in the common case where a process first reads a variable and then shortly thereafter writes it. A single prefetch with ownership even before the read in this case hides both read and write latency. It also halves the traffic, as seen in Figure 11.20, and hence improves the performance of other references as well by reducing contention and bandwidth needs. The quantitative benefits of prefetching in exclusive mode are discussed in (Mowry 1994).

Hardware-Controlled versus Software-Controlled Prefetching

Having seen how hardware-controlled and software-controlled prefetching work, let us consider their relative advantages. The most important advantages of hardware-controlled prefetching are: it does not require any software support from the programmer or compiler; it does not require recompiling code (which may be very important in practice when the source code is not available); and it does not incur instruction overhead or code expansion. On the other hand, its most obvious disadvantages are that it requires substantial hardware support and the prefetching algorithms are hardwired into the machine. However, there are many other trade-offs

having to do with coverage, minimizing unnecessary prefetches, and maximizing effectiveness. Let us summarize these trade-offs, focusing on compiler-generated rather than programmer-generated prefetches in the software case.

- **Coverage.** The hardware and software schemes take very different approaches to analyzing what to prefetch. Software schemes can examine all the data accesses in the code but have only static information, whereas hardware observes a window of dynamic access patterns and predicts future references based on current patterns. Software schemes have greater potential for achieving coverage of complex access patterns but are limited by the analysis, whereas hardware may be limited by the cost of maintaining sophisticated history and the accuracy of necessary techniques like branch prediction. Unlike hardware, the compiler (or even programmer) cannot react to some forms of dynamic information, such as the occurrence of replacements due to unpredicted cache conflicts. Progress is being made in improving the coverage of both approaches in prefetching more types of access patterns (Zhang and Torrellas 1995; Luk and Mowry 1996), but the costs in the hardware case appear high. It is possible to use run-time feedback to improve software prefetching coverage, but there has not been much progress in this direction.
- **Reducing unnecessary prefetches.** Hardware prefetching is driven by increasing coverage and does not perform locality analysis to reduce unnecessary prefetches. It may therefore waste cache access bandwidth, and even interconnect bandwidth, and may replace useful data from the cache. Especially on a bus-based machine, wasting too much interconnect bandwidth on prefetches has at least the potential to saturate the bus and to reduce rather than enhance performance (Tullsen and Eggers 1993).
- **Maximizing effectiveness.** In software prefetching, scheduling is based on prediction. However, it is often difficult to predict how long a prefetch will take to complete, for example, where in the extended memory hierarchy it will be satisfied, and how much contention it will encounter. Hardware can in theory adapt its scheduling at run time since it lets the lookahead PC get only as far ahead as it needs to. However, hiding long latencies becomes difficult because of branch prediction, and every mispredicted branch causes the lookahead PC to be reset, leading to ineffective prefetches until it gets far enough ahead of the PC again. Thus, both the software and hardware schemes have potential problems with effectiveness or just-in-time prefetching.

Hardware prefetching is used in dynamically scheduled microprocessors to prefetch data for operations that are waiting in the reorder buffer but cannot yet be issued. However, in that case, hardware does not have to detect patterns and analyze what to prefetch. While this restricted form of hardware prefetching is becoming popular in microprocessors, so far the on-chip support needed for more general hardware analysis and prefetching of nonunit stride accesses has not been considered worthwhile. On the other hand, microprocessors are increasingly providing prefetching instructions to be used by software (even in uniprocessor systems). Compiler technology for prefetching is progressing as well. Usually, software

prefetching brings data into the first-level cache rather than into a prefetch buffer. This and some other policy issues for prefetching are discussed in Exercise 11.19.

Sender-Initiated Precommunication

In addition to update-based protocols, support for explicit, software-controlled “update,” “deliver,” or “producer prefetch” instructions has been explored. An example is the “poststore” instruction in the KSR1 multiprocessor from Kendall Square Research, which pushes the contents of the whole cache block into the caches that currently contain a (presumably old) copy of the block. A reasonable place to insert these update instructions is at the last write to a shared cache block before a release synchronization operation since it is that data that is likely to be needed by consumers. The destination nodes of the updates are the sharers in the directory entry, just as with update protocols, under the usual assumption that past sharing patterns are a good predictor of future behavior. (Alternatively, the destinations may be specified in software by the instruction itself, or the data may be pushed only to the home main memory rather than other caches, i.e., a write through rather than an update, which hides some but not all of the latency from the destination processors.) These software-based update techniques have some of the same problems as hardware update protocols but to a lesser extent since not every write generates a bus transaction. As with update protocols, competitive hybrid schemes are also possible (Ohara 1996; Grahn and Stenstrom 1996).

Compared to prefetching, the software-controlled sender-initiated communication has the advantage that communication happens just when the data is produced. Also, it reduces traffic for repeating producer-consumer patterns compared to an invalidation-based scheme. However, it has several disadvantages. For one, the data may be communicated too early and may be replaced from the consumer’s cache before use, particularly if it is placed in the primary cache. For another, this scheme precommunicates only communication (coherence) misses, not capacity or conflict misses. In addition, whereas a consumer knows what data it will reference and can issue prefetches for that data, a producer may deliver unnecessary data into processor caches if past sharing patterns are not a perfect predictor of future patterns or may even deliver the same data value multiple times. Further, a prefetch checks the cache and is dropped if the data is found in the cache, reducing unnecessary network traffic; the software update or deliver performs no such checks and can increase traffic and contention, though it reduces traffic when it is successful since it deposits the data in the right places without requiring multiple protocol transactions. Finally, the receiver no longer controls how many precommunicated messages it receives, so buffer overflow may occur. The wisdom gleaned from simulation results so far is that prefetching schemes work better than deliver or update schemes for most applications (Ohara 1996), though the two can complement each other if both are provided (Abdel-Shafi et al. 1997).

Both prefetch and software update schemes can be extended with the capability to transfer larger blocks of data (e.g., multiple cache blocks, a whole object, or an

arbitrarily defined region of addresses) rather than a single cache block. These are called block prefetch and block put mechanisms (block put differs from the block transfer discussed in Section 11.5 in that the data is deposited in the cache and not in main memory). The issues here are similar to those encountered by prefetch and software update instructions, except for differences due to their size. For example, it may not be a good idea to prefetch or deliver a large block to the primary cache.

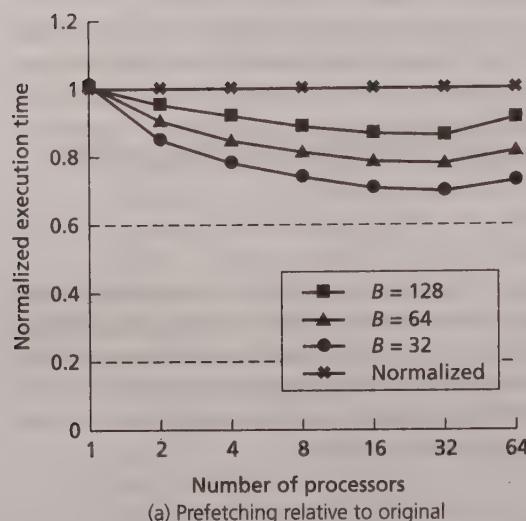
11.6.3 Performance Benefits

Performance results from prefetching so far have mostly been examined through simulation. To illustrate the potential, let us examine results from programmer-inserted software prefetches in some of the example applications used in this book (Woo, Singh, and Hennessy 1994). Programmer-inserted prefetches are used since they can be more aggressive than the best available compiler algorithms. We also consider results from state-of-the-art compiler algorithms.

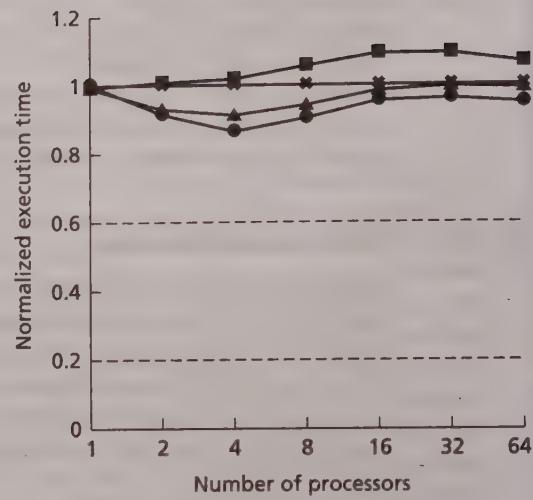
Benefits with Single-Issue, Statically Scheduled Processors

Let us first look at how prefetching performs for the programs and platform presented in Section 11.4.3. To facilitate comparison with block transfer, this experiment focuses on prefetching only remote accesses (cache misses that cause communication). Figure 11.21(a) shows that for a program with predictable access patterns and very good spatial locality like FFT, prefetching remote data helps performance substantially. As with block transfer, the benefits are less for large cache blocks than for small ones since large cache blocks already achieve significant prefetching in themselves. Figure 11.21(b) directly compares the performance of block transfer with that of the prefetched version and shows that the results are quite similar for this program. Prefetching is able to deliver most of the benefits of even the very aggressive block transfer that we assume as long as enough prefetches are allowed to be outstanding at a time. Figure 11.22 shows the same results for the Ocean application. Like block transfer, prefetching helps little here since less time is spent in communication and since not all of the prefetched data is useful (due to poor spatial locality on communicated data along column-oriented partition boundaries).

Prefetching is often much more successful on local accesses. For example, in the iterative nearest-neighbor grid computations in the Ocean application, with barriers between sweeps it is difficult to issue prefetches for boundary elements from a neighbor partition far enough in advance: the new values are produced only a short while before they are needed. However, a process can very easily issue prefetches early enough for grid points within its assigned partition, which are not touched by any other process. Results from a state-of-the-art compiler algorithm show that the compiler can be quite successful in prefetching regular computations on dense arrays, where the access patterns are very predictable (Mowry 1994). These results, shown for two applications in Figure 11.23, include both local and remote accesses for 16-processor executions. Typically, the only problems in these cases are in the

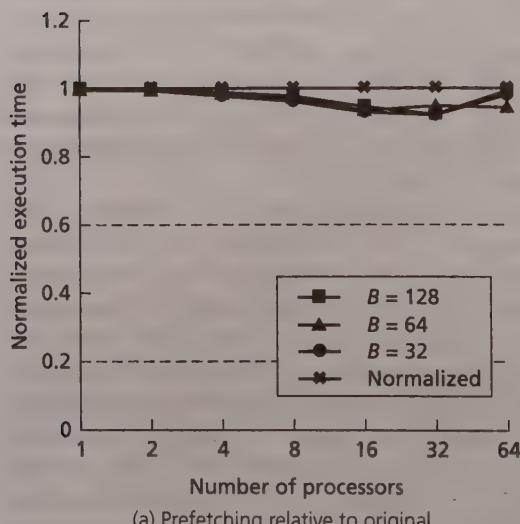


(a) Prefetching relative to original

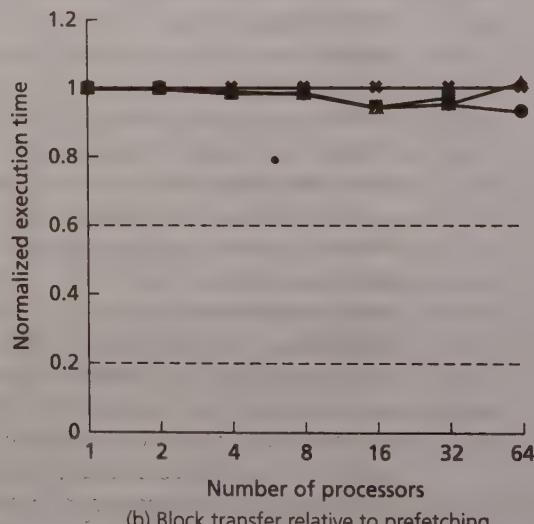


(b) Block transfer relative to prefetching

FIGURE 11.21 Performance benefits of prefetching remote data in a Fast Fourier Transform. (a) shows the performance of the prefetched version relative to that of the original version. The graph can be interpreted just as described in Figure 11.13: each curve shows the execution time of the prefetched version relative to that of the nonprefetched version for the same cache block size (B) for each number of processors. (b) shows the performance of the version with block transfer (but no prefetching), described earlier, relative to the version with prefetching (but no block transfer) rather than relative to the original version. It enables us to compare the benefits from block transfer with those from prefetching remote data. The prefetching experiments allow a total of 16 simultaneous outstanding memory operations, including prefetches, from a processor.



(a) Prefetching relative to original



(b) Block transfer relative to prefetching

FIGURE 11.22 Performance benefits of prefetching remote data in Ocean

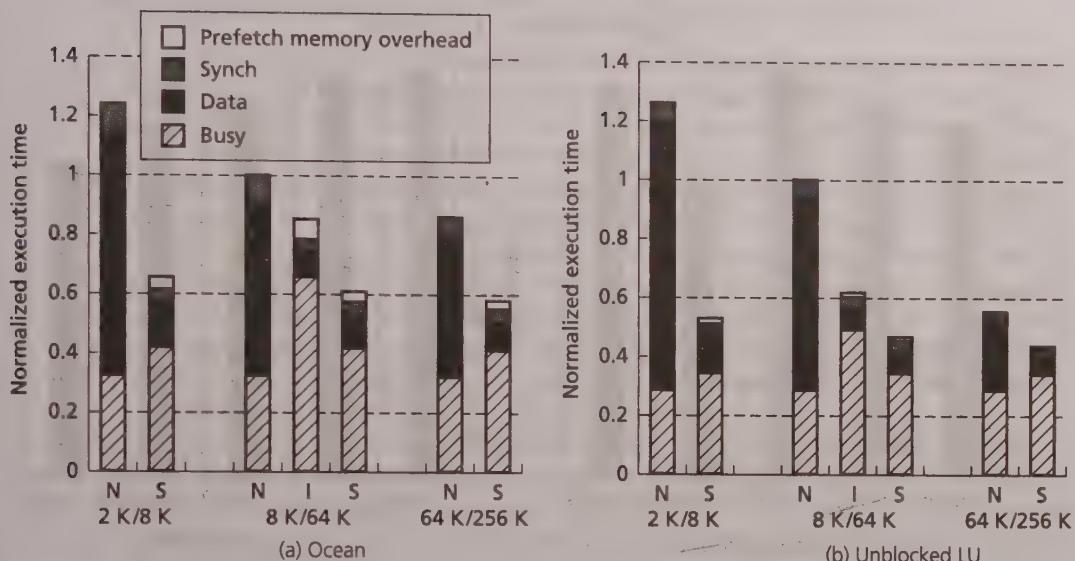


FIGURE 11.23 Performance benefits from compiler-generated prefetching. Results are shown for two parallel programs running on 16-processor simulated machines. The first is an older version of the Ocean simulation program, which partitions in chunks of complete rows and hence both has a higher inherent communication-to-computation ratio and does not have the problem of poor spatial locality at column-oriented boundaries. The second is an unblocked and, hence, lower-performance dense LU factorization. Both local and remote accesses are prefetched, unlike in Figures 11.21 and 11.22. There are three sets of bars for different combinations of L_1/L_2 cache sizes. The bars for each combination are the execution times for no prefetching (N) and selective prefetching (S). For the intermediate combination (8-K L_1 cache and 64-K L_2 cache), results are also shown for the case where prefetches are issued indiscriminately (I), without locality analysis. All execution times are normalized to the time without prefetching for the 8-K/64-K cache size combination. The processor, memory, and communication architecture parameters are chosen to approximate those of the relatively old Stanford DASH multiprocessor and can be found in (Mowry 1994). Latencies on modern systems are much larger relative to processor speed than on DASH. We can see that prefetching helps performance and that the choice of cache sizes makes a substantial difference to the impact of prefetching. The increase in "busy" time with prefetching (especially indiscriminate prefetching) is due to the fact that prefetch instruction overhead is included in busy time. Note that the benefits of prefetching would be much smaller for blocked LU factorization since there would be much less data wait time to hide; since blocked LU factorization is much more popular in practice than unblocked, this raises an important methodological point.

ability to prefetch far enough in advance (e.g., when the misses occur at the beginning of a loop nest or just after a synchronization point) and in the ability to analyze and predict conflict misses.

Some success has also been achieved on sparse array or matrix computations that use indirect addressing, but more irregular, pointer-based applications have not seen much success through compiler-generated prefetching. For example, the compiler algorithm is not successful for the tree traversals in the Barnes-Hut application for the reasons discussed in Example 11.3. Programmers can often do a better job in

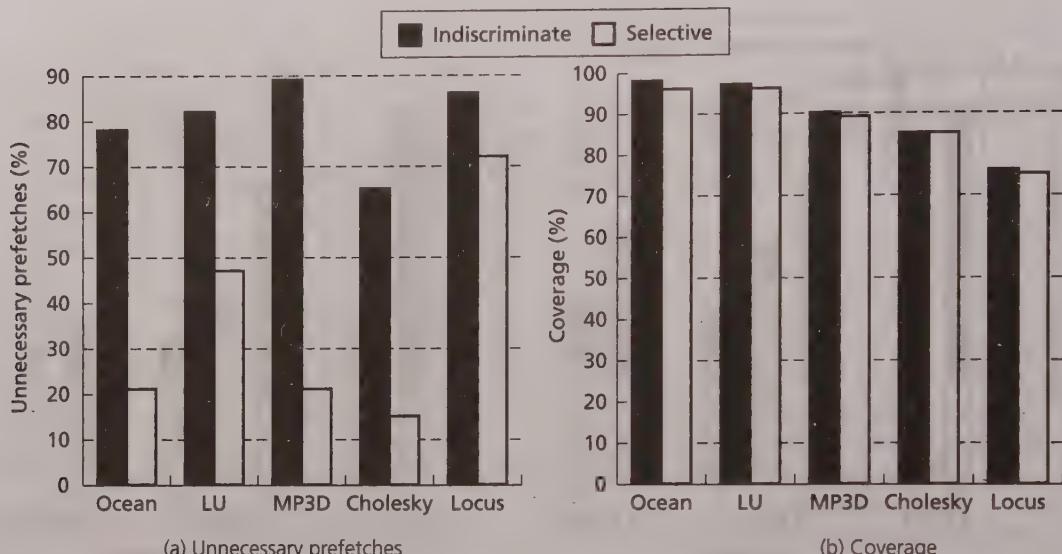


FIGURE 11.24 The benefits of selective prefetching through locality analysis. The fraction of prefetches that are unnecessary is reduced substantially while the coverage is not compromised. MP3D is an application for simulating rarefied hydrodynamics, Cholesky is a sparse matrix factorization kernel, and LocusRoute (abbreviated here as “Locus”) is a wire-routing application from VLSI CAD. MP3D and Cholesky use indirect array accesses, while LocusRoute uses pointers to implement linked lists and therefore makes it more difficult to achieve good coverage.

these cases, as discussed earlier, and profile data gathered at run time may be useful to identify the data accesses that generate the most misses.

For the cases where prefetching is successful overall, locality analysis has been found to substantially reduce the number of prefetches issued without losing much in coverage and hence to perform much better than indiscriminate prefetching of all predictable accesses without locality analysis (see Figure 11.24). Prefetching with exclusive ownership is found to hide write latency substantially in an architecture in which the processor implements sequential consistency by stalling on writes, but it is less important when write latency is already being hidden through a relaxed memory consistency model. It does reduce traffic considerably in any case.

Finally, quantitative evaluations show that as long as caches are reasonably large, cache interference effects due to prefetching are negligible (Mowry 1994; Chen and Baer 1994). They also illustrate that by being more selective, software prefetching indeed tends to induce less unnecessary traffic and fewer cache conflict misses than hardware prefetching. However, the overhead due to extra instructions and associated address calculations can sometimes be substantial in software schemes, especially for applications with irregular access patterns.

Benefits with Multiple-Issue, Dynamically Scheduled Processors

The effectiveness of software-controlled prefetching has been measured (through simulation) on multiple-issue, dynamically scheduled processors (Luk and Mowry 1996; Bennett and Flynn 1996a, 1996b) and compared with its effectiveness on simple, statically scheduled processors (Ranganathan et al. 1997). Despite the latency tolerance already provided by dynamically scheduled processors (including hardware prefetching of operations in the reorder buffer), software-controlled prefetching is found to be effective in further reducing execution time. The percentage reduction in data wait time is somewhat smaller than in statically scheduled processors. However, since data wait time is a greater fraction of execution time (dynamically scheduled superscalar processors reduce instruction processing time much more effectively than they can reduce memory stall time), the percentage improvement in overall execution time due to prefetching is often comparable in the two cases.

Prefetching is less effective in reducing data wait time with dynamically scheduled superscalar processors for two reasons. The increased instruction processing rate means there is less computation time to overlap with prefetches and prefetches often end up being late. Also, dynamically scheduled processors tend to cause more contention for resources that are encountered by a memory operation even before it reaches the L_1 cache (e.g., outstanding request tables, functional units, tracking buffers, etc.). This is because they allow more memory operations to be outstanding at the same time and they do not block on read misses. Prefetching tends to further increase the contention for these resources, thus increasing the latency of non-prefetch accesses. Since this latency occurs before the L_1 cache, it is not hidden effectively by prefetching. Resource contention of this sort is also the reason that simply issuing prefetches earlier does not always solve the late prefetches problem: not only are early prefetches often wasted but they also tie up these processor resources for an even longer time since they tend to keep more prefetches outstanding at a time. The study in (Ranganathan et al. 1997) was unable to improve performance significantly by varying how early prefetches are issued. One advantage of dynamically scheduled superscalar processors compared to single-issue statically scheduled processors from the viewpoint of prefetching is that the instruction overhead of prefetching is usually much smaller since the prefetch instructions can occupy empty slots in the superscalar processor and are hence overlapped with other instructions.

Comparison with Relaxed Memory Consistency

Studies that compare prefetching with relaxed consistency models have found that the two techniques are quite complementary on statically scheduled processors with blocking reads (Gupta et al. 1991). Relaxed models tend to reduce write stall time but do not do much for read stall time, whereas prefetching helps to reduce read stall time. A substantial difference in performance remains between sequential and relaxed consistency even after adding prefetching to both, however, since prefetching is not able to hide write latency as effectively as relaxed consistency can. On

dynamically scheduled processors with nonblocking reads, relaxed models are helpful in reducing read stall time as well as write stall time. Prefetching also helps reduce both, so it is interesting to examine whether starting from sequential consistency performance is helped more by prefetching only or by using only a relaxed consistency model. Even when all optimizations to improve the performance of sequential consistency are applied (like hardware prefetching, speculative reads, and write buffering), it is found to be more advantageous to use a relaxed model without software prefetching than to use a sequentially consistent model with software prefetching on dynamically scheduled processors (Ranganathan et al. 1997). The reason, again, is that although prefetching can help reduce read stall time somewhat better than relaxed consistency, it does not help to hide write latency nearly as well as can be done with relaxed consistency.

11.6.4 Summary

To summarize our discussion of precommunication in a cache-coherent shared address space, the most popular method to date is for microprocessors to provide support for prefetch instructions to be used by software-controlled prefetches, whether inserted by a compiler or a programmer. The same mechanisms are used for either uniprocessor or multiprocessor systems. Prefetching has been found to be quite successful in hiding latency in predictable applications with relatively regular data access patterns, and successful compiler algorithms have been developed for this case. On hardware-coherent, prefetching turns out to be quite successful in competing with block data transfer even in cases where the latter technique works well, even though prefetching involves the processor on every cache block access. (Block transfer is likely to be relatively more successful in systems in which the endpoint overhead per communication is much larger, for example, in software implementations of a shared address space.) However, prefetching irregular computations, particularly those that use pointers heavily, has a long way to go. Programmer-inserted prefetching still tends to outperform compiler-generated prefetching since the programmer has knowledge of access patterns across computations that enable earlier or better scheduling of prefetches. Hardware prefetching is popular only in very limited forms, as in prefetching operations that are in the reorder buffer in dynamically scheduled processors. While hardware prefetching has important advantages in not requiring that programs be recompiled, it is not used for analysis and scheduling in general-purpose prefetching and its future in microprocessors is not clear. Support for sender-initiated precommunication instructions is also not as popular as support for prefetching. Some implementation issues for prefetching are discussed in Exercise 11.18.

11.7

MULTITHREADING IN A SHARED ADDRESS SPACE

Hardware-supported multithreading is perhaps the most versatile technique for hiding latency. It has the following conceptual advantages over other approaches:

- It requires no special software analysis or support (other than having more explicit threads or processes in the parallel program than the number of processors).
- Because it is invoked dynamically, it can handle unpredictable situations, like cache conflicts and communication misses, just as well as predictable ones.
- Whereas the previous techniques are targeted at hiding memory access latency, it can potentially tolerate any long-latency event just as easily, as long as the event can be detected at run time. This includes synchronization and instruction latency.
- Like prefetching, it does not change the memory consistency model since it does not reorder the actual memory operations within a thread.

Despite these potential advantages, multithreading is currently the least popular latency tolerating technique in commercial systems, for two reasons. First, it requires substantial changes to the microprocessor architecture. Second, its utility has so far not been adequately proven for uniprocessor or desktop systems, which constitute the vast majority of the marketplace. We shall see why in the course of this discussion. However, with latencies becoming increasingly longer relative to processor speeds, with more sophisticated microprocessors that already provide mechanisms that can be extended for multithreading, and with new multithreading techniques being developed to combine multithreading with instruction-level parallelism, this trend may change in the future.

Let us begin with the simple form of multithreading that we considered in the context of message passing, in which instructions are executed from one thread until that thread encounters a long-latency event, at which point it is switched out and another thread switched in. The state of a thread is called the *context* of that thread, so multithreading is also called *multiple-context processing*. The state, which must be saved and restored across context switches, includes the processor registers, the program counter, the stack pointer, and some per-process parts of the processor status word (e.g., the condition codes). The cost of a context switch may also involve flushing or squashing instructions already in the processor pipeline, as we shall see. If the latency that we are trying to tolerate is large enough, then we can save the context to memory in software when the thread is switched out and load it back when the thread is switched back in. This is how multithreading is typically orchestrated on message-passing machines, so a standard single-threaded microprocessor can be used in that case. In a hardware-supported shared address space, and even more so on a uniprocessor, the latencies we are trying to tolerate are not that high. The overhead of saving and restoring state in software may be too high to be worthwhile, and we are likely to require hardware support. Let us examine this relationship between switch overhead and latency a little more quantitatively.

Consider processor utilization, that is, the fraction of time that a processor spends executing useful instructions rather than being stalled or incurring overhead. The time a thread spends executing before it encounters a long-latency event is called the *busy* time. The total amount of time spent switching among threads is called the *switching* time. If no other thread is ready, the processor is stalled until one

becomes ready or until the long-latency event it stalled on completes. The total amount of time spent stalled for any reason is called the *idle* time. The utilization of the processor can then be expressed as

$$\text{Utilization} = \frac{\text{Busy}}{\text{Busy} + \text{Switching} + \text{Idle}} \quad (11.2)$$

It is clearly important to keep the switching cost low. Even if we are able to tolerate all the latency through multithreading, thus removing idle time completely, utilization and hence performance are limited by the time spent context switching.

11.7.1 Techniques and Mechanisms

For current microprocessors that issue instructions from only a single thread in a given cycle, hardware-supported multithreading falls broadly into two categories, determined by the decision about *when to switch threads*. The approach assumed so far—in message passing, in multiprogramming to tolerate disk latency, and in this section—has been to let a thread run until it encounters a long-latency event (e.g., a cache miss, a synchronization event, or a high-latency instruction such as a divide) and then switch to another ready thread. This is called the *blocked* approach since a context switch happens only when a thread is blocked or stalled for some reason. Among shared address space systems, this approach is used in the MIT Alewife research prototype (Agarwal et al. 1995). The other major hardware-supported approach is to simply switch threads every processor cycle if possible, whether a long-latency event occurs or not, effectively interleaving the processor resource among a pool of ready threads at a single-cycle granularity. When a thread encounters a long-latency event, it is marked as not being ready and is not available to run until that event completes and the thread joins the ready pool again. This is called the *interleaved* approach. Let us examine both approaches in some detail, looking at their qualitative features and trade-offs as well as their quantitative evaluation and implementation details. After covering both approaches for processors that issue instructions from only a single thread in a cycle, we will examine the integration of multithreading with instruction-level (superscalar) parallelism, which has the potential to overcome the limitations of the traditional approaches (see Section 11.7.5).

Blocked Multithreading

The hardware support for blocked multithreading usually involves maintaining multiple hardware register files and program counters for use by different threads. An *active thread*, or a context, is a thread that is currently assigned one of these hardware copies. The number of active threads may be smaller than the number of *ready threads* (threads that are not stalled but are ready to run) and is limited by the number of hardware copies of the resources. Let us first take a high-level look at the relationship among the latency to be tolerated, the thread-switching overhead, and the

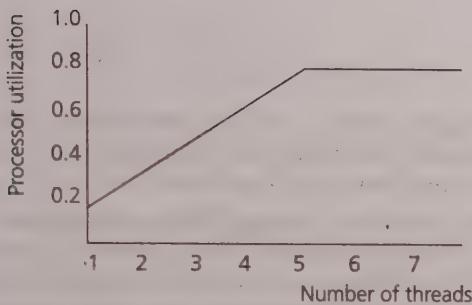


FIGURE 11.25 Processor utilization versus number of threads in blocked multithreading. The figure shows the two regimes of operation: the linear regime and saturation. It assumes $R = 40$, $L = 200$, and $C = 10$ cycles.

number of active threads by using the same type of analysis we used earlier for processor utilization (Culler 1994).

Suppose a processor provides support for N active threads at a time (N -way multithreading). And suppose that each thread operates by repeating the following sequence: execute useful instructions without stalling for R cycles (R , the busy time between stalls, is called the *run length*); encounter a high-latency event and switch to another thread. Now suppose that the latency we are trying to tolerate each time is L cycles and the overhead of a thread or context switch is C cycles. Given a fixed set of values for R , L , and C , a graph of processor utilization versus the number of threads N will look like that shown in Figure 11.25. There are two distinct regimes of operation: the utilization increases linearly with the number of threads up to a threshold, at which point it saturates. Let us see why.

Initially, increasing the number of threads allows more useful work to be done by other threads in the interval L that a thread is stalled, and latency continues to be hidden. Once N is sufficiently large, by the time we cycle through all the other threads—each with its run length of R cycles and switch cost of C cycles—and return to the original thread, we may have tolerated all L cycles of latency. Beyond this, there is no benefit to having more threads since the latency is already hidden. The value of N for which this saturation occurs is given by $(N - 1)R + NC = L$, or

$$N_{sat} = \frac{R + L}{R + C}$$

Beyond this point, the processor is always either busy executing a run from a thread or incurring switch overhead, so the utilization according to Equation 11.2 is

$$u_{sat} = \frac{R}{R + C} = \frac{1}{1 + \frac{C}{R}} \quad (11.3)$$

If N is not large enough relative to L , then the runs of all $N - 1$ other threads will complete before the latency L passes. A processor therefore does useful work for $R + (N - 1)*R$ or NR cycles out of every $R + L$ and is either idle or switching for the rest of the time, leading to a utilization in this linear regime of

$$u_{lin} = \frac{NR}{R+L} = N \cdot \frac{1}{1 + \frac{L}{R}} \quad (11.4)$$

This analysis is clearly simplistic since it uses a fixed average run length of R cycles and ignores the burstiness of misses and other long-latency events. Average-case analysis may lead us to assume that less threads suffice than are actually necessary to handle the bursty situations where latency tolerance may be most crucial. (A more accurate queuing model is given by [Culler 1994].) However, the analysis suffices to make the key points. Since the best utilization we can get with any number of threads, u_{sat} , decreases with increasing switch cost C , it is very important that we keep switch cost low. Switch cost also affects the types of latency that we can hide; for example, pipeline latencies or the latencies of misses that are satisfied in the second-level cache may be difficult to hide unless the switch cost is very low.

Switch cost can be kept low if we provide hardware support for several active threads, including separate register files, PCs, and so on. We can simply switch from one hardware state to another upon a context switch without saving and restoring state in software, which is the approach taken in most hardware multithreading proposals. Typically, a large register file is either statically divided into as many equally sized register frames as the active threads support (called a *segmented register file*), or the register file is dynamically managed as a cache that holds the registers of active contexts.

Although replicating context state in hardware can bring the cost of this aspect of switching among active threads down to a single cycle (it's like changing a pointer instead of copying a large data structure), there is another time cost for context switching that we have not discussed so far. This cost arises from the use of pipelining in instruction execution.

When a long-latency event occurs, we want to switch out the current thread. Suppose the long-latency event is a cache miss. The cache access is made only in the data fetch stage of the instruction pipeline, which is quite late in the pipeline. Typically, the hit/miss result is only known in the write-back stage, which is at the end of the pipeline. This means that by the time we know that a cache miss has occurred and the thread should be switched out, several other instructions from that thread (potentially k instructions, where k is the pipeline depth) have already been fetched and are in the pipeline (see Figure 11.26). We are faced with three possibilities: (1) allow these subsequent instructions to complete, but start to fetch instructions from the new thread at the same time; (2) allow the instructions to complete before starting to fetch instructions from the new thread; or (3) squash the instructions from the pipeline and then start fetching from the new thread.

The first case is complex to implement for two reasons. First, since instructions from different threads will be in the pipeline at the same time, the standard uniprocessor pipeline registers and dependence resolution mechanisms (interlocks) must be modified. Every instruction must be tagged with its context as it proceeds through the pipeline, and/or multiple sets of pipeline registers may be used to distinguish results from different contexts. In addition to the increase in area, the use of multiple pipeline registers at each pipeline stage means that the registers must be

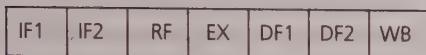


FIGURE 11.26 Impact of late miss detection in a pipeline. Thread A is the current thread running on the processor. A cache miss occurring on instruction A_i from this thread is only detected after the second data fetch stage (DF2) of the pipeline (i.e., in the write-back [WB] cycle of A_i 's traversal through the pipeline). At this point, the following six instructions from thread A (A_{i+1} through A_{i+6}) are already in the different stages of the assumed seven-stage pipeline (two cycles of instruction fetch [IF], one cycle of register fetch [RF], one cycle of execute [EX], followed by two cycles of data fetch and the write back). If all the instructions are squashed when the miss is detected (the crossed-out slots in the lower drawing), we lose at least seven cycles of work.

multiplexed onto the latches for the next stage, which may increase the processor cycle time. Since part of the motivation for the blocked scheme is its design simplicity and its ability to use commodity processors with as little design effort and modification as possible, this may not be a very appropriate choice. The second problem with this choice is that the instructions already in the pipeline from the thread that incurred the cache miss may stall the pipeline because they may depend on the data returned by the miss.

The second choice avoids having instructions from multiple threads simultaneously in the pipeline but still must contend with stalls due to dependent instruction. The third choice avoids both problems and is simple to implement since the standard uniprocessor pipeline suffices and already has the ability to squash instructions. It is the favored choice for the blocked scheme, even though it does cause a number of cycles equal to the pipeline depth to be wasted on a switch.

How does a context switch get triggered in the blocked approach to hide the different kinds of latency? On a cache miss, the switch can be triggered by the detection of the miss in hardware. For synchronization latency, we can simply ensure that an explicit context switch instruction follows every synchronization event that is expected to incur latency (or even all synchronization events). Since the synchronization event may need to be satisfied by another thread that runs on the same processor, an explicit switch is necessary to avoid deadlock without waiting for timeouts. Long pipeline stalls can also be handled by inserting a switch instruction following a long-latency instruction such as a divide. Finally, short pipeline stalls like data hazards are likely to be very difficult to hide with the blocked approach.

To summarize, the blocked approach has a relatively low implementation cost (as we shall see in more detail later) and good single-thread performance (if only a single thread runs on a processor, there are no context switches and this scheme performs just like a standard uniprocessor would). The disadvantage is that the context switch overhead is high: approximately the depth of the pipeline, even when registers and other processor state do not have to be saved to or restored from memory.

This overhead limits the types of latencies that can be hidden as well as the effectiveness. Example 11.4, taken from (Laudon, Gupta, and Horowitz 1994), examines the performance impact.

EXAMPLE 11.4 Suppose four threads, A, B, C, and D, run on a processor. The threads have the following activity:

A issues two instructions, with the second instruction incurring a cache miss, then issues four more.

B issues one instruction, followed by a two-cycle pipeline dependence, followed by two more instructions, the last of which incurs a cache miss, followed by two more.

C issues four instructions, with the fourth instruction incurring a cache miss, followed by three more.

D issues six instructions, with the sixth instruction causing a cache miss, followed by one more.

Show how successive pipeline slots are either occupied by threads or wasted in a blocked multithreaded execution. Assume a simple four-stage pipeline, and hence a four-cycle context switch time, and a cache miss latency of 10 cycles (small numbers are used here for ease of illustration).

Answer The solution is shown in Figure 11.27, assuming that threads are chosen round-robin starting from thread A. We can see that while most of the memory latency is hidden, this is at the cost of context switch overhead. Assuming the pipeline is in steady state at the beginning of this sequence, we can count cycles starting from the time the first instruction reaches the WB stage (i.e., the first cycle shown for the multithreaded execution in the bottom part of the figure). Of the 51 cycles taken in the multithreaded execution, 21 are useful busy cycles, 2 are pipeline stalls, no idle cycles are stalled on memory, and 28 are context switch cycles, leading to a processor utilization of $(21/51) * 100$, or only 41%, despite the extremely low cache miss penalty assumed. ■

Interleaved Multithreading

In the interleaved approach, in every processor clock cycle a new instruction is chosen from a different thread that is ready and active (i.e., assigned a hardware context) so that threads are switched every cycle rather than only on long-latency events. When a thread incurs a long-latency event, it is simply disabled or removed from the pool of ready threads until that event completes and the thread is labeled ready again (it is still active in that it retains its hardware state resources). Segmented or replicated register files are used here as well to avoid the need to save and restore registers. The key advantage of the interleaved scheme is that there is no context switch overhead. No event needs to be detected in order to trigger a context switch since this is done every cycle, and with enough threads, instructions from the same thread will not be in the pipeline at the same time, so there is no need to squash instructions. Thus, if there are enough concurrent threads, in the best case all latency will be hidden without any switch cost, and the processor will perform useful work in every cycle. An example of this ideal scenario is shown in Figure 11.28, where we assume six active threads for illustration. The typical disadvantage

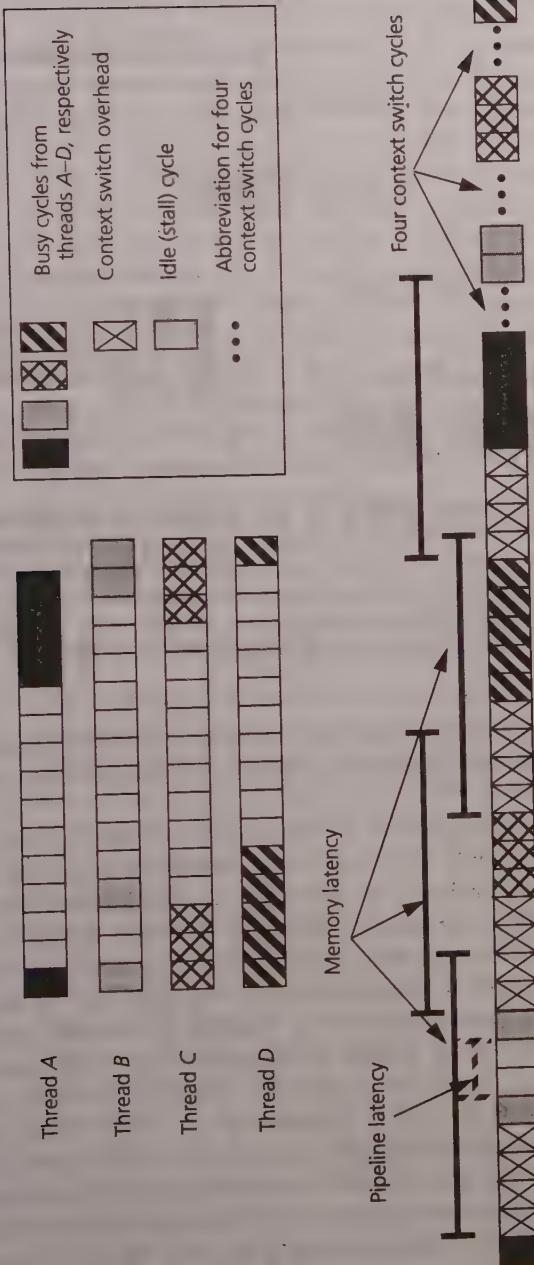


FIGURE 11.27 Latency tolerance in the blocked multithreading approach. The instruction shown in each slot is the one that is (or would be) in the last (WB) stage of the pipeline in that cycle. The top part of the figure shows how the four active threads on a processor would behave if each was the only thread running on the processor. For example, the first thread issues one instruction (whose WB cycle is the first cycle shown for thread A at the top of the figure), then issues a second one that incurs a cache miss (so the second cycle shown for thread A, which would have been the WB cycle of that instruction, is empty, as are several following), then issues some more. The bottom part shows how the processor switches among threads when they incur a cache miss. If the second instruction from thread A had not missed, it would have been in the WB state in the second cycle shown at the bottom. However, since it misses, that cycle is wasted and the three other instructions from thread A that have already been fetched into the four-deep pipeline have to be squashed in order to switch contexts. Note that the two-cycle pipeline latency does not generate a thread switch since the switch overhead of four cycles is larger than this latency.

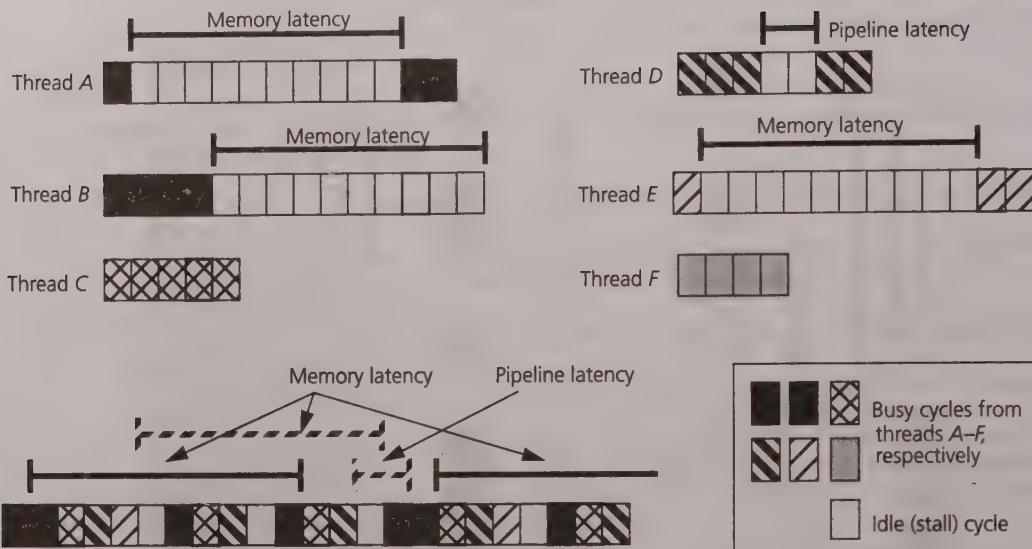


FIGURE 11.28 Latency tolerance in an ideal setting in the interleaved multithreading approach. The top part of the figure shows how the six active threads on a processor would behave if each was the only thread running on the processor. The bottom part shows how the processor switches among ready threads round-robin every cycle, leaving out those threads whose last instruction caused a stall that has not been satisfied yet. For example, thread A (solid) is not chosen again until its high-latency memory reference is satisfied and its turn comes again in the round-robin scheme.

of the interleaved approach is the higher hardware cost and complexity, though the specifics of this and other potential disadvantages depend on the particular type of interleaved approach used.

Interleaved schemes have undergone a fair amount of evolution. The early schemes severely restricted the number and type of instructions from a given thread that could be in the processor's pipeline at a time. This reduced the need for hardware interlocks and squashing instructions. It simplified processor design but had severe implications for the performance of single-threaded programs. More recent interleaved schemes greatly reduce these restrictions, as we shall see. In practice, another distinguishing feature among interleaved schemes is whether they use caches to reduce latency before trying to hide it. Machines built so far using the interleaved technique do not use caches at all but rely completely on latency tolerance through multithreading (Smith 1981; Alverson et al. 1990). More recent research proposals advocate the use of interleaved techniques and full pipeline interlocks with caches. (Laudon, Gupta, and Horowitz 1994). (Recall that blocked multithreaded systems use caching since they want to keep a thread running efficiently for as long as possible.) Let us look at some interesting stages in the development of interleaved schemes.

The Basic Interleaved Scheme

The first interleaved multithreading scheme was used in the Denelcor HEP (heterogeneous element processor) multiprocessor, developed between 1978 and 1985 (Smith 1981). Each processor had up to 128 active contexts, 64 user-level and 64 privileged, though only about 50 were actually available to the user. The large number of active contexts was needed even though the memory latency was small—about 20–40 cycles without contention—since the machine had no caches, and the memory latency was incurred on every memory reference. (Memory modules were all on the other side of a multistage interconnection network, but the processor had an additional direct connection to one of these modules that it could consider its “local” module). The 128 active contexts were supported by replicating the register file and other critical state 128 times. The pipeline on the HEP was 8 cycles deep. It supported interlocks among nonmemory instructions but did not allow more than one memory, branch, or divide operation to be in the pipeline at a given time. This meant that several threads had to be active on each processor at a time to utilize the pipeline effectively, even without any memory or other stalls. The absence of caches and the need to hide memory latency further increased the number of threads needed. This meant that the degree of explicit concurrency in a program had to be much larger than the number of processors, restricting the range of applications that would perform well..

Better Use of the Pipeline

The drawbacks of allowing only a single memory operation from a thread in the pipeline at a time are poor single-thread performance and the very large number of threads needed. Systems descended from the HEP have therefore alleviated this restriction. These systems include the Horizon (Kuehn and Smith 1988) and the more recent Tera (Alverson et al. 1990) multiprocessors. They still do not use caches to reduce latency, relying completely on latency tolerance for all memory references.

The first of these designs, the Horizon, was never actually built. Unlike HEP, the design allows multiple memory operations from a thread to be in the pipeline simultaneously. Yet it does not provide hardware pipeline interlocks even for nonmemory instructions. Rather, the analysis of dependences is left to the compiler. The idea is quite simple. Based on compiler analysis, every instruction is tagged with a three-bit “lookahead” field, which specifies the number of immediately following instructions in that thread that are sure to be independent of that instruction. Suppose the lookahead field for an instruction has the value five. This means that the next five instructions (memory or otherwise) are independent of the current instruction, and so can be in the pipeline with the current instruction even though there are no hardware interlocks to resolve dependences. Thus, if a long-latency event is encountered by that instruction, the thread does not immediately become unready but can issue five more instructions before it becomes unready. The maximum value of the lookahead field is seven: the machine will prohibit more than seven instructions from the current thread from entering the pipeline until the current one leaves. The small

number of lookahead bits provided is influenced by the high premium on bits in the instruction word, by the ability of the compiler to utilize more lookahead, and particularly by the register pressure introduced by having three results per instruction times the number of lookahead instructions; more lookahead and greater register pressure might have been counterproductive for instruction scheduling.

Each “instruction” or cycle in Horizon can include up to three operations, one of which may be a memory operation. This means that 21 independent operations must be found to achieve the maximum lookahead, so sophisticated instruction scheduling by the compiler is clearly very important. For a typical program it is likely that a memory operation is issued every instruction or two. Since the maximum lookahead size is larger than the average distance between memory operations, it is very useful to allow multiple memory operations at a time in the pipeline. However, with the absence of caches every memory operation is a long-latency event, so a large number of ready threads is still needed to hide latency. In particular, single-thread performance—the performance of programs that are not multithreaded—is not helped much by a small amount of lookahead without caches and may still be quite limited.

The Tera architecture, built by Tera Computer Company, is the latest in the series of interleaved multithreaded architectures that do not use caches. Tera manages instruction dependences differently than Horizon and HEP, using a combination of the approaches. It provides hardware interlocks for instructions that do not access memory (like HEP) and Horizon-like lookahead for memory instructions.

The Tera machine separates operations into memory operations, arithmetic (or logical) operations, and control operations (e.g., branches). The unusual, custom-designed processor can issue three operations per instruction, much like Horizon, either one from each category or two arithmetic operations and one memory operation. Arithmetic and control operations go into one pipeline, which has hardware interlocks to allow multiple operations from the same thread. The pipeline is very deep, and there is a sizable minimum issue delay between consecutive instructions from a thread even if there are no dependences between them (about 16 cycles, with the pipeline being deeper than this). Thus, while more than one instruction from a thread can be in this pipeline at the same time, several interleaved threads (about 16) are required to hide even the instruction latency. Even without memory references, a single thread would at best complete one instruction every 16 cycles.

Memory operations pose a bigger problem. Although Tera uses very aggressive memory and network technology, the average latency of a memory reference without contention is about 70 cycles (a processor cycle is 2.8 ns). Tera therefore uses compiler-generated lookahead fields for memory operations, with a slightly different semantics than in Horizon. Every instruction that includes a memory operation (called a memory instruction) has a 3-bit lookahead field that tells how many immediately following instructions (memory or otherwise) are independent of that memory operation. Those following instructions do not have to be independent of one another, just of that memory operation. The thread can then issue that many instructions past the memory instruction before it has to render itself unready. This change in the use of lookahead makes it easier for the compiler to schedule instruc-

tions to have larger lookahead values and eases register pressure as well. Example 11.5 makes this concrete.

EXAMPLE 11.5 Suppose that the minimum issue delay between consecutive instructions from a thread in Tera is 16 and the average memory latency to hide is 70 cycles. What is the smallest number of threads that would be needed to hide latency completely in the best case, and how much lookahead would we need per memory instruction in this case?

Answer A minimum issue delay of 16 means that we need about 16 threads to keep the pipeline full without considering memory. Since memory operations are almost one per instruction in each thread, 16 threads can suffice to hide 70 cycles of memory latency if each thread issues about four independent instructions before it is made unready after a memory operation, that is, if a lookahead of about 4 can be sustained. Since latencies are in fact often larger than the average uncontended 70 cycles, a higher lookahead of at most 7 (3 bits) is provided. Even longer latencies would ask for larger lookahead values and sophisticated compilers (or more threads). So would a desire for fewer threads, though this would require reducing the minimum issue delay in the nonmemory pipeline as well. ■

While it may seem from the above that supporting a few tens of active threads should be enough, Tera, like HEP, supports 128 active threads in hardware. For this, it replicates all processor state (program counter, processor status word, and registers) 128 times, resulting in a total of 4,096 64-bit general registers (32 per thread) and 1,024 branch target registers (8 per thread) in the processor. The large number of threads is supported for several reasons and reflects the fundamental reliance of the machine on latency tolerance rather than reduction. First, some instructions may not have much lookahead at all, particularly read instructions; with three operations per instruction, a lookahead value of four instructions implies that there must be 12 independent operations between the read and the dependent use. Second, with most memory references going into the network, they may incur contention, in which case the latency to be tolerated may be much longer than 70 cycles. Third, the goal is not only to hide instruction and memory latency but also to tolerate synchronization wait time, which is caused by load imbalance or contention for critical resources and is usually much larger than data access latency.

The designers of the Tera system and its predecessors take a much more radical view to the redesign of the multiprocessor building block than advocated by the other latency-tolerating techniques. Unlike the approach we have followed so far—reduce latency first, then hide the rest—this approach does not pay much attention to reducing latency at all; the processor is redesigned with a primary focus on tolerating the latency of fine-grained accesses through interleaved multithreading. The argument is that the commodity microprocessor building block, with its reliance on caches and support for only one or a small number of hardware contexts, is inappropriate for general-purpose multiprocessing. Because of the high latencies and physically distributed memory in modern convergence architectures, the use of relatively “latency-intolerant” commodity microprocessors as the building block implies that much attention must be paid to data locality in both the caches and in data distribution at the main memory level. This makes the task of programming for performance

too complicated, especially since compilers have not yet succeeded in managing locality automatically, in any but the simplest cases, and their potential is unclear. The Tera approach argues that the only way to make multiprocessing truly general purpose is to take this burden of locality management off software and place it on the architecture in the form of much greater latency tolerance support in the processor. If enough extra threads can be found and this technique is successful, the programmer's view of the machine can indeed be a PRAM (i.e., the cost of data access can be ignored), and the programmer can concentrate on concurrency rather than latency management. Of course, this approach sacrifices the tremendous leverage obtained by using commodity microprocessors and caches and faces head-on the challenge of the enormous effort that must be invested in the design of a nonstandard high-performance processor and the associated system software. It is also likely to result in poor single-thread performance, which means that even uniprocessor applications must be heavily multithreaded (or the system very heavily multiprogrammed) to achieve good performance.

Full Single-Thread Pipeline Support and Caching

While the interleaved approach described so far is very different than the blocked multithreading approach, both have several limitations. The Tera interleaved approach improves the basic HEP approach, but still requires many concurrent threads for good utilization. Not using caches implies that every memory operation is a long-latency operation. In addition to increasing the number of threads needed and the difficulty of hiding the latency, this means that every memory reference consumes memory and perhaps communication bandwidth, so the machine must provide tremendous bandwidth as well.

The blocked multithreading approach, on the other hand, requires less modification to a commodity microprocessor. It utilizes caches and does not switch threads on cache hits, thus providing good single-thread performance and requiring a smaller number of threads. However, it has high context switch overhead and cannot hide short latencies. The high switch overhead also makes it less suited to tolerating the not-so-large latencies on uniprocessors. It is therefore difficult to justify either of these schemes for uniprocessors and hence for the high-volume marketplace.

It is possible to use an interleaved approach with both caching and full single-thread pipeline support, thus requiring a smaller number of threads to hide memory latency, incurring lower context switch overhead than a blocked scheme and providing better support for uniprocessors. One such proposal has been studied in detail (Laudon, Gupta, and Horowitz 1994). From the HEP and Tera approaches, this interleaved approach takes the idea of maintaining a set of active threads, each with its own set of registers and status words, and having the processor select an instruction from one of the ready threads every cycle. The selection may be simple, such as round-robin among the ready threads. A thread that incurs a long-latency event makes itself unready until the event completes, as before. A key difference is that the pipeline is a standard microprocessor pipeline and has full bypassing and forwarding

support so that instructions from the same thread can be issued in consecutive cycles (as in a blocked scheme); there is no minimum issue delay as in Tera. In the best case, a k -deep pipeline may contain k instructions from the same thread. In addition, the use of caches to reduce latency implies that most memory operations are not long-latency events; a given thread is therefore ready a larger fraction of the time, and the number of threads needed to hide latency is kept small. For example, if each thread incurs a cache miss every 30 cycles, and a miss takes 120 cycles to complete, then only five threads (the one that misses and four others) are needed to achieve full processor utilization.

The overhead in this interleaved scheme arises from the same source as in the blocked scheme. A cache miss, which renders a thread unready, is detected late in the pipeline; if there are only a few interleaved threads, then the thread that incurs the miss may have fetched other instructions into the pipeline by this time. Unlike in the Tera, where the compiler guarantees through lookahead that such subsequent instructions in the pipeline are independent of the memory instruction, here we must do something about these instructions. For the same reason as in the blocked scheme, the proposed approach chooses to squash these instructions, that is, to mark them as not being allowed to modify any processor state. The key difference with the blocked scheme is that, because instructions from other threads are interleaved cycle by cycle in the pipeline, not all instructions need to be squashed—only those from the thread that incurred the miss. The cost of making a thread unready is therefore typically much smaller than that of a context switch in the blocked scheme, where all instructions in the pipeline must be squashed. In fact, if enough ready and active threads are available, which requires a larger degree of hardware state replication than is advocated by this approach, other instructions from the thread that misses may not be in the pipeline at all, and no instructions will need to be squashed (as in the HEP/Tera approaches). The comparison with the blocked approach is shown in Figure 11.29.

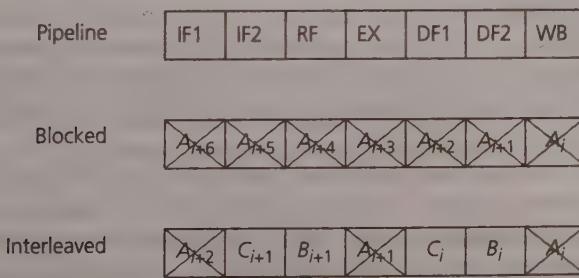


FIGURE 11.29 The cost of making a thread unready in the interleaved scheme with full single-thread support compared with the context switch cost in the blocked scheme. The figure first shows the assumed seven-stage pipeline, followed by the impact of late miss detection in the blocked scheme in which all instructions in the pipeline are from thread A and have to be squashed, followed by the situation for the interleaved scheme. In the interleaved scheme, instructions from three different threads are in the pipeline, and only those from thread A need to be squashed. The switch cost, or overhead incurred on a miss, is three cycles rather than seven in this case.

The result of the lower cost of making a thread unready is that shorter latencies, such as local memory access or long instruction latencies, can be tolerated more easily than in the blocked case, making this interleaved scheme more appropriate for multithreading on uniprocessors as well. Very short latencies that cannot be hidden by the blocked scheme, such as those of short pipeline hazards, are usually hidden naturally by the cycle-by-cycle interleaving of threads without even needing to make a thread unready. The effect of the differences on the simple four-thread, four-deep pipeline example used for the blocked scheme in Figure 11.27 is illustrated in Figure 11.30. In this simple example, assuming the pipeline is in steady state at the beginning of this sequence, the processor utilization is 21 cycles out of the 30 cycles taken in all, or 70% (compared to 21 out of 51 cycles or 41% for the blocked scheme example in Figure 11.27). While this example is contrived and uses unrealistic parameters for ease of graphical illustration, the fact remains that on modern superscalar processors that issue a memory operation in almost every cycle, the context switch overhead of switching on every cache miss in a blocked scheme may become quite expensive. The disadvantage of this scheme compared to the blocked scheme is greater implementation complexity.

The blocked scheme and this last interleaved scheme (henceforth called “the interleaved scheme”) start with simple, commodity processors with full pipeline interlocks and caches and modify them to make them multithreaded. As stated earlier, even if they are used with superscalar processors, they only issue instructions from within a single thread in a given cycle. A more sophisticated multithreading approach exists for superscalar processors, but for simplicity let us first examine the performance and implementation issues for these two more directly comparable approaches.

11.7.2 Performance Benefits

Simulation studies have shown that both the blocked scheme and the interleaved scheme (with full pipeline interlocks and caching) can hide read and write latency quite effectively (Laudon, Gupta, and Horowitz 1994; Kurihara, Chaiken, and Agarwal 1991). The number of active contexts needed is found to be quite small, usually in the vicinity of four to eight, although this may change as the latencies become longer relative to processor cycle time.

Let us examine some of the simulation results for parallel programs (Laudon 1994). The architectural model is again a cache-coherent multiprocessor with 16 processors using a flat, memory-based directory protocol. The processor model is single issue and modeled after the MIPS R4000 for the integer pipeline and the DEC Alpha 21064 for the floating-point pipeline. The cache hierarchy used is a small (64-KB) single-level cache, and the latencies for different types of accesses are modeled after the Stanford DASH multiprocessor prototype (Lenoski et al. 1992). Overall, both the blocked and interleaved schemes were found to improve performance substantially. Of the seven applications studied, the speedup from multithreading ranged from 2.0 to nearly 3.5 for three applications, from 1.2 to 1.6 for three others, and was negligible for the last application because it had very little extra parallelism.

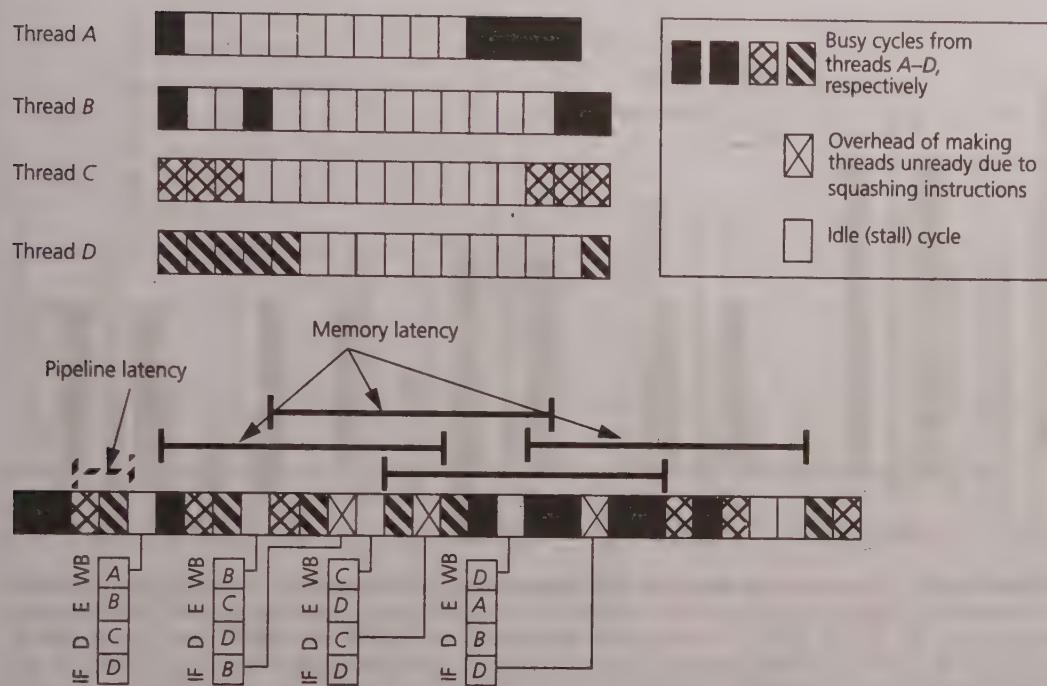


FIGURE 11.30 Latency tolerance in the interleaved scheme. A four-stage pipeline is assumed, the stages being instruction fetch (IF), decode (D), execute (E), and write back (WB). The top part of the figure shows how the four active threads on a processor would behave if each was the only thread running on the processor. The bottom part shows how the processor switches among threads. As in Figure 11.27, the instruction in a slot (cycle) is the one that retires (or would retire) from the pipeline in that cycle. In the first four cycles shown, an instruction from each thread retires. In the fifth cycle, A would have retired its second instruction but discovers that it has missed and needs to become unready, so that slot is an idle slot. The three other instructions in the pipeline at that time (shown below the idle slot) are from the three other threads, so there is no switch cost except for the one cycle due to the instruction that missed. When B's next instruction reaches the WB stage (the ninth cycle), it detects a miss and has to become unready. At this time, since A is already unready, an instruction each from C and D have entered the pipeline, as has one more from B (this one is now in its IF stage, as shown, and would have retired in the twelfth cycle). Thus, the instruction from B that misses wastes a cycle, and one instruction from B has to be squashed. Similarly, C's instruction that would have retired in the thirteenth cycle misses and causes another instruction from C to be squashed, and so on.

to begin with (see Figure 11.31). The interleaved scheme was found to always outperform the blocked scheme, as expected from the preceding discussion, with a geometric mean speedup over all applications of 2.75 compared to 1.9.

The advantages of the interleaved scheme are found to be greatest for applications that incur a lot of latency due to short pipeline stalls (such as those for result dependences in floating-point add, subtract, and multiply instructions) since this latency cannot be hidden by the blocked scheme and is hidden with no overhead by the interleaved scheme. Longer pipeline latencies, such as the tens of cycle latencies of

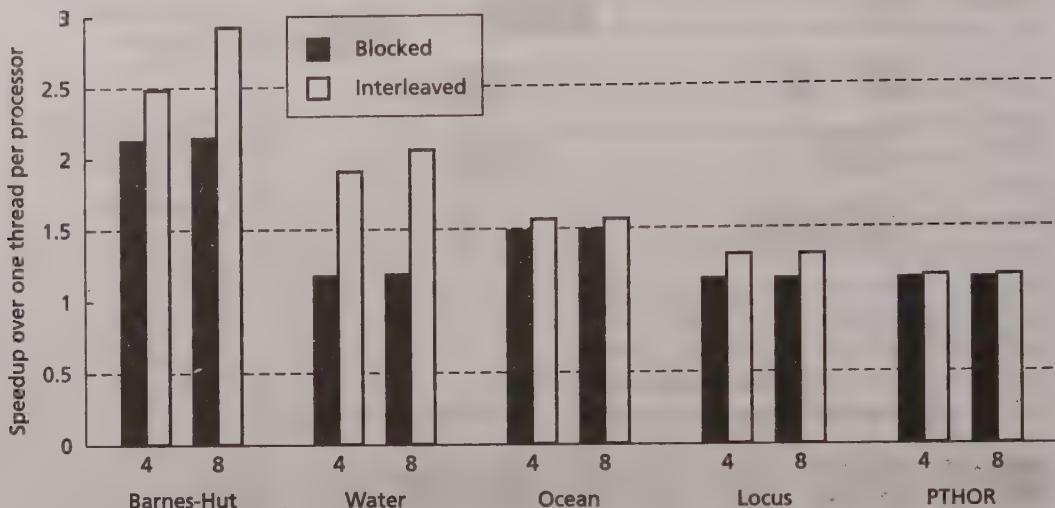


FIGURE 11.31 Speedups for blocked and interleaved multithreading. The bars show speedups for different numbers of contexts (4 and 8) relative to a single-context-per-processor execution for seven applications. All results are for 16 processor executions. The Locus application was introduced in Figure 11.24. Water is a molecular dynamics simulation of water molecules in liquid state. PTHOR is a parallel event-driven simulator of logic circuits whose concurrency profile was shown in Figure 2.5. The multiprocessor simulation model assumes a single-level, 64-KB, direct-mapped write-back cache per processor. The memory system latencies assumed are 1 cycle for a hit in the cache, 24–45 cycles with a uniform distribution for a miss satisfied in local memory, 75–135 cycles for a miss satisfied at a remote home, and 96–156 cycles for a miss satisfied in a dirty node that is not the home. These latencies are clearly low by modern standards. The MIPS R4000-like integer unit has a 7-cycle pipeline, and the floating-point unit has a 9-stage pipeline (5 execute stages). The divide instruction has a 61-cycle latency, and unlike other functional units the divide unit is not pipelined. Both schemes switch threads (or make the thread unready) on a divide instruction. The blocked scheme uses an explicit switch instruction on synchronization events and divides, which has a cost of 3 cycles (less than a full 7-cycle context switch because the decision to switch is known after the decode stage rather than at the write-back stage). The interleaved scheme uses a backoff instruction (discussed in Section 11.7.4) in these cases, which has a cost of 1–3 cycles depending on how many instructions need to be squashed as a result.

divide operations, can be tolerated quite well by both schemes, though the interleaved scheme still performs better because of its lower switch cost. The advantages of the interleaved scheme are found to be retained even when the organizational and performance parameters of the extended memory hierarchy are changed (for example, longer latencies and multilevel caches). They are, likely to be even greater with modern processors that issue multiple operations per cycle since the frequency of cache misses and, hence, context switches is likely to increase. A potential disadvantage of both types of multithreading is that multiple threads of execution share the same cache, TLB, and branch prediction unit, raising the possibility of negative interference between them (e.g., mapping conflicts across threads in a low-associativity cache); however, these negative effects have been found to be quite small in published studies.

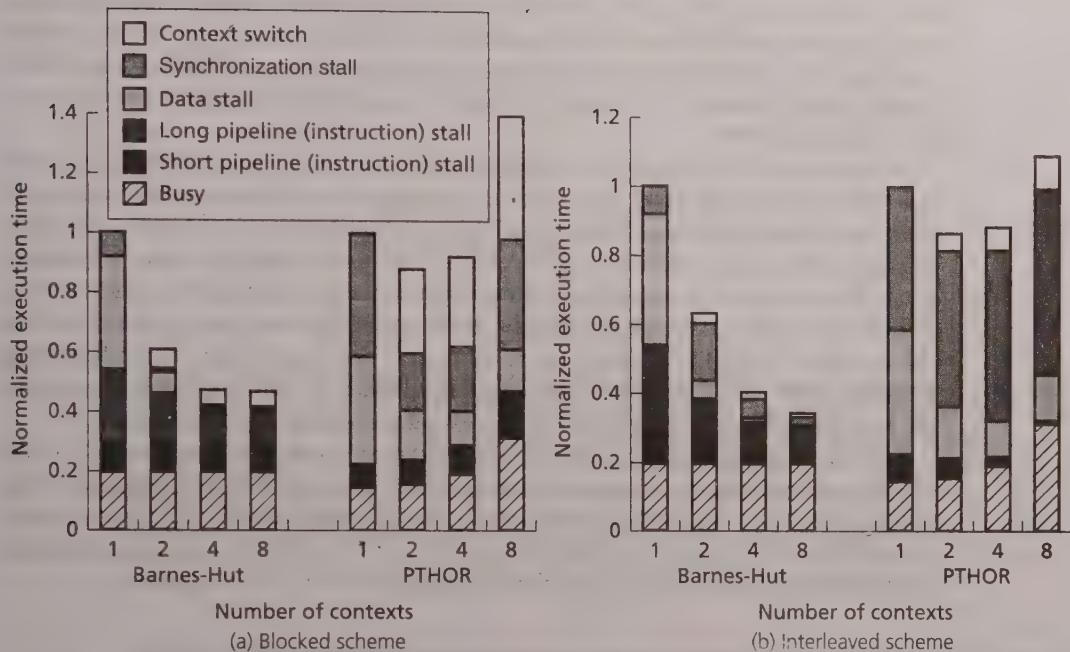


FIGURE 11.32 Execution time breakdowns for two applications under multithreading. Busy time is the time spent executing application instructions; the pipeline stall time is the time spent stalled due to pipeline or instruction dependences; data stall time and synchronization stall time are the time spent stalled on the memory system and at synchronization events, respectively. Finally, context switch time is the time spent in context switch overhead.

Figure 11.32 shows more detailed breakdowns of execution time, averaged over all processors, for two applications that illustrate interesting effects. With a single context, Barnes-Hut shows significant memory stall time due to the small single-level direct-mapped cache used (as well as the small problem size of only 4-K bodies). The use of more contexts per processor is able to hide most of the data access latency, and the lower switch cost of the interleaved scheme is clear from the figure. The other major form of latency in Barnes-Hut (and in Water) is pipeline stalls due to long-latency floating-point instructions, particularly divides. The interleaved scheme is able to hide this latency more effectively than the blocked scheme. However, both start to taper off in their ability to hide divide latency at more than four contexts. This is because the simulated divide unit is not pipelined, so it quickly becomes a resource bottleneck when divides from different contexts compete for it. PTHOR is an example of an application in which the use of more contexts does not help very much and even hurts as more contexts are used. Memory latency is hidden quite well as soon as we go to two contexts, but the major bottleneck is synchronization latency. The application simply does not have enough extra parallelism (slackness) to exploit multiple contexts effectively: even though multiple threads are used,

they spend most of their time serialized at synchronization points. Note that busy time increases with the number of contexts in PTHOR. This is because the application uses a set of distributed task queues, and more time is spent maintaining these queues as the number of threads increases. These extra instructions cause more cache misses as well.

Multithreading hides the same types of data access latency as prefetching, and one may be better than the other in certain circumstances (recall that multithreading also hides synchronization and instruction latency, which prefetching doesn't directly). The performance benefits of using the two techniques together are not well understood. For example, the use of multithreading may cause constructive or destructive interference in the cache among threads; this is very difficult to predict, which makes the analysis of what to prefetch more difficult. Like prefetching, multithreading complements relaxed memory consistency quite well with blocking-read processors; the interactions with more aggressive processors are not yet well understood.

The next two subsections discuss some detailed implementation issues for the blocked and interleaved schemes, focusing on the additional implementation complexity needed to implement each scheme beyond that needed for a commodity microprocessor. Readers can skip to Section 11.7.5 to see the more sophisticated multithreading scheme for superscalar processors without loss of context.

11.7.3 Implementation Issues for the Blocked Scheme

Both the blocked and interleaved schemes have three kinds of requirements: *state replication*, *program counter (PC) unit enhancements*, and *control enhancements*. State replication essentially involves replicating the registers, program counter, and relevant portions of the processor status word once per active context, as discussed earlier. The PC of the processor requires significant changes for multithreading control. For control enhancements, logic and registers are needed to manage switching between contexts, making contexts ready and unready, and so on. We treat each of these requirements in turn.

State Replication

Let us look at the register file and the processor status word separately. Giving every active context its own register file or piece of a larger, statically segmented register file allows registers to be accessed quickly, though this may not use the silicon area efficiently (see Figure 11.33). For example, since only one context runs at a time until it encounters a long-latency event in the blocked scheme, only one register file is actively being used for some time while the others are idle. At the very least, we would like to share the read and write ports across register files since these ports often take up a substantial portion of the silicon area of the files. In addition, some contexts might require more or fewer registers than others, and the relative needs may change dynamically. Thus, allowing the contexts to share a large register file dynamically according to need may provide better register utilization than dividing

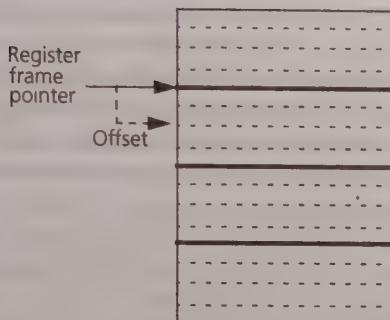


FIGURE 11.33 A segmented register file for a multithreaded processor. The file is divided into four frames, assuming that four contexts can be active at a given time. The register values for each active context remain in that context's register frame across context switches, and each context's frame is managed by the compiler as if it were itself a complete register file. A register in a frame is accessed through the current (hardware) register frame pointer, by specifying an offset within the frame, so the compiler need not be aware of which particular frame a context is using (which is determined at run time). Switching to a different active context requires only that the current frame pointer be changed.

the registers statically. This results in a cachelike structure, indexed by context identifier and register offset, with the potential disadvantage that the register file is larger and so has a higher access time. Several proposals have been made to improve register file efficiency (Nuth and Dally 1995; Laudon 1994; Omondi 1994; Smith 1985), but substantial replication is needed in all cases. The MIT Alewife machine uses the register windows mechanism of a modified Sun Sparc processor to provide a replicated register file.

A modern “processor status word” is actually several registers; only some parts of it (such as floating-point status/control, etc.) contain process-specific state rather than global machine state, so only these parts need to be replicated. In addition, multithreading introduces a new global status word called the *context status word* (CSW). This contains an identifier that specifies which context is currently running, a bit that says whether context switching is enabled (we shall see that it may be disabled while exceptions are handled), and a bit vector that tells us which of the currently active contexts are ready to execute. Finally, TLB control registers need to be modified to support different address space identifiers from the different contexts and to allow a single TLB entry to be used for a page that is shared among contexts.

Program Counter Unit

Different active contexts must also have their PCs available in hardware. Processors that support exceptions efficiently provide a mechanism to do this with minimal hardware replication since in many ways exceptions behave like context switches. In addition to the PC chain, which holds the PCs for the instructions that are in the different stages of the pipeline, a register called the *exception program counter* (EPC) is

provided in such processors. The EPC is fed by the PC chain, so it always contains the address of the last instruction retired from the pipeline. When an exception occurs, the loading of the EPC is stopped with the faulting instruction, all the incomplete instructions in the pipeline are squashed, and the exception handler address is put in the PC. When the exception handler returns, the EPC is loaded into the PC so that the faulting instruction is reexecuted. This is exactly the functionality we need for multiple contexts. We simply need to replicate the EPC register, providing one per active context. The EPC for a context serves to handle both exceptions as well as context switches for that thread. When a given context is operating, the PC chain feeds into its EPC while the EPCs of other contexts simply retain their values. On an exception, the current context's EPC behaves exactly as just described for the single-threaded case. On a context switch, the current context's EPC stops being loaded at the faulting (long-latency) instruction, and the incomplete instructions in the pipeline are squashed (as for an exception). The PC is loaded from the EPC of the next selected context, which therefore starts executing from its first unexecuted instruction, and so on. The only drawback of using the EPCs for these dual purposes is that now an exception handler cannot take a context switch (since the address of the next unexecuted instruction loaded into its EPC by the context that incurred the exception will be lost), so context switches may have to be disabled when an exception occurs and reenabled when the exception handler returns. However, the PCs can still be managed through software saving and restoring even in this case.

Control

The key functions of the control logic in a blocked implementation are to detect when to switch contexts, to choose the context to switch to, and to orchestrate and perform the switch. Let us discuss each briefly.

A context switch in the blocked scheme may be triggered by three events: a cache miss; an explicit context switch instruction, used for synchronization events and very long-latency instructions; and a time-out. The time-out is used to ensure that a single context does not run too long or spin waiting on a flag to be set by another thread running on the same processor. The decision to switch on a cache miss is based on three signals: the cache miss notification, the bit that says that context switching is enabled, and a signal that states that another context is ready to run. A simple way to implement an explicit context switch instruction is to have it behave as if the following instruction generated a cache miss (i.e., to raise the cache miss signal or generate another signal that has the same effect on the context switch logic); this will cause the context to switch and to be restarted later from that following instruction. Finally, the time-out signal can be generated via a resettable threshold counter.

While many policies can be used to select the next context upon a switch, in practice simply switching to the next active and ready context in a round-robin fashion—without concern for special relationships among contexts or the history of the contexts' executions—seems to work quite well. The signals this requires are the

current context identifier, the vector of ready and active contexts, and the signal that detects the need to switch.

Finally, orchestrating a context switch in the blocked scheme requires the following actions. They are required to complete by different stages in the processor pipeline, so the control logic must enable the corresponding signals in the appropriate time windows.

- Save the address of the first uncompleted instruction from the current thread (in that context's EPC, say).
- Squash all incomplete instructions in the pipeline.
- Start executing from the (saved) PC of the selected context, obtained from its EPC.
- Load the appropriate address space identifier in the TLB's bound registers.
- Load the relevant control/status registers from the (saved) processor status words of the new context, including the floating-point control/status register and the context identifier.
- Switch the register file control to the register file for the new context, if applicable.

In summary, the major costs of implementing blocked context switching come from replicating and managing the register file, which increases both area and perhaps register file access time. If the latter is in the critical path of the processor cycle time, it may require the pipeline depth to be increased to maintain a high clock rate, which can increase the penalty for branch mispredictions. All these factors must be considered in evaluating performance benefits. The other hardware costs are very small.

11.7.4 Implementation Issues for the Interleaved Scheme

A key reason that the blocked scheme is relatively easy to implement is that most of the time the processor behaves like a single-threaded processor, invoking additional complexity and processor state changes only at context switches. The interleaved scheme needs a little more support since it switches among threads every cycle. The processor state may have to be changed every cycle and the instruction issue unit must be capable of issuing from multiple active streams in consecutive cycles. A mechanism is also needed to make contexts active and inactive and to feed the active/inactive status into the instruction unit every cycle. Let us again look at the state replication, PC unit, and control needs separately.

State Replication

The register file must be replicated or managed dynamically as for the blocked scheme, but the pressure on fast access to different parts of the entire register file is greater since successive cycles may access the registers of different contexts. We cannot rely on the more gradually changing access patterns of the blocked scheme. (Thus, the Tera processor uses a banked or interleaved register file, and a thread may

be rendered unready because of a busy register bank as well.) The parts of the process status word that must be replicated are similar to those in the blocked scheme, though again the processor must be able to switch status words every cycle.

Program Counter Unit

The greatest difference in changes is to the PC unit. Instructions from different threads are in the pipeline at the same time, and the processor must be able to issue instructions from a different thread every cycle, avoiding unready threads. The processor pipeline is also impacted since, to implement bypassing and forwarding correctly, the processor's PC chain must now carry a context identifier for each pipeline stage. In the PC unit itself, new mechanisms are needed for handling context availability and for squashing instructions, for keeping track of the next instruction to issue, for handling branches, and for handling exceptions. Let us examine some of these issues briefly. A fuller treatment can be found in the literature (Laudon 1994).

Consider context availability. Contexts become unavailable because of either cache misses or explicit backoff instructions that make the context unavailable for a specified number of cycles. The backoff instructions are issued, for example, at synchronization events (if the synchronization event has not been satisfied by the time the specified backoff period expires and the thread is made available again, the cycles for that thread may be wasted until the synchronization event is satisfied, or another backoff may be issued). The issuing of further instructions from that context is stopped by clearing a "context available" signal. To squash the instructions already in the pipeline from that context, we must broadcast a squash signal as well as the context identifier to all stages since we don't know which stages contain instructions from that context. In the case of a cache miss, the address of the instruction that caused the miss is loaded into the EPC. Once the cache miss is satisfied and the context becomes available again, the PC bus is loaded from the EPC when that context is selected next. Explicit backoff instructions are handled similarly to cache misses, except that we do not want the context to resume from the backoff instruction itself but rather from the instruction that follows it. A bit called the *next bit* can be included in the EPC to orchestrate resumption from either the faulting instruction or the next one.

Even in a standard, single-context uniprocessor, three sources can determine the next instruction to be issued from a given thread: the next sequential instruction, the predicted branch from the branch target buffer (BTB), and the computed branch if the prediction is detected to be wrong. When only a single context is in the pipeline at a time, the appropriate next instruction address can be driven onto the PC bus from the "next PC" (NPC) register as soon as it is determined. In an interleaved processor, however, in the cycle when the next instruction address for a given context is determined and ready to be put on the PC bus, it may not be the context scheduled for that cycle. Further, since the NPC for a context may be determined in different pipeline stages for different instructions—for example, it is determined much later for a mispredicted branch than for a correctly predicted branch or a non-branch instruction—different contexts could produce their NPC value during the

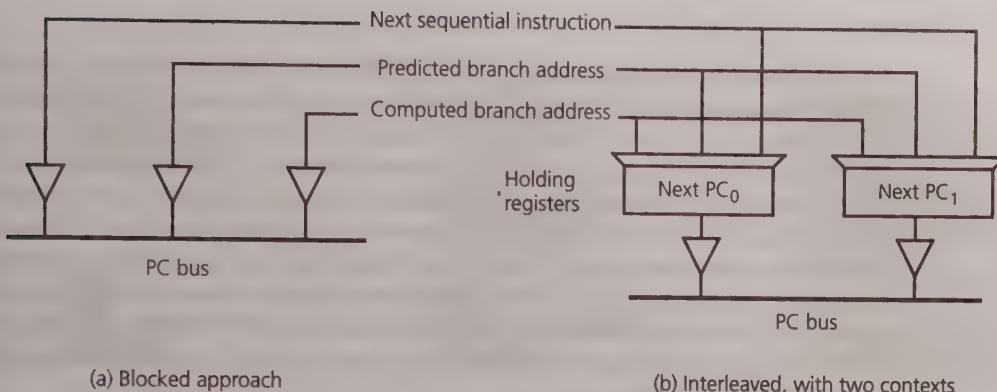


FIGURE 11.34 Driving the PC bus in the blocked and interleaved multithreading approaches, with two contexts. If more contexts were used, the interleaved scheme would require more replication whereas the blocked scheme would not.

same cycle. Thus, the NPC value for each context must be held in a holding register until it is time to execute the next instruction from that context, at which point it will be driven onto the PC bus (see Figure 11.34).

Branches too require some additional mechanisms. The context identifier must be broadcast to the pipeline stages when squashing instructions due to a mispredicted branch, but this is the same functionality needed when making contexts unavailable. By the time the actual branch target is computed, the predicted instruction that was fetched speculatively could be anywhere in the pipeline or may not even have been issued yet (since other contexts will be interleaved unpredictably). To find this predicted instruction address to determine the correctness of the prediction, it may be necessary for branch instructions to carry along with them their predicted address as they proceed along the pipeline stages. For example, a *predicted PC* register chain can run along parallel to the PC chain and be loaded and checked as the branch reaches the appropriate pipeline stages.

Finally, consider what happens when an exception occurs in one context. One choice is to have that context be rendered unready to make way for the exception handler and let the exception handler be interleaved with the other user contexts (the Tera takes an approach similar to this). In this case, another user thread may also take an exception while the first exception handler is running, so the exception handlers must be able to cope with multiple concurrent handler executions. Another option is to render all the contexts unready when an exception occurs in any context, squash all the instructions in the pipeline, and reenable all contexts when the exception handler returns. This can cause a loss of performance if exceptions are frequent. It also means that, when an exception occurs, the exception PCs (EPCs) of all active contexts must be loaded with the address of the first uncompleted instruction from their respective threads. This is more complicated than in the blocked case, where only the single EPC of the currently running (excepting) context needs to be saved.

Control

Two interesting issues related to control outside of the PC unit are tracking context availability information and feeding it to the PC unit, and choosing and switching to the next context every cycle. The “context available” signal is modified on a cache miss, when the miss returns, and on backoff instructions and their expiration. Availability status due to cache misses can be tracked by maintaining pending miss registers per context, which are loaded upon a miss and checked upon miss return to reenable the appropriate context. For explicit backoff instructions, we can maintain a counter per context, initialized to the backoff value when the backoff instruction is encountered (the context availability signal is also cleared at this time). The counter is decremented every cycle until it reaches zero, at which point the availability signal for that context is set again.

Backoff instructions can be used to tolerate instruction latency as well, but with the interleaving of contexts it may be difficult to choose a good number of backoff cycles. This is further complicated by the fact that the compiler may rearrange instructions transparently. Backoff values are implementation specific and may have to be changed for subsequent generations of processors. Fortunately, short instruction latencies are often handled naturally by the interleaving of other contexts without any backoff instructions, as we saw in Figure 11.30. Robust solutions for long instruction latencies may require more complex hardware support such as scoreboarding.

As for choosing the next context, a reasonable approach once again is to select contexts round-robin qualified by context availability.

11.7.5 Integrating Multithreading with Multiple-Issue Processors

So far, our discussion of multithreading has been orthogonal to the number of operations issued per cycle. While the Tera system issues three operations per cycle, the packing of operations from a thread into wider instructions is done by the compiler, and the hardware simply chooses a three-operation instruction from a single thread in every cycle. A single thread usually does not have enough instruction-level parallelism to fill all the available slots in every cycle, as is already being found in modern multiple-issue processors and is likely to become worse if support for issuing more operations per cycle is provided. With many threads available, a natural alternative is to let available operations from different threads be scheduled in the same cycle, thus filling the issue slots more effectively. This approach has been called *simultaneous multithreading*, and there have been many proposals for it (Hirata et al. 1992; Tullsen, Eggers, and Levy 1995). It is like interleaved multithreading, but operations from the different available threads compete for the issue slots and functional units in every cycle.

Put another way, traditional multiple-issue processors suffer from two inefficiencies. First, not all slots in a given cycle are filled due to limited ability to find instruction-level parallelism within a thread. Second, many cycles have nothing scheduled because of long-latency instructions. Simple multithreading addresses the

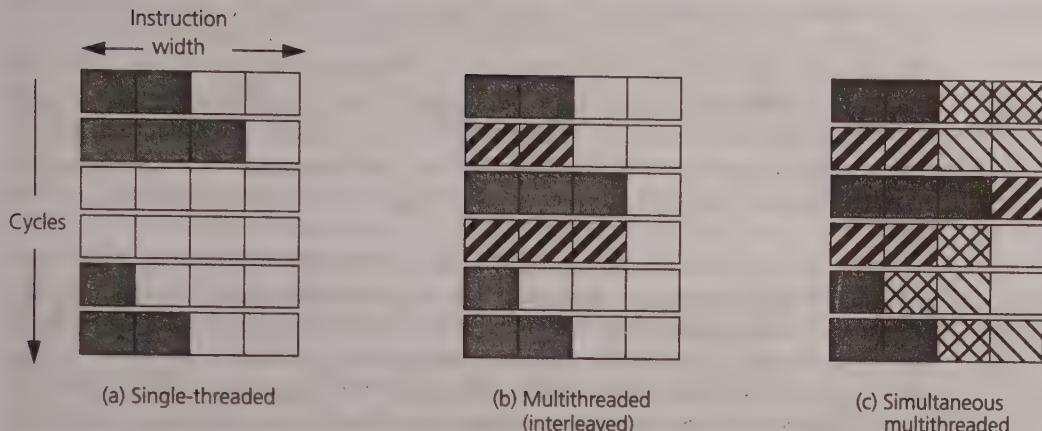


FIGURE 11.35 Simultaneous multithreading. The potential improvements are illustrated for both simple interleaved multithreading and simultaneous multithreading for a four-issue processor. Shaded and patterned boxes distinguish operations from different threads, while blank boxes indicate empty slots in instructions.

second problem but not the first whereas simultaneous multithreading tries to address both (see Figure 11.35).

Choosing operations from different threads to schedule in the same cycle may be difficult for a compiler, but many of the mechanisms for it are already present in dynamically scheduled microprocessors. The instruction fetch unit must be extended to fetch operations from different hardware contexts in a cycle, but once operations from the different contexts are fetched and placed in the reorder buffer, the issue logic can choose operations from this buffer regardless of which context they are from. Studies of single-threaded, multiple-issue dynamically scheduled processors have shown that the causes of empty cycles and empty slots are quite well distributed among instruction latencies, cache misses, TLB misses, and load delay slots (with the first two often being particularly important). The variety of the sources of wasted time and of their latencies indicates that fine-grained multithreading may be a good solution.

In addition to the issues we discussed for interleaved multiprocessors, several new issues arise in implementing simultaneously multithreaded processors (Tullsen et al. 1996). First, how flexible should instruction fetching from different threads be? The greater the flexibility allowed—compared to fetching from only one context in a cycle or fetching at most two operations from each thread in a cycle—the greater the complexity in the fetching logic and instruction cache design. However, more flexibility reduces the frequency of empty fetch slots. Second, how should we choose which context or contexts to fetch instructions from in the next cycle? We could choose contexts in a fixed order of priority (say, try to fill from context 0 first, then fill the rest from context 1, and so on) or we could choose based on execution

characteristics of the contexts (for example, give priority to the context that has the fewest instructions currently in the fetch unit or the reorder buffer or to the context that currently has the fewest outstanding cache misses). Finally, which operations should we choose from the reorder buffer among the ones that are ready in each cycle? The standard practice in dynamically scheduled processors is to choose the oldest operation that is ready, but other choices, based on which threads the operations are from and how the threads are behaving, may be more appropriate in this case.

Little or no performance data exists on simultaneous multithreading in the context of multiprocessors. For uniprocessors, a performance study examining the potential benefits as well as the impact of some of the trade-offs just discussed finds that the technique is promising and that support for speculative execution is less important with simultaneous multithreading than with single-threaded dynamically scheduled processors because there are more available threads and, hence, non-speculative instructions to choose from (Tullsen et al. 1996).

Overall, as data access and synchronization latencies become larger relative to processor speed, and as the data access patterns of multiprocessor applications become more complex and unpredictable (as multiprocessing continues to mature and expand), multithreading promises to become increasingly successful in hiding latency. Whether it will actually be incorporated in microprocessors depends on a host of factors, such as what other latency tolerance techniques are employed (e.g., prefetching, dynamic scheduling, and relaxed consistency models) and how multithreading interacts with them. Since multithreading already requires extra explicit threads and significant complexity and replication of state, an interesting alternative is to place multiple simpler processors on a chip with the multiple threads running on different processors. While the qualitative trade-offs are quite clear, how this organization compares with multithreading in cost and performance is not yet well understood, either for desktop systems or as a node for a larger multiprocessor.

Table 11.2 summarizes and compares some key features of the four major techniques for hiding latency in a shared address space, as presented in Sections 11.4–11.7. The techniques can be and often are combined. For example, processors with blocking reads can use relaxed consistency models to hide write latency and prefetching or multithreading to hide read latency. And we have seen that dynamically scheduled processors can benefit from all of prefetching, relaxed consistency models, and multithreading individually. How these different techniques interact in dynamically scheduled processors, how well prefetching might complement multithreading even in blocking read processors, and how these techniques succeed in hiding latency as the gap between processor speed and data access latency widens is likely to be better understood in the future.

11.8

LOCKUP-FREE CACHE DESIGN

Throughout this chapter, we have seen that in addition to the support needed in the processor—and the additional bandwidth and low occupancies needed in the memory and communication systems—several latency tolerance techniques in a shared

Table 11.2 Key Properties of the Four Latency Tolerance Techniques in a Shared Address Space

Property	Relaxed Models	Prefetching	Multithreading	Block Data Transfer
Types of Latency Tolerated	Write (blocking read processors) Read and write (dynamically scheduled processors)	Write Read	Write Read Synchronization Instruction	Write Read
Requirements of Software	Labeling synchronization operations	Predictability	Explicit extra concurrency	Identifying and orchestrating block transfers
Extra Hardware Support	Little	Little	Substantial	Not in processor but block transfer engine in memory system
Support in Commercial Micros?	Yes	Yes	Currently no	Not needed

address space require that the cache allow multiple outstanding misses at a time if the techniques are to be effective. Before we conclude the chapter, let us examine the design of such a lockup-free cache.

There are several key design questions for a cache subsystem that allows multiple outstanding misses:

- How many and what kinds of misses can be outstanding at the same time? Like the processor, it is easier for the cache to support multiple outstanding writes than reads. Two distinct points in design complexity are (1) a single read and multiple writes and (2) multiple reads and writes.
- How do we keep track of the outstanding misses? For reads, we need to track: the address of the word requested; the type of read request (i.e., read, read exclusive, or prefetch and single-word or double-word read); the place to return data when it comes into the cache (e.g., to which register within a processor or to which processor if multiple processors are sharing a cache); and the current status of the outstanding request. For writes, we do not need to track where to return the data, but the new data being written must be merged with the data block (if any) returned by the next level of the memory hierarchy. A key issue here is whether to store most of this information within the cache blocks themselves or to have a separate set of transaction buffers for outstanding misses. Of course, while fulfilling these requirements, we need to ensure that the design is free of deadlock and livelock.
- How do we deal with conflicts among multiple outstanding references to the same memory block? What kinds of conflicting misses to a block should we disallow (e.g., by stalling the processor)? For example, should we allow writes to words within a block to which a read miss is outstanding?

- How do we deal with conflicts between multiple outstanding requests that map to the same line in the cache, even though they refer to different memory blocks?

To illustrate the options, let us examine two different designs. They differ primarily in where they store the information that keeps track of outstanding misses. The first design uses a separate set of transaction buffers for tracking requests. The second design, to the extent possible, keeps track of outstanding requests in the cache blocks themselves.

The first design is a simplified version of that used in Control Data Corporation's Cyber 835 mainframe, introduced in 1979 (Kroft 1981). It adds a number of miss state holding registers (MSHRs) to the cache, together with some associated logic. Each MSHR handles one or more outstanding misses to a single memory block. This design allows considerable flexibility in the kinds of requests that can be simultaneously outstanding to a block, so a significant amount of state is stored in each MSHR as shown in Table 11.3.

The MSHRs are accessed in parallel to the regular cache. If the access hits in the cache, the normal cache hit actions take place. If the access misses in the cache, the actions depend on the contents of the MSHRs:

- If no MSHR is allocated for that block, a new one is allocated and initialized (if no MSHR is free, or if all cache lines within that set in the cache have pending requests, then the processor stalls). If the cache line on which the miss occurs currently contains dirty data, a write back is initiated. Then, if the processor request is a write, the data is written at the proper offset into the block in the cache, and the corresponding partial write code bits are set in the MSHR. A request to fetch the block from the main memory subsystem (e.g., BusRd, BusRdX) is also initiated.
- If an MSHR is already allocated for the block, the new request is merged with the previously pending requests for the same block. For example, a new write request can be merged by writing the data into the allocated cache block and by setting the corresponding partial write bits in the MSHR. A read request to a word that has been written completely in the cache (by earlier writes) can simply read the data from the cache already. A read request to a word that has not been requested is handled by setting the proper unit identification tags. If it is to a word that has already been requested, then either a new MSHR must be allocated (since there is only one unit identification tag per word) or the processor must be stalled. Since a write does not need a unit identification tag, a write request for a word to which a request is already pending is handled easily: the data returned by main memory can simply be forwarded to the processor. Of course, the first such write to a block will have to generate a request that asks for exclusive ownership.

Finally, when the data for the block returns to the cache, the cache block pointer in the MSHR indicates where to put the contents. The partial write codes are used to avoid overwriting more recently written data in the cache, and the send-to-CPU bits and unit identification tags are used to forward replies to waiting functional units.

Table 11.3 MSHR State Entries and Their Roles

State	Description	Purpose
Cache Block Pointer	Pointer to cache block allocated to this request	Specifies cache line within a set in a set-associative cache
Request Address	Address of pending request	Allows merging of requests
Unit Identification Tags (one per word)	Identifies requesting unit in processor	Where to return this word
Send-to-CPU status (one per word)	Valid bits for unit identification tags	Is unit ID tag valid?
Partial Write Codes (one per word)	Bit vector for tracking partial writes to a word	Which words returning from memory to overwrite
Valid Indicator	Valid MSHR contents	Are contents valid?

This design requires an associative lookup of MSHRs but allows the cache to have all kinds of memory references outstanding at the same time. Fortunately, since the MSHRs do the complex task of merging requests and tearing apart replies for a given block, to the extended memory hierarchy below it appears simply as if there are requests to distinct blocks coming from the processor. The coherence protocols we have discussed in the previous chapters are already designed to handle these multiple outstanding requests to different blocks without deadlock.

The alternative to this design is to store most of the relevant state for outstanding write requests in the cache lines themselves and not use separate MSHRs. In addition to the standard MESI states for a write-back cache, we add three transient or pending states: *invalid pending* (IP), *shared pending* (SP), and *exclusive pending* (EP), which indicate what the state of the block was when a currently outstanding write miss was issued. In each of these three states, the cache tag is valid and the cache block is awaiting data from the memory system. Each cache block also has a bit vector of *subblock write bits* (SWBs), with 1 bit per word. In both the EP and SP states, the bits that are turned ON indicate the words in the block that have been written by the processor since the block was requested from memory and that the data returned from memory should not overwrite. However, words for which the bits are OFF are considered invalid in the EP state but valid (not stale) in the SP state. Finally, there is a set of separate pending read registers; these contain the address and type of pending read requests.

The key benefit of keeping this extra state information with each cache block is that no additional storage is needed to keep track of pending write requests. On a write that does not find the block in modified state, the block simply goes into the appropriate pending state, initiates the appropriate transaction, and sets the SWB bits to indicate which words the current write has modified so the subsequent merge will happen correctly. Writes that find the block already in pending state only require that the word is written into the line and the corresponding SWB is set. Reads may use the pending read registers. If a read finds the desired word in a valid

state in the block (including a pending state with the SWB on), then it simply returns it. Otherwise, it is placed in a pending read register that keeps track of it.

If the block accessed on a read or write is not in the cache (no tag match), then a write back may be generated. The block is set to the invalid pending state, all SWBs are turned off (except for the word being written if it is a write), and the appropriate transaction is placed on the bus. If the tag does not match and the existing block is already in pending state, then the processor stalls. Finally, when a response to an outstanding request arrives, the corresponding cache block is updated except for the words that have their SWBs on. The cache block moves out of the pending state. All pending read registers are checked to see if any are waiting for this cache block; if so, data is returned for those requests and those pending read registers are freed. Details of the actual state changes, actions, and race conditions can be found in (Laudon 1994). One key observation that makes race conditions relatively easy to deal with is that, even though words are written into cache blocks before ownership is obtained for the block, those words are not visible to requests from other processors until ownership is obtained.

Overall, these two lockup-free cache designs are not that different conceptually. The latter solution keeps the state for writes in the cache blocks and reduces the number of pending registers needed and the complexity of the associative lookups; however, it is more memory intensive than MSHRs since extra state is stored with all lines in the cache, even though very few of them will have an outstanding request at a time. The correctness interactions with the rest of the protocol are similar and modest in the two cases.

11.9

CONCLUDING REMARKS

With the increasing gap between processor speeds and memory access and communication times, latency tolerance will be increasingly critical in future multiprocessors (and uniprocessors as well). Many latency tolerance techniques have been developed, and each has its relative advantages and disadvantages. They all rely on excess concurrency in the application program beyond the number of processors used, and they all tend to increase the bandwidth demands placed on the communication architecture. This greater stress makes it all the more important that the other performance aspects of the communication architecture (the processor overhead, the assist occupancy, and the network bandwidth) be efficient and well balanced. For example, since the overhead incurred on the main processor cannot be hidden from that processor, if overhead is a dominant component of data access latency, then latency tolerance techniques other than making messages larger might not be very effective.

For cache-coherent multiprocessors, latency tolerance techniques are supported in hardware by both the processor and the cache memory system, leading to a rich space of design alternatives. Most of these hardware-supported latency tolerance techniques are also applicable to uniprocessors; in fact, their commercial success depends on their viability in the high-volume uniprocessor market where the latencies to be hidden are smaller. Techniques like dynamic scheduling, relaxed memory con-

sistency models, and prefetching are commonly encountered in microprocessor architectures today. The most general latency hiding technique—multithreading—is not yet popular commercially, largely because it is unproven for uniprocessors. Recent directions in integrating multithreading with dynamically scheduled superscalar processors appear promising, but they bear comparison with multiple simpler processors on a chip. An interesting general question is how well the provisions made for hiding uniprocessor latencies will succeed in hiding multiprocessor latencies.

Despite the rich space of issues and alternatives in hardware support, much of the latency tolerance problem today is also a software problem. To what extent can a compiler automate prefetching so a user does not have to worry about it? And if automation to a desirable extent is not possible, how can the user naturally convey information about what and when to prefetch to the compiler? If block transfer is indeed useful on cache-coherent machines, how will users program to this mixed model of both implicit communication through reads and writes as well as explicit transfers? Relaxed consistency models carry with them the software problem of specifying the appropriate constraints on reordering (i.e., of labeling conflicting operations as necessary). Finally, will programs be decomposed and assigned with enough extra explicit parallelism (extra threads) that multithreading will be successful? Automating and simplifying the software support required for latency tolerance is a task that is far from fully accomplished. In fact, how latency tolerance techniques will play out in the future and what software support they will use remain interesting open questions in parallel architecture.

11.10 EXERCISES

- 11.1 Why is latency reduction generally a better idea than latency tolerance?
- 11.2 Suppose a processor communicates k words in m messages of equal size, the assist occupancy for processing a message is o , and there is no overhead on the processor. What is the best-case latency as seen by the processor if only communication, not computation, can be overlapped with communication? First, assume that acknowledgments are free (i.e., are propagated instantaneously and don't incur overhead); then include acknowledgments. Draw timelines and state any important assumptions.
- 11.3 You have learned about a variety of different techniques to tolerate and hide latency in shared memory multiprocessors. These techniques include blocking, prefetching, multiple context processors, and relaxed consistency models. For each of the following scenarios, discuss why each technique will or will not be an effective means of reducing/hiding latency. Assume a processor with blocking reads and list any other assumptions that you make.
 - a. A complex graph algorithm with abundant concurrency using linked pointer structures.
 - b. A parallel sorting algorithm where communication is producer initiated and is achieved through long-latency write operations. Receiver-initiated communication is not possible.

- c. An iterative equation solver in which the inner loop consists of a matrix-matrix multiply. Assume both matrices are huge and don't fit into the cache.
- 11.4 You are charged with implementing message passing on a new parallel supercomputer. The architecture of the machine is still unsettled, and your boss says the decision of whether to provide hardware cache coherence will depend on the message-passing performance of the two systems under consideration since you want to be able to run message-passing applications that ran on the previous-generation architecture.
- In the system without cache coherence (the “NCC” system), the engineers on your team tell you that message passing should be implemented as successive transfers of 1 KB. To avoid problems with buffering at the receiver side, you’re required to acknowledge each individual 1-KB transfer before the transmission of the next one can begin (so only one block can be in flight at a time). Each 1-KB transfer requires 200 cycles of setup time, after which it begins flowing into the network. This overhead accounts for time to determine where to read the buffer from in memory and to set up the DMA engine, which performs the transfer. Assume that from the time that the 1-KB chunk reaches the destination it takes 20 cycles for the destination node to generate a response, and it takes 50 cycles on the sending node to accept the ACK and proceed to the next 1-KB transfer.
- In the system with cache coherence (the “CC” system), messages are sent as a series of 128-byte cache line transfers. In this case, however, acknowledgments only need to be sent at the end of every 4-KB page. Here, each transfer requires 50 cycles of setup time, during which time the line can be extracted from the cache, if necessary, to maintain cache coherence. This line is then injected into the network, and only when the line is completely injected into the network can processing on the next line begin.
- The following are the system parameters: clock rate = 10 ns (100 MHz), network latency = 30 cycles, network bandwidth = 400 MB/s. State any other assumptions that you make.
- What is the latency (until the last byte of the message is received at the destination) and achieved bandwidth for a 4-KB message in the NCC system?
 - What is the corresponding latency and bandwidth in the CC system?
 - A designer on the team shows you that you can easily change the CC system so that the processing for the next line occurs while the previous one is being injected into the network. Calculate the 4-KB message latency for the CC system with this modification.
- 11.5 Consider the example of transposing a matrix of data in parallel, as is used in computations such as high-performance Fast Fourier Transforms. Figure 11.36 shows the transpose pictorially. Every processor transposes one “patch” of its assigned rows to every other processor, including one to itself. Performing the transpose through reads and writes was discussed in the Chapter 8 Exercises. Since it is completely predictable which data a processor has to send to which other processors, a processor can send an entire patch at a time in a single message rather than communicate the patches through individual read or write cache misses.

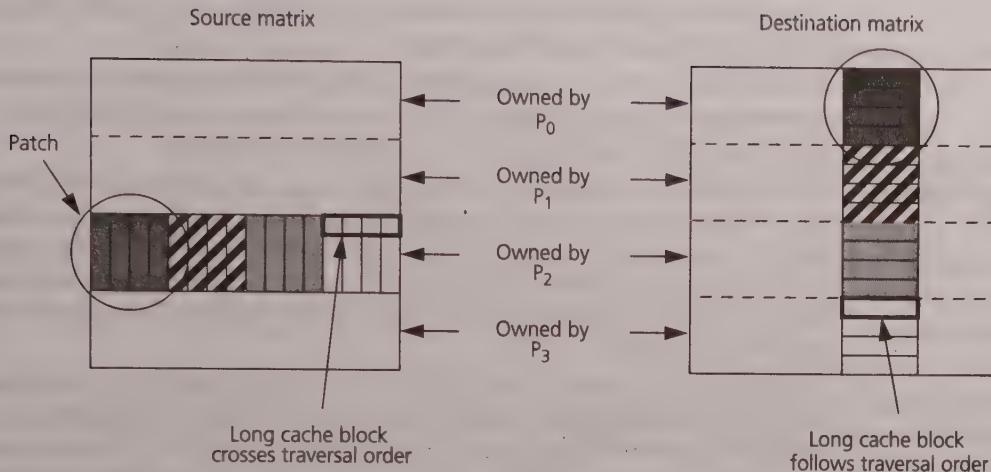


FIGURE 11.36 Sender-initiated matrix transposition. The source and destination $n \times n$ matrices are partitioning among processes in groups of contiguous rows. Each process divides its set of n/p rows into p patches of size $n/p \times n/p$. Consider process P_2 as a representative example: it sends one patch to every other process and transposes one patch (third from left in this case) locally. Every patch may be transferred as a single block transfer message rather than through individual remote writes or remote reads (if receiver initiated).

- What would you expect the curve of block transfer performance relative to read-write performance to look like?
 - What special features would the block transfer engine benefit most from?
 - Write pseudocode for the block transfer version of the code.
 - Suppose you want to use block data transfer in the Raytrace application. For what purposes would you use it and how? Do you think you would gain significant performance benefits?
- 11.6 An interesting performance issue in block data transfer in a cache-coherent shared address space has to do with the impact of long cache blocks and spatial locality. Assume that the data movement in the block transfer leverages the cache coherence mechanisms. Consider the simple equation solver on a regular grid, with its near-neighbor communication. Suppose the n -by- n grid is partitioned into square sub-blocks among p processors.
- Compared to the results shown for FFT in this chapter, how would you expect the curves for this application to differ when each boundary row or column is sent directly in a single block transfer, and why?
 - How might you structure the block transfer to send only useful data, and how would you expect performance in this case to compare with the previous one?
 - What parameters of the architecture would most affect the trade-offs in part (b)?

- d. If you indeed wanted to use a block cache transfer, what deeper changes might you make to the parallel program?
- 11.7 If the second-level cache is a blocking cache in the performance study of proceeding past writes with blocking reads, is there any advantage to allowing write → write reordering over not allowing it? If so, under what conditions?
- 11.8
- a. Write buffers can allow several optimizations, such as buffering itself, merging writes to the same cache line that is in the buffer, and forwarding values from the buffer to read operations that find a match in the buffer. What constraints must be imposed on these optimizations to maintain program order under SC? Is there a danger with the merging optimizations for the processor consistency model?
 - b. To maintain all program orders as in SC, we can optimize in the following way. In our baseline implementation, the processor stalls immediately upon a write until the write completes. The alternative is to place the write in the write buffer without stalling the processor. To preserve the program order among writes, the write buffer retires a write (i.e., passes it further along the memory hierarchy and potentially makes it visible to other processors) only after the write is complete. The order from writes to reads is maintained by flushing the write buffer upon a read miss.
 - (i) What overlap does this optimization provide?
 - (ii) Would you expect it to yield a large performance improvement? Why or why not?
- 11.9 Consider the implementation requirements for proceeding past memory operations in a cache-coherent shared address space. To proceed past writes, we need a write buffer and nonblocking writes. To proceed past reads effectively, we need nonblocking reads as well as instruction lookahead and speculative execution. At the memory system level, we need lockup-free caches with multiple outstanding misses. These structures close to the processor take care of preserving the consistency model, and the rest of the extended memory hierarchy can reorder operations as it pleases. Consider now the mechanisms needed to preserve a consistency model, given this support.
- a. Most of the mechanisms we need have to do with determining completion of an operation or operations. In a processor with blocking reads, what new mechanisms are needed for preserving release consistency compared to sequential consistency, and what additional structures, if any, would you use to implement them?
 - b. Suppose a write→write MEMBAR is encountered in a processor whose write buffer otherwise allows writes to be reordered. Does the processor have to stall, or can it proceed past the MEMBAR? Explain how the write→write ordering dictated by the MEMBAR will be provided and when the write buffer and processor must stall.
 - c. A key mechanism needed for any consistency model is to count incoming acknowledgments for writes that generate invalidations. The machinery

needed to do this can be located near the processor or near the memory controller. What are the main trade-offs, and what empirical information would help you decide what to do?

- 11.10
 - a. How early can you issue a binding prefetch in a cache-coherent system?
 - b. We have talked about the advantages of nonbinding prefetches in multiprocessors. Do nonbinding prefetches have any advantages in uniprocessors over binding prefetches that prefetch into the register file?
- 11.11 Sometimes a predicate must be evaluated to determine whether or not to issue a prefetch. This introduces a conditional (if) expression around the prefetch inside the loop containing the prefetches. Construct a simple example, with pseudocode, and describe the performance problem. How would you fix the problem?
- 11.12 Describe situations in which a producer-initiated deliver operation might be more appropriate than prefetching or an update protocol. Would you implement a deliver instruction if you were designing a machine?

- 11.13 Consider the loop

```
for i ← 1 to 200
    sum = sum + A[index[i]];
end for
```

Write a version of the code with nonbinding software-controlled prefetches inserted. Include the prologue, the steady state of the software pipeline, and the epilogue. Assume the memory latency is such that it takes five iterations of the loop for a data item to return and that data is prefetched into the primary cache.

- a. When prefetching indirect references as in this example, extra instructions are needed for address generation. One possibility is to save the computed address in a register at the time of the prefetch and then reuse it later for the load. Are there any problems with or disadvantages to this?
 - b. What if an exception occurs when prefetching down multiple levels of indirection in accesses? What complications are caused and how might they be addressed?
- 11.14 Describe some hardware mechanisms (at a high level) that might be able to prefetch irregular accesses, such as records or lists.
 - 11.15 Show how the following loop would be rewritten with prefetching so as to hide latency on a uniprocessor:

```
for i=0 to 128 {
    for j=1 to 32
        A[i,j] = B[i] * C[j]
}
```

Try to reduce overhead by prefetching only those references that you expect to miss in the cache. Assume that a read prefetch is expressed as PREFETCH(&variable) and it fetches the entire cache line in which variable resides in shared mode. A read-exclusive prefetch operation, which is expressed as RE_PREFETCH

(`&variable`), fetches the line in exclusive mode. The machine has a cache miss latency of 32 cycles. Explicitly prefetch each needed variable (don't take into account the cache block size). Assume that the cache is large (so you don't have to worry about conflict misses) but not so large that you can prefetch everything at the start. In other words, we are looking for just-in-time prefetching. The matrix $A[i, j]$ is stored in memory with $A[i, j]$ and $A[i, j+1]$ contiguous. Assume that the computation in the loop takes 8 cycles to complete.

- 11.16 Describe two examples in which a prefetching compiler's decision to assume that everything in the cache is invalidated when it sees a synchronization operation is very conservative, and show how a programmer can do better. It might be useful to think of the case study applications used in the book.
- 11.17 One alternative to prefetching is to use nonblocking load operations and to issue these operations significantly before the data is needed for computation. What are the trade-offs between prefetching and using nonblocking loads in this way?
- 11.18 Implementation issues for software-controlled prefetching can be divided into two categories: instruction set enhancements and keeping track of outstanding prefetches.
- a. In what ways are prefetch instructions different from ordinary instructions?
 - b. There are many options for the format that a prefetch instruction can take. For example, some architectures allow a load instruction to have multiple flavors, one of which can be reserved for a prefetch. Or in architectures that reserve a particular register to always have the value zero (e.g., the MIPS and Sparc architectures), a load with that register as the destination can be interpreted as a prefetch since such a load does not change the contents of the register. A third option is to have a separate prefetch instruction in the instruction set, with a different opcode than a load.
 - (i) Which do you think is the best alternative and why?
 - (ii) What addressing mode would you use for a prefetch instruction and why?
 - c. Is it necessary to maintain state in the processor itself for outstanding prefetches? Does it improve performance? Why or why not? Would you merge this support with that for keeping track of outstanding writes or use separate structures? Discuss the trade-offs.
- 11.19 Consider some policy issues for software-controlled prefetching.
- a. Suppose we issue prefetches when we expect the corresponding references to miss in the primary (first-level) cache. A question that arises is, which levels of the memory hierarchy beyond the first-level cache should we probe to see if the prefetch can be satisfied there? Since the compiler algorithm usually schedules prefetches by conservatively assuming that the latency to be hidden is the largest latency (uncontended, say) in the machine, one possibility is not to even check intermediate levels of the cache hierarchy but to always get the data from main memory or from another processor's cache (if the block is dirty). What are the problems with this method, and which one do you think is most important?

- b. Since prefetches are hints, they can be dropped by the hardware without affecting correctness. Would you drop a prefetch (i) when it incurs a TLB miss, and (ii) when the buffer that keeps track of outstanding memory operations (including prefetches) is full? What are the key issues that inform your choices in the two cases?

11.20 Consider the trade-offs between prefetching into the primary cache and prefetching only into the second-level cache.

- What are the qualitative trade-offs and issues that inform a choice? What would you do?
- Using only the following parameters, construct an analytical expression for the circumstances under which prefetching into the primary cache is beneficial. p_t is the number of prefetches that bring data into the primary cache early enough, p_d is the number of cases in which the prefetched data is displaced from the cache before it is used, p_c is the number of cache conflicts in which prefetches replace useful data, p_f is the number of prefetch fills (i.e., the number of times a prefetch tries to put data into the primary cache), l_s is the access latency to the second-level cache (beyond that to the primary cache), and l_f is the average number of cycles that a prefetch fill stalls the processor. After generating the full-blown general expression, your goal is to find a condition on l_f that makes prefetching into the first-level cache worthwhile. To do this, you can make the following simplifying assumptions: $p_s = p_t - p_d$, and $p_c = p_d$. What about the analysis, or what it leaves out, strikes you as making it most difficult to rely on it in practice?

11.21 Consider a “blocked” context-switching processor (i.e., a processor that switches contexts only on long-latency events). Assume that arbitrarily many threads and contexts are available and clearly state any other assumptions that you make in answering the following questions. The threads of a given application have been analyzed to show the following execution profile:

- 40% of cycles spent on instruction execution (busy cycles)
 - 30% of cycles spent stalled on L₁ cache misses but L₂ hits (10-cycle miss penalty)
 - 30% of cycles spent stalled on L₂ cache misses (30-cycle miss penalty)
- What will be the busy time if the context switch latency (cost) is 5 cycles?
 - What is the maximum context switch latency that will ensure that busy time is greater than or equal to 50%?

11.22 In blocked, multiple-context processors with caches, a context switch occurs whenever a reference misses in the cache. The blocking context at this point goes into “stalled” state, and it remains there until the requested data arrives back at the cache. At that point, it returns to “ready” state, and it will be allowed to run when the active contexts ahead of it block. When an active context first starts to run, it reissues the reference it had blocked on. In the scheme just described, can the interaction between the multiple contexts potentially lead to deadlock? If so, concretely describe an example where none of the contexts make forward progress. How might you prevent the problem? If not, say why not.

- 11.23 What do you think would happen to the idealized curve for processor utilization versus degree of multithreading in Figure 11.25 if cache misses were taken into account? Draw this more realistic curve on the same figure as the idealized curve.
- 11.24 Write the logic equation that decides whether to generate a context switch in the blocked scheme, given the following input signals: CacheMiss or CM, MissSwitch-Enable or MSE (enable switching on a cache miss), CE (signal that allows processor to enable context switching), OneCount(CValid) (number of ready contexts), ES (explicit context switch instruction), and TO (time-out). Write another equation to decide when the processor should stall rather than switch.
- 11.25 In discussing the implementation of the PC unit for the blocked multithreading approach, we said that the use of the exception PC for exceptions as well as context switches meant that context switching must be disabled upon an exception. Does this indicate that the kernel cannot use the hardware-provided multithreading at all? If so, why? If not, how would you arrange for the kernel to use the multiple hardware contexts?
- 11.26 Why is exception handling more complex in the interleaved scheme than in the blocked scheme? How would you handle the issues that arise?
- 11.27 How do you think the Tera processor might do lookahead across branches? The processor provides JUMP_OFTEEN and JUMP_SELDOM branch operations. Why do you think it does this?
- 11.28 Consider a simple, HEP-like multithreaded machine with no caches. Assume that the average memory latency is 100 clock cycles. Each context has blocking loads and the machine enforces sequential consistency.
- Given that 20% of a typical workload's instructions are loads and 10% are stores, how many active contexts are needed to hide the latency of the memory operations?
 - How many contexts would be required if the machine supported release consistency (still with blocking loads)? State any assumptions that you make.
 - How many contexts would be needed for parts (a) and (b) if we assumed a blocked multiple-context processor instead of the cycle-by-cycle interleaved HEP processor? Assume cache hit rates of 90% for both loads and stores.
 - For part (c), what is the peak processor utilization, assuming a context switch overhead of 10 cycles?
- 11.29 Studies of applications have shown that combining release consistency and prefetching always results in better performance than when either technique is used alone. This is not the case when multiple contexts and prefetching techniques are combined; the combined performance can sometimes be worse. Explain the latter observation, using an example situation to illustrate.

Future Directions

In the course of writing this book, the single factor that stood out most among the many interesting facets of parallel computer architecture was the tremendous pace of change. Critically important designs became “old news” as they were replaced by newer designs. Major open questions were answered while new ones took their place. Start-up companies left the marketplace as established companies made bold strides into parallel computing and powerful competitors joined forces. The first teraflops performance was achieved, and workshops had already been formed to understand how to accelerate progress toward petaflops. The movie industry produced its first full-length computer-animated motion picture on a large cluster, and for the first time a parallel chess program defeated a grand master. Meanwhile, multiprocessors emerged in huge volume with the Intel Pentium Pro and its glueless cache coherence memory bus. Parallel algorithms were put to work to improve uniprocessor performance by better utilizing the storage hierarchy. Networking technology, memory technology, and even processor design were all thrown up for grabs as we began looking seriously at what to do with a billion transistors on a chip.

Looking forward to the future of parallel computer architecture, the one prediction that can be made with certainty is continued change. The incredible pace of change makes parallel computer architecture an exciting field to study and in which to conduct research. We need to continually revisit basic questions, such as, What are the proper building blocks for parallel machines? What are the essential requirements on the processor design, the communication assist and how it integrates with the processor, and the memory and the interconnect? Will these continue to utilize commodity desktop components, or will a new divergence take place as parallel computing matures and the great volume of computers shifts into everyday appliances? The pace of change makes for rich opportunities in the industry but also for great challenges.

Although it is impossible to precisely predict where the field will go, this final chapter seeks to outline some of the key areas of development in parallel computer architecture and the related technologies. Whatever directions the market takes and whatever technological breakthroughs occur, the fundamental issues addressed throughout this book will still apply. The realization of parallel programming models will still rest upon the support for naming, ordering, and synchronization. Designers will still battle with overhead, latency, bandwidth, and cost. The core

techniques for addressing these issues will remain valid; however, the way that they are employed will surely change as the critical coefficients of performance, cost, capacity, and scale continue to change. New algorithms will be invented, changing the fundamental application workload requirements, but the basic analysis techniques will remain.

Given the approach taken throughout the book, it only makes sense to structure the discussion of potential future directions around hardware and software. For each, we need to ask which trends are likely to continue, thus providing a basis for evolutionary development, and which are likely to stop abruptly, either because a fundamental limit is struck or because a breakthrough changes the direction. Section 12.1 examines trends in technology and architecture; Section 12.2 looks at how changing software requirements may influence the direction of system design and considers how the application base is likely to broaden and change.

12.1

TECHNOLOGY AND ARCHITECTURE

Technological forces shaping the future of parallel computer architecture can be placed into three categories: evolutionary forces, as indicated by past and current trends, fundamental limits that wall off further progress along a trend, and breakthroughs that create a discontinuity and establish new trends. Of course, only time will tell how these actually play out. This section examines all three scenarios and the architectural changes that might arise.

To help sharpen the discussion, let us consider two questions. At the high end, how will the next factor-of-1,000 increase in performance be achieved? At the more moderate scale, how will cost-effective parallel systems evolve? In 1998, computer systems form a parallelism pyramid roughly as in Figure 12.1. Overall shipments of uniprocessor PCs, workstations, and servers is on the order of tens to hundreds of millions. The 2–4 processor end of the parallel computer market, which makes up the second level, is on the scale of 100,000 to a few million. These are almost exclusively servers, with some growth toward the desktop. This segment of the market grew at a moderate pace throughout the 1980s and early 1990s and then shot up with the introduction of low-cost SMPs manufactured by leading PC vendors, as well as the traditional workstation and server vendors pushing costs down to expand volume. The next level is occupied by machines of 5 to 30 processors. These are exclusively high-end servers. The volume is in the tens of thousands of units and has been growing steadily; this segment dominates the high-end server market, including the enterprise market, which used to be the mainframe market. At the scale of several tens to a hundred processors, the volume is on the order of a few thousand systems. These tend to be dedicated engines supporting massive databases, large scientific applications, or major engineering investigations, such as oil exploration, structural modeling, or fluid dynamics. Volume shrinks rapidly beyond a hundred processors, with the order of tens of systems at the thousand-processor scale. Machines at the very top end have been on the scale of 1,000 to 2,000 processors since 1990. In 1996–1997, this figure stepped up toward 10,000 processors. The most visible machines at the very top end are dedicated to advanced scientific com-

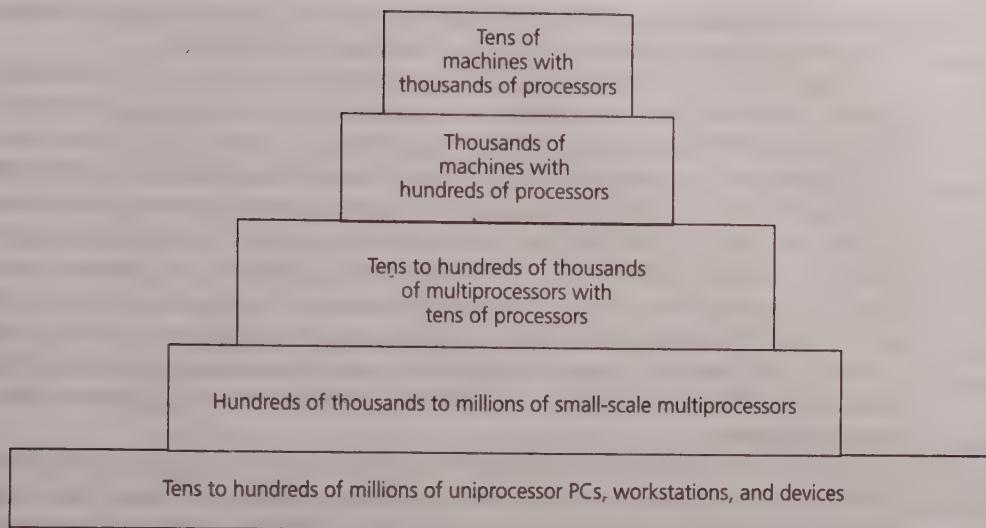


FIGURE 12.1 Market pyramid for parallel computers. The most powerful machines—parallel computers—at the tip of the market pyramid are focused on the requirements of the most demanding applications and must harness the latest advances in technology.

puting, including the U.S. Department of Energy “ASCI” teraflops machines and the Hitachi SR2201 funded by the Japanese Ministry of Technology and Industry.

12.1.1 Evolutionary Scenario

If current technology trends hold, parallel computer architecture can be expected to follow an evolutionary path in which economic and market forces play a crucial role. Let's expand this evolutionary forecast so that we can consider how the advance of the field may diverge from this path. Currently, we see processor performance increasing by about a factor of 100 per decade (or 200 per decade if the basis is LINPACK or SpecFP). DRAM capacity also increases by about a factor of 100 per decade (quadrupling every three years). Thus, current trends would suggest that the basic balance of computing performance to storage capacity (MFLOPS/MB) of the nodes in parallel machines could remain roughly constant. This ratio varies considerably in current machines, depending on application target and cost point, but under the evolutionary scenario the family of options would be expected to continue into the future with large increases in both capacity and performance. Simply riding the commodity growth curve, we could look toward achieving petaflops-scale performance by the year 2010, or perhaps a couple of years earlier if the scale of parallelism is increased, but such systems would be in excess of \$100 million to construct. It is less clear what level of communication performance these machines will provide, for reasons that are discussed in the following. To achieve this scale of performance a lot earlier in a general-purpose system would involve an investment and

a scale of engineering that is probably not practical, although special-purpose designs may advance the time frame for a limited set of applications.

To understand what architectural directions may be adopted, the VLSI technology trends underlying the performance and capacity trends of the components are important. Microprocessor clock rates are increasing by a factor of 10–15 per decade while transistors per microprocessor increase by more than a factor of 30 per decade. DRAM cycle times, on the other hand, improve much more slowly—roughly a factor of two per decade. Thus, the gap between processor speed and memory speed is likely to continue to widen. In order to stay on the processor performance growth trend, the increase in the ratio of memory access time to processor cycle time will require that processors employ better latency avoidance and latency tolerance techniques. In addition, the increase in processor instruction rates, due to the combination of cycle time and parallelism, will demand that the bandwidth delivered by the memory increase.¹

Both of these factors—the need for latency avoidance and for high memory bandwidth—as well as the increase in on-chip storage capacity will cause the storage hierarchy to continue to become deeper and more complex. These two factors will cause the degree of dynamic scheduling of instruction-level parallelism to increase. Latency tolerance fundamentally involves allowing a large number of instructions, including several memory operations, to be in progress concurrently. Providing the memory system with multiple operations to work on at a time allows pipelining and interleaving to be used to increase bandwidth. Thus, the VLSI technology trends are likely to encourage the design of processors that are both more insulated from the memory system and more flexible so that they can adapt to the behavior of the memory system. This bodes well for parallel computer architecture because the processor component is likely to become increasingly robust to infrequent long-latency operations.

Unfortunately, neither caches nor dynamic instruction scheduling reduces the actual latency on an operation that crosses the processor chip boundary. Historically, each level of cache added to the storage hierarchy increases the cost of access to memory. (For example, we saw that the CRAY T3D and T3E designers eliminated a level of cache in the workstation design to decrease the memory latency, and the presence of a second-level on-chip cache in the T3E increased the communication latency.) This phenomenon of increasing latency with hierarchy depth is natural because designers rely on the hit being the frequent case; increasing the hit rate and reducing the hit time do more for processor performance than decreasing the miss penalty. The trend toward deeper hierarchies presents a problem for parallel architecture since communication, by its very nature, involves crossing out of the lowest level of the memory hierarchy on the node. The miss penalty contributes to the communication overhead, regardless of whether the communication abstraction is

1. Often, in observing the widening processor-memory speed gap, comparisons are made between processor rates with memory access times by taking the reciprocal of the access time. Comparing throughput and latency in this way makes the gap appear artificially wider.

shared address access or messages. Good architecture and clever programming can reduce the unnecessary communication to a minimum, but still each algorithm has some level of inherent communication. It is an open question whether designers will be able to achieve the low miss penalties required for efficient communication while attempting to maximize processor performance through deep hierarchies. There are, however, some positive indications in this direction. Many scientific applications have relatively poor cache behavior because they sweep through large data sets. Beginning with the IBM Power2 architecture and continuing in the SGI Power-Challenge and Sun UltraSparc, attention has been paid to improving the out-of-cache memory bandwidth, at least for sequential access. These efforts have proved very valuable for database applications. So we may be able to look forward to node memory structures that can sustain high bandwidth, even in the evolutionary scenario.

Indications are strong that multithreading will be utilized in future processor generations to hide the latency of local memory access. By introducing thread-level parallelism on a single processor, this direction further reduces the cost of the transition from one processor to multiple processors, thus making small-scale SMPs even more attractive on a broad extent. It also establishes an architectural direction that may yield much greater latency tolerance in the long term.

Link and switch bandwidths are increasing, although this phenomenon does not have the smooth evolution of CMOS under improving lithography and fabrication techniques. Links tend to advance through discrete technological changes. For example, copper links have transitioned through a series of driver circuits: unterminated lines carrying a single bit at a time were replaced by terminated lines with multiple bits pipelined on the wire; these may be replaced by active equalization techniques (Horowitz 1997). At the same time, links have gotten wider as connector technology has improved, allowing finer-pitched, better-matched connections, and cable manufacturing has advanced, providing better control over signal skew. For several years, it has seemed that fiber would soon take over as the technology of choice for high-speed links. However, the cost of transceivers and connectors has impeded its progress. This may change in the future, as the efficient LED arrays that have been available in gallium arsenide (GaAs) technologies become effective in CMOS. The real driver of cost reduction, of course, is volume. The arrival of gigabit Ethernet, which uses the FiberChannel physical link, may finally drive the volume of fiber transceivers up enough to cause a dramatic cost reduction. In addition, high-quality parallel fiber has been demonstrated. Thus, flexible high-performance fiber with a small physical cross section may provide an excellent link technology for some time to come.

The bandwidths that are required even for a uniprocessor design, and the number of simultaneous outstanding memory transactions needed to obtain this bandwidth, are stretching the limits of what can be achieved on a shared bus. Many system designs have already streamlined the bus design by requiring all components to transfer entire cache lines. Adapters for I/O devices are constructed with one-block caches that support the cache coherency protocol. Thus, essentially all systems will be constructed as SMPs, even if only one processor is attached. Increasingly, the bus

is being replaced by a switch and the snooping protocols are being replaced by directories. For example, the HP/Convex Exemplar uses a crossbar, rather than a bus, with the PA-8000 processor, and the Sun UltraSparc UPA has a switched interconnect within the 2–4 processor node, although these nodes are connected by a packet-switched bus in the Enterprise 6000. The IBM PowerPC-based G30 uses a switch for the datapath but still uses a shared bus for address and snoop. The SGI Origin and Sun Enterprise 10000 have gone entirely to switches. Where buses are still used, they are packet switched (split phase). Thus, even in the evolutionary scenario, we can expect to see high-performance networks integrated ever more deeply into high-volume designs. This trend makes the transition from high-volume, moderate-scale parallel systems to large-scale, moderate-volume parallel systems more attractive because less new technology is required.

Higher-speed networks are a dominant concern for current I/O subsystems as well. A great deal of attention has been paid to improved I/O support, with PCI replacing traditional vendor I/O buses. There is a very strong desire to support faster local area networks, such as gigabit Ethernet, OC-12 ATM (622 Mb/s), SCI, Fiber-Channels, and P1394.2. A standard PCI bus can provide roughly 1 Gb/s of bandwidth. Extended PCI with 64-bit, 66-MHz operation exists and promises to become more widespread in the future, offering multigigabit performance on commodity machines. Several vendors are looking at ways of providing direct memory bus access for high-performance interconnects or distributed shared memory extensions.

These trends ensure that small-scale SMPs will continue to be very attractive and that clusters and more tightly packaged collections of commodity nodes will remain a viable option for the large scale. It is very likely that these designs will continue to improve as high-speed network interfaces become more mature. We are already seeing a trend toward better integration of network interfaces with the cache coherence protocols so that control registers can be cached and DMA can be performed directly on user-level data structures (Mukherjee and Hill 1997). For many reasons, large-scale designs are likely to use SMP nodes, so clusters of SMPs are likely to be a very important vehicle for parallel computing. With the recent introduction of the CC-NUMA-based designs, such as the HP/Convex SPP, the SGI Origin, and especially the Pentium Pro-based machines, large-scale cache-coherent designs look increasingly attractive. The core question is whether a truly composable SMP-based node will emerge so that large clusters of SMPs can essentially be snapped together as easily as adding memory or I/O devices to a single node.

12.1.2 Hitting a Wall

So, if current trends hold, the evolution of parallel computer architecture looks bright. Why might this not happen? Might we hit a wall instead? There are three basic possibilities: a latency wall, an overhead wall, and a cost or power wall.

The latency wall fundamentally is the speed of light or, rather, of the propagation of electrical signals. We will soon see processors operating at clock rates in excess of 1 GHz or a clock period of less than 1 ns. Signals travel about a foot per ns. In the evolutionary view, the physical size of the node does not get much smaller; it gets

faster with more storage, but it is still several chips with connectors and PC board traces. Indeed, from 1987 to 1997, the footprint of a basic processor and memory module did not shrink much. There was an improvement from two nodes per board to about four when first-level caches moved on chip and DRAM chips were turned on their side as SIMMs, but most designs maintained one level of cache off chip. Even if the off-chip caches are eliminated (as in the CRAY T3D and T3E), the processor chip consumes more power with each generation, and a substantial amount of surface area is needed to dissipate the heat. Thus, 1,000-processor machines will still be meters across, not inches.

Although the latency wall is real, there are several reasons why it probably will not impede the practical evolution of parallel architectures in the foreseeable future. One reason is that latency tolerance techniques at the processor level are quite effective on the scale of tens of cycles. Some studies have suggested that caches are losing their effectiveness even on uniprocessors because of memory access latency (Burger, Goodman, and Kagi 1996). However, these studies assume memory operations that are not pipelined and a processor typical of mid-1990s designs. Other studies suggest that if memory operations are pipelined and if the processor is allowed to issue several instructions from a large instruction window, then branch prediction accuracy is a far more significant limit on performance than latency (Jouppi and Ranganathan 1997). With perfect prediction, such an aggressive design can tolerate memory access times in the neighborhood of 100 cycles for many applications. Multithreading techniques provide an alternative source of instruction-level parallelism that can be used to hide latency, even with imperfect branch prediction. But such latency tolerance techniques fundamentally demand bandwidth, and bandwidth comes at a cost. The cost arises either through higher signaling rates, more wires, more pins, more real estate, or some combination of these. In addition, the degree of pipelining that a component can support is limited by its occupancy. To hide latency requires careful attention to the occupancy of every stage along the path of access or communication. Where the occupancy cannot be reduced, interleaving techniques must be used to reduce the effective occupancy.

Following the evolutionary path, speed-of-light effects are likely to be dominated by bandwidth effects on latency. Currently, a single cache-line-sized transfer is several hundred bits in length, and since links are relatively narrow, a single network transaction reaches entirely across the machine with fast cut-through routing. As links get wider, the effective length of a network transaction (i.e., the number of phits) will shrink, but quite a bit of room for growth remains before it takes more than a couple of concurrent transactions per processor to cover the physical latency. Moreover, cache block sizes are increasing just to amortize the cost of a DRAM access, so the length of a network transaction and, hence, the number of outstanding transactions required to hide latency may be nearly constant as machines evolve. Explicit message sizes are likely to follow a similar trend since processors tend to be inefficient in manipulating objects smaller than a cache block.

Much of the communication latency today is in the network interface (in particular, in the store-and-forward delay at the source and at the destination) rather than in the network itself. The network interface latency is likely to be reduced as designs

mature and as it becomes a larger fraction of the network latency. Consider, for example, what would be required to cut through one or both of the network interfaces. On the source side, there is no difficulty in translating the destination to a route and spooling the message onto the wire as it becomes available from the processor. However, the processor may not be able to provide the data into the NI as fast as the NI spools data into the network; so as part of the link protocol, it may be necessary to hold back the message (transferring idle phits on the wire). This machinery is already built into most switches. Machines such as the Intel Paragon, Meiko CS-2, and CRAY T3D provide flow control all the way through the NI and back to the memory system in order to perform large block transfers without a store-and-forward delay. Alternatively, it may be possible to design the communication assist such that once a small message starts onto the wire (e.g., a cache line) it is completely transferred without delay.

Avoiding the store-and-forward delay on the destination is a bit more challenging because, in general, it is not possible to determine that the data in the message is good until the data has been received and checked. If it is spooled directly into memory, junk may be deposited. The key observation is that it is much more important that the address be correct than that the data contents be correct because we do not want to spool data into the wrong place in memory. A separate checksum can be provided on the header. The header is checked before the message is spooled into the destination node. A large transfer typically involves a completion event so that data can be spooled into memory and checked before being marked as "arrived." Note that this means that the communication abstraction should not allow applications to poll data values within the bulk transfer to detect completion. For small transfers, a variety of tricks can be played to move the data into the cache speculatively. Basically, a line is allocated in the cache and the data is transferred, but if it does not checksum correctly, the valid bit on the line is never set. Thus, greater attention will need to be paid to communication events in the design of the communication assist and memory system, but it is possible to streamline network transactions much more than the current state of the art to reduce latency.

The primary reason that parallel computers will not hit a fundamental latency wall is that overall communication latency will continue to be dominated by overhead. The latency will be there, but it will still be a modest fraction of the actual communication time. The reason for this lies deep in the current industrial design process. Where there are one or more levels of cache on the processor chip, an off-chip cache, and then the memory system, in designing a cache controller for a given level of the memory hierarchy, the designer is given a problem that has a fast side toward the processor and a slow side toward the memory. The design goal is to minimize the expression

$$\text{Average Memory Access } (S) = \text{Hit Time} \times \text{Hit Rate}_s + (1 - \text{Hit Rate}_s) \times \text{Miss Time} \quad (12.1)$$

for a typical address stream, S , delivered to the cache on the processor side.

This design goal presents an inherent trade-off because improvements in any one component generally come at the cost of worsening the others. Thus, along each direction in the design space, the optimal design point is a compromise between extremes. The Hit Time is generally fixed by the target rate of the fast component. This establishes a limit against which the rest of the design is optimized; that is, the designer will do whatever is required to keep the Hit Time within this limit. We can consider cache organizational improvements, such as higher associativity, to improve the Hit Rate, but only as long as it can be accomplished in the desired Hit Time (Przybyski, Horowitz, and Hennessy 1988). The critical aspect for parallel architecture concerns the Miss Time. How hard is the designer likely to work to drive down the Miss Time? The usual rule of thumb is to make the two additive components roughly equal. This guarantees that the design is within a factor of two of optimal and tends to be good in practice. The key point is that since Miss Rates are small for a uniprocessor, the Miss Time can be a large multiple of the Hit Time. For first-level caches with greater than 95% hit rates, it may be 20 times the Hit Time and for lower-level caches it will still be an order of magnitude. A substantial fraction of the Miss Time is occupied by the transfer to the lower level of the storage hierarchy, and small additions to this have only a modest effect on uniprocessor performance. The cache designer will utilize this small degree of freedom in many useful ways. For example, cache line sizes can be increased to improve the Hit Rate, at the cost of a longer Miss Time.

In addition, each level of the storage hierarchy adds to the cost of the data transfer because another interface must be crossed. In order to modularize the design, interfaces tend to decouple the operations on either side. There is some cost to the handshake between caches on chip; there is a larger cost in the interface between an on-chip cache and an off-chip cache and a much larger cost to the more elaborate protocol required across the memory bus. In addition, for communication, there is the protocol associated with the network itself. The accumulation of these effects is why the actual communication latency tends to be many times the lower bound imposed by the speed of light. The natural response of the designer responsible for dealing with communication aspects of a design is invariably to increase the minimum data transfer size, for example, increasing the cache line size or the smallest message fragment. This shifts the critical time from latency to occupancy. If each transfer is large enough to amortize the overhead, the additional speed-of-light latency is again a modest addition.

Wherever the design is partitioned into multiple levels of storage hierarchy with the emphasis placed on maximizing a level relative to the processor-side reference stream, the natural tendency of the designers will result in a multiplication of overhead with each level between the processor and the communication assist. In order to get close to the speed-of-light latency limit, a very different design methodology will need to be established for processor design, cache design, and memory design. One of the architectural trends that may bring about this change is the use of extensive out-of-order execution or multithreading to hide latency, even in uniprocessor systems. These techniques change the cache designer's goal. Instead of minimizing

the sum of the two components in Equation 12.1, the goal is essentially to minimize each component.

When a miss occurs, the processor does not wait for it to be serviced; it continues executing and issues more requests, many of which hit. In the meantime, the cache is busy servicing the miss. Hopefully, the miss will be complete by the time another miss is generated or the processor runs out of things it can do without the miss completing. The miss needs to be detected and dispatched for service without many cycles of processor overhead, even though it will take some time to process it. In effect, the miss needs to be handed off for processing essentially within the Hit Time budget.

Moreover, it may be necessary to sustain multiple outstanding requests to keep the processor busy, as fully explained in Chapter 11. The Miss Time may be too large for a one-to-one balance in the components of Equation 12.1 to be met, either because of latency or occupancy effects. Misses and communication events tend to cluster, so the interval between operations that need servicing is frequently much less than the average.

Little's Law suggests the existence of another potential wall—a cost wall. It says that if the total latency that needs to be hidden is L and the rate of long-latency requests is ρ , then the number of outstanding requests per processor when the latency is hidden is ρL , or greater when clustering is considered. With this number of communication events in flight, the total bandwidth delivered by the network with P processors needs to be $P\rho L(P)$, where $L(P)$ reflects the increase in latency with machine size. This requirement establishes a lower bound on the cost of the network. To deliver this bandwidth, the aggregate bandwidth of the network itself will need to be much higher, as discussed in Chapter 10, since there will be bursts, collisions, and so on. Thus, to stay on the evolutionary path, latency tolerance will need to be considered in many aspects of the system design, and network technology will need to improve in bandwidth and in cost.

12.1.3 Potential Breakthroughs

We have seen so far a rosy evolutionary path for the advancement of parallel architecture, with some dark clouds that might hinder this advance. Is there also a silver lining? Are there aspects of the technological trends that may create new possibilities for parallel computer design? The answer is certainly in the affirmative, but the specific directions are not certain. Whereas it is possible that dramatic technological changes, such as quantum devices, free space optical interconnects, molecular computing, or nanomechanical devices, are around the corner, there appears to be substantial room left in the advance of conventional CMOS VLSI devices (Patterson 1995). The simple fact of the continued increase in the level of integration is likely to bring about a revolution in parallel computer design.

From an academic viewpoint, it is easy to underestimate the importance of packaging thresholds in the process of continued integration, but history shows that these factors are dramatic indeed. The general effect of the thresholds of integration is illustrated in Figure 12.2, which shows two qualitative trends. The straight line

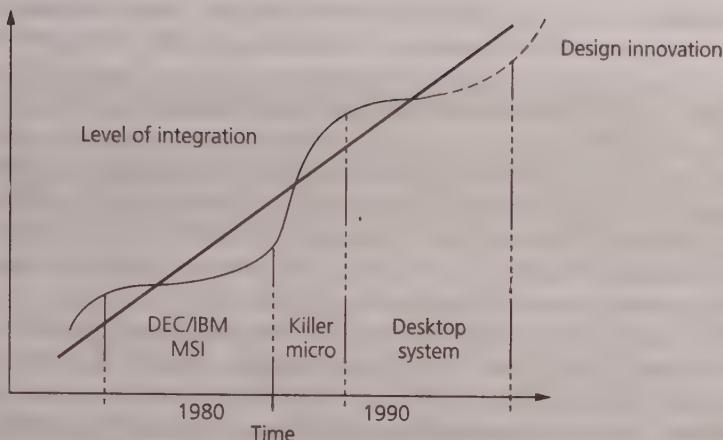


FIGURE 12.2 Tides of change. The history of computing oscillates between periods of stable development and rapid innovation. Enabling technology generally does not hit all of a sudden; it arrives and evolves. The “revolutions” occur when technology crosses a key threshold. One example is the arrival of the 32-bit microprocessor in the mid-1980s, which broke the stable hold of the less integrated minicomputers and mainframes and enabled a renaissance in computer design, including low-level parallelism on chip and high-level parallelism through multiprocessor designs. In the late 1990s, this transition is fully mature, and microprocessor-based desktop and server technology dominate all segments of the market. There is tremendous convergence in parallel machine design. However, the level of integration continues to rise, and soon the single-chip computer will be as natural as the single-board computer of the 1980s. The question is what new renaissance of design this will enable.

reflects the steady increase in the level of systems integration with time. Overlaid on this is a curve depicting the amount of innovation in system design. A given design regime tends to be stable for a considerable period in spite of technological advance, but when the level of integration crosses a critical threshold, many new design options are enabled and a design renaissance takes place. The figure shows two of the epochs in the history of computer architecture.

Recall that during the late 1970s and early 1980s computer system design followed a stable evolutionary path with clear segments: minicomputers, dominated by the DEC Vax in engineering and academic markets; mainframes, dominated by IBM in the commercial markets; and vector supercomputers, dominated by CRAY Research in the scientific market. The minicomputer had burst on the scene as a result of an earlier technology threshold where MSI and LSI components, especially semiconductor memories, permitted the design of complex systems with relatively little engineering effort. In particular, this level of integration permitted the use of microprogramming techniques to support a large virtual address space and complex instruction set. The mainframes had persisted from an earlier epoch. The vector supercomputer niche reflected the end of a transition. Its exquisite ECL circuit design, coupled with semiconductor memory in a clean load-store architecture wiped

out the earlier, more exotic parallel machine designs. These three major segments evolved in a predictable, evolutionary fashion, each with their market segment, while the microprocessor marched forward from 4 bits to 8 bits to 16 bits, bringing with it the personal computer and the graphics workstation.

In the mid-1980s, the microprocessor reached a critical threshold where a full 32-bit processor fit on a chip. Suddenly, the entire picture changed. A complete computer of significant performance and capacity fit on a board. Several such boards could be put in a system. Bus-based cache-coherent SMPs appeared from many small companies, including Synapse, Encore, Flex, and Sequent. Relatively large message-passing systems appeared from Intel, nCUBE, Ametek, Inmos, and Thinking Machines Corporation. At the same time, several minisupercomputer vendors appeared, including Multiflow, FPS, Culler Scientific, Convex, Scientific Computing System, and Cydrome. Several of these companies failed as the new plateau was established. The workstation and later the personal computer absorbed the technical computing aspect of the minicomputer market. SMP servers took over the larger-scale data centers, transaction processing, and engineering analysis, eliminating the minisuper. The vector supercomputers gave way to massively parallel microprocessor-based systems. Since that time, the evolution of designs has again stabilized. The understanding of cache coherence techniques has advanced, allowing shared address support at an increasingly large scale. The transition of scalable, low-latency networks from MPPs to conventional LAN or computer room environments has allowed casually engineered clusters of PCs, workstations, or SMPs to deliver substantial performance at very low cost, essentially as a personal supercomputer. Several very large machines are constructed as clusters of shared memory machines of various sizes. The convergence observed throughout this book is clearly in progress, and the basic design question facing parallel machine designers is how the commodity components will be integrated, not what components will be used.

Meanwhile, the level of integration in microprocessors and memories is fast approaching a new critical threshold where a complete computer fits on a chip, not a board. Microprocessors are on the way to 100 million transistors by the turn of the century. Soon after the turn of the century, the gigabit DRAM chip will arrive. This new threshold is likely to bring about a new design renaissance as profound as that of the 32-bit microprocessor of the mid-1980s, the semiconductor memory of the mid-1970s, and the integrated circuit of the 1960s. Basically, the strong differentiation between processor chips and memory chips will break down, and most chips will have processing logic and memory.

It is easy to enumerate reasons why the processor-and-memory level of integration will take place and is likely to enable dramatic change in computer design, especially parallel computer design. Several research projects are investigating aspects of this new design space, under a variety of acronyms (PIM, IRAM, C-RAM, etc.). To avoid confusion with these acronyms, let us give the processor-and-memory concept yet another acronym—PAM. Only history will reveal which old architectural ideas will gain new life and which completely new ideas will arrive. Let's look at some of the technological factors leading toward new design options.

Table 12.1 Fraction of Microprocessor Chips Devoted to Memory

Year	Micro-processor	On-Chip Cache Size	Total Transistors	Fraction Devoted to Memory	Die Area (mm ²)	Fraction Devoted to Memory
1993	Intel Pentium	I: 8 KB D: 8 KB	3.1 M	32%	~300	32%
1995	Intel Pentium Pro	I: 8 KB D: 8 KB L ₂ : 512 KB	P: 5.5 M +L ₂ : 31 M (Total: 88%)	P: 18% +L ₂ : 100% (Total: 88%)	P: 242 +L ₂ : 282 (Total: 64%)	P: 23% +L ₂ : 100% (Total: 64%)
1994	Digital Alpha 21164	I: 8 KB D: 8 KB L ₂ : 96 KB	9.3 M	77%	298	37%
1996	Digital Strong-Arm SA-110	I: 16 KB D: 16 KB	2.1 M	95%	50	61%

I: instruction; D: data; P: processor die; L₂: level 2

One clear factor is that microprocessor chips are mostly memory. It is SRAM memory used for caches, but memory nonetheless. Table 12.1 shows the fraction of the transistors and die area used for caches and memory interface, including store buffers and so on, for four recent microprocessors from two vendors (Patterson et al. 1997). The actual processor is a small and diminishing component of the microprocessor chip, even though processors are getting quite complicated. This trend is made even more clear by Figure 12.3, which shows the fraction of the transistors devoted to caches in several microprocessors over the past decade (Burger 1997).

The vast majority of the real estate and an even larger fraction of the transistors are used for data storage and organized as multiple levels of on-chip caches. This investment in on-chip storage is necessary because of the time to access off-chip memory, that is, the latency of chip interface, off-chip caches, memory bus, memory controller, and the actual DRAM. For many applications, the best way to improve performance is to increase the amount of on-chip storage.

One clear opportunity this technological trend presents is putting multiple processors on chip. Since the processor is only a small fraction of the chip real estate, the potential peak performance can be increased dramatically at a small incremental cost. The argument for this approach is further strengthened by the diminishing returns in performance for processor complexity; for example, the real estate devoted to register ports, the instruction prefetch window, and hazard detection, and bypassing each increase more than linearly with the number of instructions issued per cycle while performance improves little beyond four-way issue superscalar. Thus, for the same area, multiple processors of a less aggressive design can be employed (Olukotun et al. 1996). This motivates reexamination of sharing issues

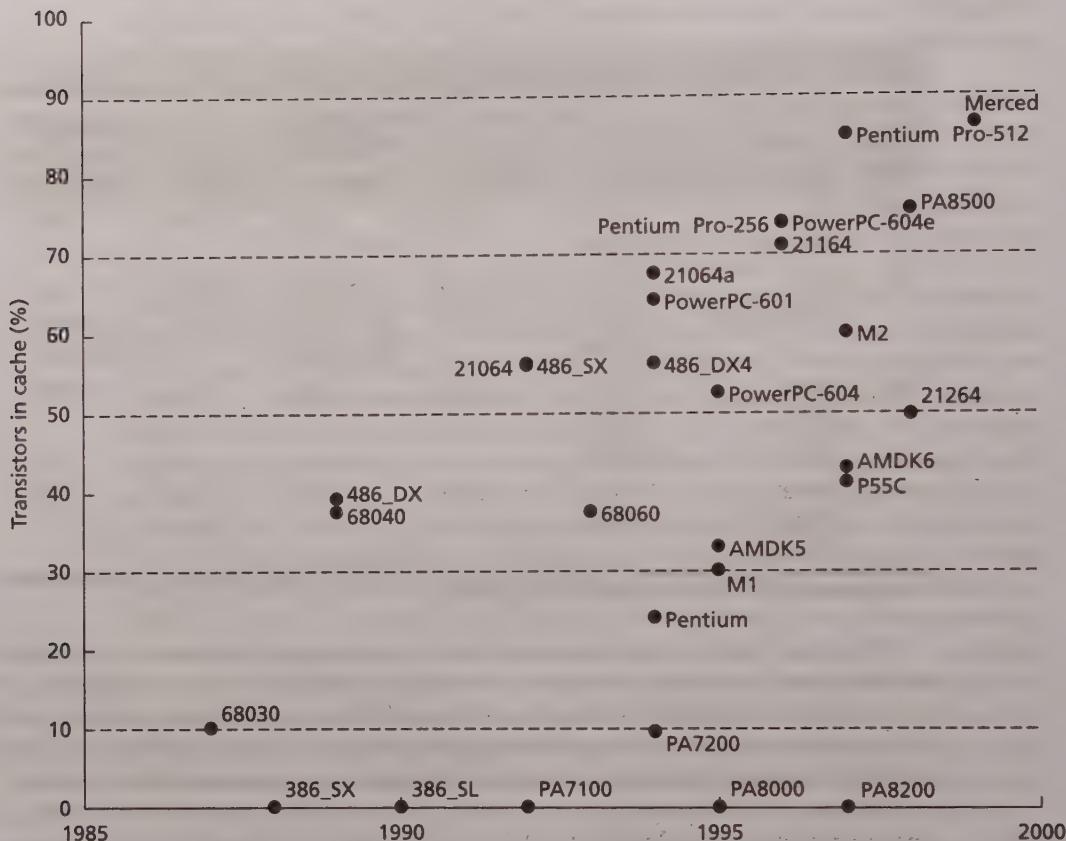


FIGURE 12.3 Fraction of transistors on microprocessor chips devoted to caches. Since caches migrated on chip in the mid-1980s, the fraction of the transistors on commercial microprocessors that are devoted to caches has risen steadily. Although the processors are complex and exploit substantial instruction-level parallelism, only by providing a great deal of local storage and exploiting locality can their bandwidth requirements be satisfied at reasonable latency.

that have evolved along with technology on SMPs since the mid-1980s. Most of the early machines shared an off-chip first-level cache, then there was room for separate caches, then L₁ caches moved on chip, and L₂ caches were sometimes shared and sometimes not. Many of the basic trade-offs remain the same in the board-level and chip-level multiprocessors: sharing caches closer to the processor allows for finer-grained data sharing and eliminates further levels of coherence support but increases access time due to the interconnect on the fast side of the shared cache. Sharing at any level presents the possibility of positive or negative interference, depending on the application usage pattern. However, the board-level designs were largely determined by the particular properties of the available components. With multiple processors on chip, all the design options can be considered within the same homo-

geneous medium. In addition, the specific trade-offs are different because of the different costs and performance characteristics. Given the broad emergence of inexpensive SMPs, especially with glueless cache coherence support, multiple processors on a chip is a natural path of evolution for multiprocessors.

A somewhat more radical change is suggested by the observation that the large volume of storage next to the processor could be DRAM, rather than SRAM, storage. Traditionally, SRAM uses the same manufacturing techniques as processors, and DRAM uses quite different techniques. The engineering requirements for microprocessors and DRAMs are traditionally very different. Microprocessor fabrication is intended to provide high clock rates and ample connectivity for datapaths and control using multiple layers of metal, whereas DRAM fabrication focuses on density and yield at minimal cost. The packages are very different. Microprocessors use expensive packages with many pins for bandwidth and materials designed to dissipate large amounts of heat. DRAM packages have few pins, low cost, and are suited to the low-power characteristics of DRAM circuits. However, these differences are diminishing. DRAM fabrication processes have become better suited for processor implementations, with two or three levels of metal and better logic speed (Saulsbury, Pong, and Nowatzky 1996).

The drive toward integrating logic into the DRAM is driven partly by necessity and partly by opportunity. The immense increase in capacity (factor of four every three years) has required that the internal organization of the DRAM and its interface change. Early designs consisted of a single, square array of bits. The address was presented in two pieces, so a row could be read from the array and then a column selected. As the capacity increased, it was necessary to place several smaller arrays on the chip and to provide an interconnect between the many arrays and the pins. In addition, with limited pins and a need to increase the bandwidth, part of the DRAM chip needs to run at a higher rate. Many modern DRAM designs, including synchronous DRAM, enhanced DRAM, and RAMBUS, make effective use of the row buffers within the DRAM chip and provide high bandwidth transfers between the row buffers and the pins. These approaches require that DRAM processes be capable of supporting logic as well. At the same time, there were many opportunities to incorporate new logic functions into the DRAM, especially for graphics support in video RAMs. For example, 3D-RAM places logic for *z*-buffer operations directly in the video RAM chip that provides the frame buffer.

The attractiveness of integrating processor and memory is very much a threshold phenomenon. Although processor design was constrained by chip area, there was certainly no motivation to use fabrication techniques other than those specialized for fast processors; and although memory chips were small, so many were used in a system that there was no justification for the added cost of incorporating a processor. However, the capacity of DRAMs has been increasing more rapidly than the transistor count or, more importantly, the area used for processors. At the gigabit DRAM or perhaps the following generation, the incremental cost of the processor is modest, perhaps 20%. From the processor designer's viewpoint, the advantage of DRAM over SRAM is that it has better density by more than an order of magnitude.

However, access time is greater, access is more restrictive, and refresh is required (Saulsbury, Pong, and Nowatzky 1996).

Somewhat more subtle threshold phenomena further increase the attractiveness of PAM. The capacity of DRAM has been growing faster than the demand for storage in most applications. The rapid increase in capacity has been beneficial at the high end because it became possible to run very large problems, which tends to reduce the communication-to-computation ratio and make parallel processing more effective. At the low end, it has had the effect of reducing the number of memory chips sold per system. When there are only a few memory chips, traditional DRAM interfaces with few pins do not work well, so new DRAM interfaces with high-speed logic are essential. When there is only one memory chip in a typical system, a huge cost savings results by bringing the processor on chip and eliminating everything in between. However, this raises a question of what the memory organization should be for larger systems.

Augmenting the impact of critical thresholds of evolving technology are new technological factors and market changes. One is high-speed CMOS serial links. Standard cells are available that will drive in excess of 1 Gb/s on a serial link, and substantially higher rates have been demonstrated in laboratory experiments. Previously, these rates were only available with expensive ECL circuits or GaAs technology. High-speed links using a few pins provide a cost-effective means of integrating PAM chips into a large system and can form the basis for the parallel machine interconnection network. A second factor is the advancement and widespread use of configurable logic technology. This makes it possible to fabricate a single building block with processor, memory, and unconfigured logic, which can then be configured to suit a variety of applications. The final factor is the development of low-power microprocessors for the rapidly growing market of network appliances, sometimes called WebPCs or Java stations, palmtop computers, and other sophisticated electronic devices. For many of these applications, modest single-chip PAMs provide ample processing and storage capacity. The huge volume of these markets may indeed make PAM the commodity building block rather than the desktop system.

The question presented by these technological opportunities is how the organizational structure of the computer node should change. The basic starting point is indicated by Figure 12.4, which shows that each of the subsystems between the processor and the DRAM bit array presents a narrow interface because pins and wires are expensive, even though they are relatively wide internally and add latency. Within the DRAM chips, the datapath is extremely wide. The bit array itself is a collection of incredibly tightly packed trench capacitors, so little can be done there. However, the data buffers between the bit array and the external interface are still wide, less dense, and essentially SRAM and logic. Recall that when a DRAM is read, a portion of the address is used to select a row, which is read into the data buffer, and then another portion of the address is used to select a few bits from the data buffer. The buffer is written back to the row since the read is destructive. On a write, the row is read, a portion is modified in the buffer, and eventually it is written back.

Current research investigates three basic restructuring possibilities, each of which has substantial history and can be understood, explored, and evaluated in terms of

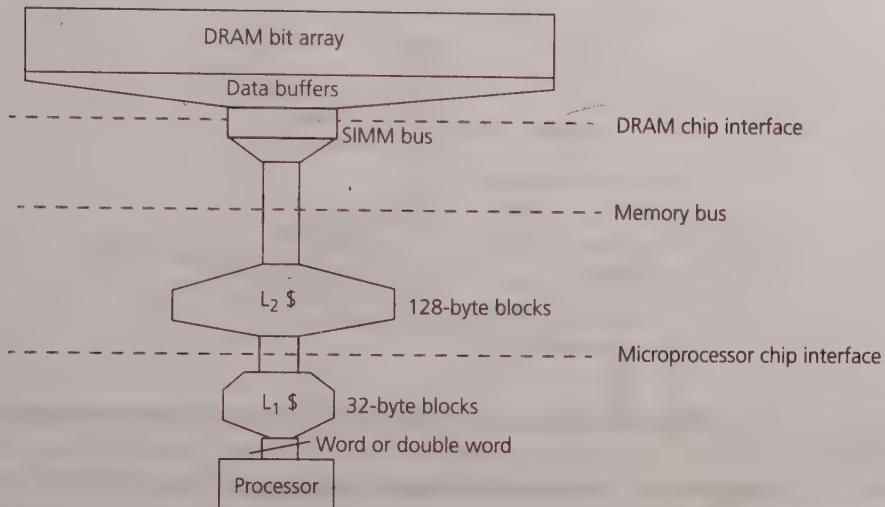


FIGURE 12.4 Bandwidths across a computer system. The processor datapath is several words wide but typically has a couple word-wide interface to its L₁ cache. The L₁ cache blocks are 32 or 64 bytes wide, but they are constrained between the processor word-at-a-time operation and the microprocessor chip interface. The L₂ cache blocks are even wider, but they are constrained between the microprocessor chip interface and the memory bus interface, both width critical. The SIMMS that form a bank of memory may have an interface that is wider and slower than the memory bus. Internally, this is a small section of a very wide data buffer, which is transferred directly to and from the actual bit arrays.

the fundamental design principles put forward in this book. They are surveyed briefly here, but the reader will surely want to consult the most recent literature and the Web.

The first option is to place simple, dedicated processing elements into the logic associated with the data buffers of more or less conventional DRAM chips, indicated in Figure 12.5. This approach has been called *processor-in-memory* (PIM) (Gokhale, Holmes, and Iobst 1995) and *Computational RAM* (Kogge 1994; Elliot, Snelgrove, and Stumm 1992). It is fundamentally SIMD processing of a restricted class of data parallel operations. Typically, these will be small bit-serial processors providing basic logic operations, but they could operate on multiple bits or even a word at a time. As we saw in Chapter 1, the approach has appeared several times in the history of parallel computer architecture. Usually it appears at the beginning of a technological transition when a general-purpose operation is not quite feasible, so the specialized operation enjoys a generous performance advantage. Each time it has proved applicable for a limited class of operations, usually image processing, signal processing, or dense linear algebra, and each time it has given way to more general-purpose solutions as the underlying technology evolves.

For example, in the early 1960s, there were numerous SIMD machines proposed that would allow construction of a high-performance machine by replicating only the function units and sharing a single-instruction sequencer, including Staran,

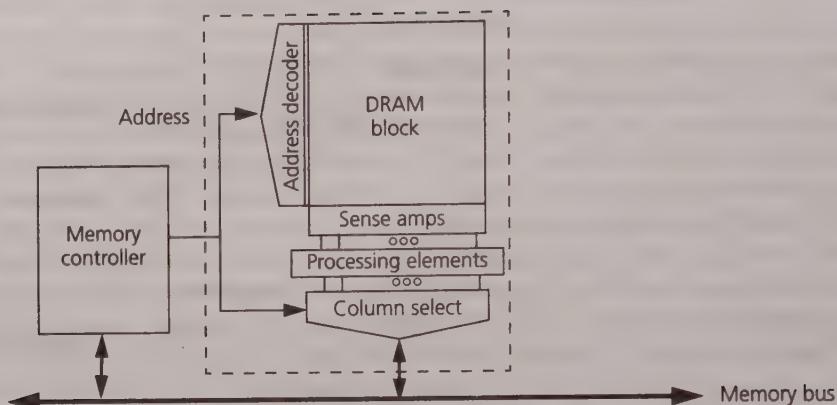


FIGURE 12.5 Processor-in-memory organization. Simple function units (processing elements) are incorporated into the data buffers of fairly conventional DRAM chips.

PEPE, and Illiac. The early effort culminated in the development of the Illiac IV at the University of Illinois, which took many years and became operational only months before the CRAY-1. In the early 1980s, the approach appeared again with the ICL DAP, which provided an array of compact processors and got a big boost in the mid-1980s when processor chips got large enough to support 32-bit serial processors but not a full 32-bit processor. The Goodyear MPP, Thinking Machines CM-1, and MasPar grew out of this window of opportunity. The key recognition that made the latter two machines much more successful than any previous SIMD approach was the need to provide a general-purpose interconnect between the processing elements, not just a low-dimensional grid, which is clearly cheap to build. Thinking Machines also was able to capture the arrival of the single-chip floating-point unit and to modify the design in the CM-2 to provide operations on 2-K 32-bit PEs rather than 64-K 1-bit PEs. However, these designs were fundamentally challenged by Amdahl's Law since the high-performance mode could only be applied on the fraction of the problem that fits the specialized operations. Within a few years, they yielded to MPP designs with a few thousand general-purpose microprocessors, which could perform SIMD operations and more general operations so that the parallelism could be utilized more of the time. The PIM approach was deployed in the CRAY 3/SSS (before the company filed Chapter 11) to provide special support for the National Security Agency. It has also been demonstrated for more conventional technology (Aimoto et al. 1996; Shimuzu et al. 1996).

A second restructuring option is to enhance the data buffers associated with banks of DRAM so they can be used as vector registers, as in Figure 12.6 (Patterson et al. 1997). There can be high-bandwidth transfers between DRAM rows and vector registers using the width of the bit arrays, but arithmetic is performed on vector registers by streaming the data through a small collection of conventional function

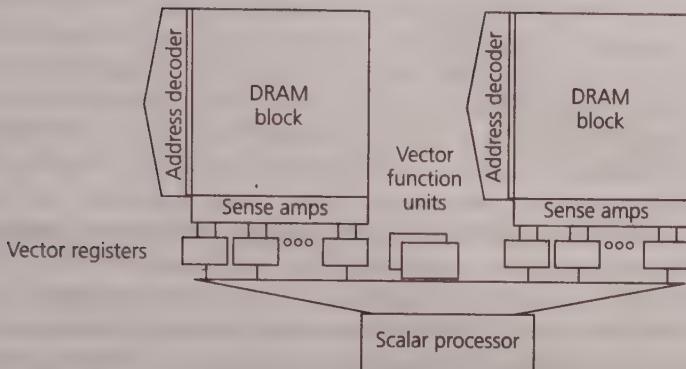


FIGURE 12.6 Vector DRAM organization. DRAM data buffers are enhanced to provide vector registers, which can be streamed through pipelined function units. On-chip memory system and vector support is interfaced to a scalar processor, possibly with its own caches.

units. This approach also rests on a good deal of history, including the very successful CRAY machines as well as several unsuccessful attempts. The CRAY-1 success was startling in part because of the contrast with the unsuccessful CDC Star-100 vector processor a year earlier. The Star-100 internally operated on short segments of data that were streamed out of memory into temporary registers. However, it provided vector operations only on contiguous (stride 1) vectors, and the core memory it employed had long latency. The success of the CRAY-1 was not just a matter of exposing the vector registers but a combination of its use of fast semiconductor memory, providing a general interconnect between the vector registers and many banks of memory so that nonunit stride and later gather/scatter operations could be performed, and a very efficient coupling of the scalar and vector operations. In other words, it provided low latency and high bandwidth for general access patterns and efficient synchronization. Several later attempts at this style of design in newer technologies failed to appreciate these lessons completely, including the FPS-DMAX, which provided linear combinations of vectors in memory; the Stellar, Ardent, and Stardent vector workstations; the Star-100 vector extension to the SparcStation; the vector units in the memory controllers of the CM-5; and the vector function units on the Meiko CS-2. It is easy to become enamored with the peak performance and low cost of the special case, but if the start-up costs are large, addressing capability is limited, or interaction with scalar access is awkward, the fraction of time that the extension is actually used drops quickly and the approach is vulnerable to the general-purpose solution.

The third option pursues a general-purpose design but removes the layers of abstraction that have been associated with the distinct packaging of processors, off-chip caches, and memory systems. Basically, the data buffers associated with the DRAMs are utilized as the final layer of caches. This is quite attractive since advanced DRAM designs have essentially been using the data buffers as caches for several years. However, in the past, they were constrained by the narrow interface

out of the DRAM chips and the limited protocols of the memory bus. In the more integrated design, the DRAM buffer caches can interface directly to the higher-level caches of one or more on-chip processors. This approach changes the basic cache design trade-offs somewhat, but conventional analysis techniques apply. The cost of long lines is reduced since the transfer between the data buffer and the bit array is performed in parallel. The current high cost of logic near the DRAM tends to place a higher cost on associativity. Thus, many approaches use direct-mapped DRAM buffer caches and employ victim buffers or analogous techniques to reduce the rate of conflict misses (Saulsbury, Pong, and Nowatzky 1996). When these integrated designs are used as a building block for parallel machines, the expected effects are observed of long cache blocks causing increased false sharing along with improved data prefetching when spatial locality is present (Nayfeh, Hammond, and Olukotun 1996).

When the PAM approach moves from the stage of academic, simulation-based study to active commercial development, we can expect to see an even more profound effect arising from the change in the design process. No longer is a cache designer in a position of optimizing one step in the path, constrained by a fast interface on one side and a slow interface on the other. The boundaries will have been removed, and the design can be optimized as an end-to-end problem. All of the possibilities we have seen for integrating the communication assist are present: at the processor, into the cache controller, or into the memory controller; but in PAM they can be addressed in a uniform framework rather than within the particular constraint of each component of the system.

However the detailed design of integrated processor and memory components shakes out, these components are likely to provide a new, universal building block for larger-scale parallel machines. Clearly, collections of them can be connected together to form distributed-memory machines, and the communication assist is likely to be much better integrated with the rest of the design since it is all on one chip. In addition, the interconnect pins are likely to be the only external interface, so communication efficiency will be even more critical. It will clearly be possible to build parallel machines on a scale far beyond what has been possible. A practical limit that has stayed roughly constant since the earliest computers is that large-scale systems are limited to about 10,000 components. Larger systems have been built but tend to be difficult to maintain. In the early days, it was 10,000 vacuum tubes, then 10,000 gates, then 10,000 chips. Recently, large machines have had about 1,000 processors and each processor required about 10 components, either chips or memory SIMMS. The first teraflops machine has almost 10,000 processors, each with multiple chips, so we will see if the pattern has changed. The complete system on a chip may well be the commodity building block of the future, used in all sorts of intelligent appliances at a volume much greater than the desktop and hence at a much lower cost. In any case, we can look forward to much larger-scale parallelism as the processors per chip continue to rise.

A very interesting question is what happens when programs need access to more data than fits on the PAM. One approach is to provide a conventional memory interface for expansion. A more interesting alternative is to simply provide cache-

coherent access to other PAM chips. The techniques are now very well understood, as is the minimal amount of hardware support required to provide this functionality. If the processor component of the PAM is indeed small, then even if only one process in the system is ever used, the additional cost of the other processors sitting near the memories is small. Since parallel software is becoming more and more widespread, the extra processing power is there when applications demand it.

12.2

APPLICATIONS AND SYSTEM SOFTWARE

Clearly, the future of parallel computer architecture will increasingly be a story of parallel software and of hardware/software interactions. We can view parallel software as falling into five different classes: applications, compilers, languages, operating systems, and tools. In parallel software too, the same basic categories of change apply: evolution, hitting walls, and breakthroughs.

12.2.1

Evolutionary Scenario

Whereas data management and information processing are likely to be the dominant applications that exploit multiprocessors, applications in science and engineering have always been the proving ground for high-end computing. Early parallel scientific computing focused largely on models of physical phenomena that result in fairly regular computational and communication characteristics. This allowed simple partitionings of the problems to be successful in harnessing the power of multiprocessors, just as it led earlier to effective exploitation of vector architectures. As the understanding of both the application domains and parallel computing grew, early adopters began to model the more complex, dynamic, and adaptive aspects that are integral to most physical phenomena, leading to applications with irregular, unpredictable characteristics. This trend is expected to continue, bringing with it the attendant complexity for effective parallelization.

As multiprocessing becomes more and more widespread, the domains of its application will evolve, as will the applications whose characteristics are most relevant to computer manufacturers. Large optimization problems encountered in finance and logistics—for example, determining good crew schedules for commercial airlines—are very expensive to solve, have high payoff for corporations, and are amenable to scalable parallelism. Methods developed under the fabric of artificial intelligence, including searching techniques and expert systems, are finding practical use in several domains and can benefit greatly from increased computational power and storage. In the area of information management, an important direction is toward increased use of extracting trends and inferences from large volumes of data, using data mining and decision support techniques (in the latter, complex queries are made to determine trends that will provide the basis for important decisions). These applications are often computationally intensive as well as database intensive and mark an interesting marriage of computation and data storage/retrieval to build computation-and-information servers. Such problems are increasingly encountered in scientific research areas as well, for example, in manipulating and analyzing the

tremendous volumes of biological sequence information that is rapidly becoming available through the sciences of genomics and sequencing and in computing on the discovered data to produce estimates of three-dimensional structure. The rise of wide-scale distributed computing—enabled by the Internet and the World Wide Web—and the abundance of information in modern society make this marriage of computing and information management all the more inevitable as a direction of rapid growth. Multiprocessors have already become servers for Internet search engines and World Wide Web queries. The nature of the queries coming to an information server may also span a broader range, from a few, very complex mining and decision support queries at a time to a staggeringly large number of simultaneous queries in the case where the clients are home computers or handheld information appliances.

As the communication characteristics of networks improve and the need for parallelism with varying degrees of coupling becomes stronger, we are likely to see a strong convergence of parallel and distributed computing. Techniques from distributed computing will be applied to parallel computing to build a greater variety of information servers that can benefit from the better performance characteristics “within the box,” and parallel computing techniques will be employed in the other direction to use distributed systems as platforms for solving a single problem in parallel. Multiprocessors will continue to play the role of servers—database and transaction servers, compute servers, and storage servers—though the data types manipulated by these servers are becoming much richer and more varied. From information records containing text, we are moving to an era where images, three-dimensional models of physical objects, and segments of audio and video are increasingly stored, indexed, queried, and served out as well. Matches in queries to these data types are often approximate, and serving them out often uses advanced compression techniques to conserve bandwidth, both of which require computation as well.

Finally, the increasing importance of graphics, media, and real-time data from sensors—for military, civilian, and entertainment applications—will lead to increasing significance of real-time computing on data as it streams in and out of a multiprocessor. Instead of reading data from a storage medium, operating on it, and storing it back, this may require operating on data on its way through the machine between input and output data ports. How processors, memory, and networks integrate with one another, as well as the role of caches, may have to be revisited in this scenario. As the application space evolves, we will learn what kinds of applications can truly utilize large-scale parallel computing and what scales of parallelism are most appropriate for others. We will also learn whether scaled-up general-purpose architectures are appropriate for the highest end of computing or whether the characteristics of these applications are so differentiated that they require substantially different resource requirements and integration to be cost-effective.

As parallel computing is embraced in more domains, we begin to see portable, prepackaged parallel applications that can be used in multiple application domains. A good example of this is an application called Dyna3D that solves systems of partial differential equations on highly irregular domains, using a technique called the

finite element method. Dyna3D was initially developed at government research laboratories for use in modeling weapons systems on exotic, multimillion-dollar supercomputers. After the cold war ended, the technology transitioned into the commercial sector, and it became the mainstay of crash modeling in the automotive industry and elsewhere on widely available parallel machines. By the time this book was written, the code was widely used on cost-effective parallel servers for performing simulations on everyday appliances, such as what happens to a cellular phone when it is dropped.

A major enabling factor in the widespread use of parallel applications has been the availability of portable libraries that implement programming models on a wide range of machines. With the advent of the message-passing interface (MPI), this has become a reality for message passing, which is used in Dyna3D: the application of Dyna3D to different problems is usually done on very different platforms, from machines designed for message passing (like the Intel Paragon) to machines that support a shared address space in hardware (like the CRAY T3D) to networks of workstations. Similar portability across a wide range of communication architecture performance characteristics has not yet been achieved for a shared address space programming model but is likely soon.

As more applications are coded in both the shared address space and message-passing models, we find a greater separation of the algorithmic aspects of creating a parallel program (decomposition and assignment) from the more mechanical and architecture-dependent aspects (orchestration and mapping), with the former being relatively independent of the programming model and architecture. Galaxy simulations and other hierarchical n -body computations, like Barnes-Hut, provide an example. The early message-passing parallel programs used an orthogonal recursive bisection method to partition the computational domain across processors and did not maintain a global tree data structure. Shared address space implementations, on the other hand, used a global tree and a different partitioning method that led to a very different domain decomposition. With time and with the improvement in message-passing communication architectures, the message-passing versions also evolved to use similar partitioning techniques to the shared address space version, including building and maintaining a logically global tree using hashing techniques. Similar developments have occurred in ray tracing, where parallelizations that assumed no logical sharing of data structures gave way to logically shared data structures that were implemented in the message-passing model using hashing. A continuation of this trend will also contribute to the portability of applications between systems that preferentially support one of the two models.

Like parallel applications, parallel programming languages are also evolving apace. The most popular languages for parallel computing continue to be based on the most popular sequential programming languages (C, C++, and Fortran), with extensions for parallelism. The nature of these extensions is now increasingly driven by the needs observed in real applications. In fact, it is not uncommon for languages to incorporate features that arise from experience with a particular class of applications and then are found to generalize to some other classes of applications as well. In a similar vein, portable libraries of commonly occurring data structures and

algorithms for parallel computing are being developed, and templates for these are being defined for programmers to customize according to the needs of their specific applications.

Several styles of parallel languages have been developed. The least common denominator is MPI for message-passing programming, which has been adopted across a wide range of platforms, and a sequential language enhanced with primitives like the ones we discussed in Chapter 2 for a shared address space. Many of the other language systems try to provide the programmer with a natural programming model, often based on a shared address space, and hide the properties of the machine's communication abstraction through software layers. One major direction has been explicitly parallel object-oriented languages, with emphasis on appropriate mechanisms to express concurrency and synchronization in a manner integrated with the underlying support for data abstraction. Another has been the development of data parallel languages with directives for partitioning, such as high-performance Fortran, that are currently being extended to handle irregular applications. A third direction has been the development of implicitly parallel languages. Here, the programmer is not responsible for specifying the assignment, orchestration, or mapping, but only for the decomposition into tasks and for specifying the data that each task accesses. Based on these specifications, the run-time system of the language determines the dependences among tasks and assigns and orchestrates them appropriately for parallel execution on a platform. The burden on the programmer is decreased, or at least localized, but the burden on the system is increased. With the increasing importance of data access for performance, several of these languages have proposed language mechanisms and run-time techniques to divide the burden of achieving data locality and reducing communication between the programmer and the system in a reasonable way.

Finally, the increasing complexity of the applications being written and the phenomena being modeled by parallel applications has led to the development of languages to support composability of bigger applications from smaller parts. We can expect much continued evolution in all these directions—appropriate abstractions for concurrency, data abstraction, synchronization, and data management; libraries and templates; explicit and implicit parallel programming languages—with the goal of achieving a good balance between ease of programming, performance, and portability across a wide range of platforms (in both functionality and performance).

The development of compilers that can automatically parallelize programs has also been evolving over the last decade or two. With significant developments in the analysis of dependences among data accesses, compilers are now able to automatically parallelize simple array-based Fortran programs and achieve respectable performance on small-scale multiprocessors. Advances have also been made in compiler algorithms for managing data locality in caches and main memory and in optimizing the orchestration of communication given the performance characteristics of an architecture (for example, making communication messages larger for message-passing machines). However, compilers are still not able to parallelize more complex programs effectively, especially those that make substantial use of pointers. Since the address stored in a pointer is not known at compile time, it is very difficult

to determine whether a memory operation made through a pointer is to the same or a different address than another memory operation. In acknowledging this difficulty, one approach that is increasingly being taken is that of interactive parallelizing compilers. Here, the compiler discovers the parallelism that it can and then gives intelligent feedback to the user about the places where it failed and asks the user questions about choices it might make in decomposition, assignment, and orchestration. Given the directed questions and information, a user familiar with the program may have or may discover the higher-level knowledge of the application that the compiler did not have. Another approach in a similar vein is integrated compile-time and run-time parallelization tools, in which information gathered at run time may be used to help the compiler—and perhaps the user—parallelize the application successfully. Compiler technology will continue to evolve in these areas as well as in the simpler area of providing support to deal with relaxed memory consistency models.

Operating systems are making the transition from uniprocessors to multiprocessors and from multiprocessors used as batch-oriented compute engines to multiprogrammed servers of computational and storage resources. In the former category, the evolution has included making operating systems more scalable by reducing the serialization within the operating system and making the scheduling and resource management policies of operating systems take spatial and temporal locality into account (for example, not moving processes around too much in the machine; having them scheduled close to the data they access; and having them scheduled, as far as possible, on the same processor every time). In the latter category, a major challenge is managing resources and process scheduling in a way that strikes a good balance between fairness and performance, just as was done in uniprocessor operating systems. Attention is paid to data locality in scheduling application processes in a multiprogrammed workload as well as to parallel performance in determining resource allocation. For example, the relative parallel performance of applications in a multiprogrammed workload may be used to determine how many processors and other resources to allocate to it at the cost of other programs. Operating systems for parallel machines are also increasingly incorporating the characteristics of multiprogrammed mainframe machines that were the mainstay servers of the past. One challenge in this area is exporting to the user the image of a single operating system (called single-system image) while still providing the reliability and fault tolerance of a distributed system. Other challenges include containing faults so that only the faulting application or the resources that the faulting application uses are affected by a fault and providing the reliability and availability that people expect from mainframes. This evolution is necessary if scalable microprocessor-based multiprocessors are to truly replace mainframes as “enterprise” servers for large organizations, running multiple applications at a time.

With the increasing complexity of application-system interactions and the increasing importance of the memory and communication systems for performance, it is very important to have good tools for diagnosing performance problems. This is particularly true of a shared address space programming model since communication there is implicit, and artifactual communication can often dominate other

performance effects. Contention is a particularly prevalent and difficult performance effect for a programmer to diagnose, especially because the point in the program (or machine) where the effects of contention are felt can be quite different from the point that causes the contention. Performance tools continue to evolve, providing feedback about where the largest overhead components of execution time are in the program, what data structures might be causing the data access overheads, and so on. We are likely to see progress in techniques to increase the visibility of performance monitoring tools, which will be helped by the recent increasing willingness of machine designers to add a few registers or counters at key points in a machine solely for the purpose of performance monitoring. We are also likely to see progress in the quality of the feedback that is provided to the user, ranging from where in the program code a lot of time is being spent stalled on data access (available today) to which data structures are responsible for the majority of this time to what the cause of the problem is (communication overhead, capacity misses, conflict misses with another data structure, or contention). The more detailed the information and the better it is cast in terms of the concepts the programmer deals with rather than in machine terms, the more likely that a programmer can respond to the information and improve performance. Evolution along this path will hopefully bring us to a good balance between software and hardware support for performance diagnosis.

A final aspect of the evolution will be the continued integration of the system software pieces described in the preceding. Languages will be designed to make the compiler's job easier in discovering and managing parallelism and to allow more information to be conveyed to the operating system to make its scheduling and resource allocation decisions.

12.2.2 Hitting a Wall

On the software side, there is a wall that we have been hitting up against for many years now, which is the wall of programmability. While programming models are becoming more portable, architectures are converging, and good evolutionary progress is being made in many areas, it is still the case that parallel programming is much more difficult than sequential programming. Programming for good performance takes a lot of work, sometimes in determining a good parallelization and other times in implementing and orchestrating it. Even debugging parallel programs for correctness is an art or at best a primitive science. The parallel debugging task is difficult because of the interactions among multiple processes with their own program orders and because of sensitivity to timing. Depending on when events in one process happen to occur relative to events in another process, a bug in the program may or may not manifest itself at run time in a particular execution. And if it does, instrumenting the code to monitor certain events can cause the timing to be perturbed in such a way that the bug no longer appears.

Although evolutionary progress has greatly increased the adoption of parallel computing, overcoming the wall will take a breakthrough that will truly allow parallel computing to realize the potential afforded to it by technology and architectural trends. It is unclear whether this breakthrough will be in languages per se, or in pro-

gramming methodology, or whether it will simply be evolutionary improvements crossing a critical threshold.

12.2.3 Potential Breakthroughs

Other than breakthroughs in parallel programming languages or methodology and parallel debugging, we may hope for a breakthrough in performance models for reasoning about parallel programs. Although many models have been quite successful at exposing the essential performance characteristics of a parallel system (Valiant 1990; Culler et al. 1996) and some have even provided a methodology for using these parameters, the more challenging aspect is modeling the properties of complex parallel applications and their interactions with the system parameters. There is not yet a well-defined methodology for programmers or algorithm designers to use for this purpose in order to determine how well an algorithm will perform in parallel on a system or which among competing partitioning or orchestration approaches will perform better. Another breakthrough may come from architecture if we can somehow design machines in a cost-effective way that makes it much less important for a programmer to worry about data locality and communication; that is, to truly design a machine that can look to the programmer like a PRAM. An example of this would be if all latency incurred by the program could be tolerated by the architecture. However, this is likely to require tremendous bandwidth, which has a high cost, and it is not clear how best to invest the application concurrency for a mix of parallel execution and latency tolerance.

The ultimate breakthrough, of course, will be the complete success of parallelizing compilers in taking a wide range of sequential programs and converting them into efficient parallel executions on a given platform, achieving good performance at a scale close to that inherently afforded by the application (i.e., close to the best you could do by hand). Besides the problems discussed, parallelizing compilers tend to look for and utilize low-level, localized information in the program and are not currently good at performing high-level, global analysis transformations. Compilers also lack semantic information about the application; for example, if a particular sequential algorithm for a phase of a problem does not parallelize well, there is nothing the compiler can do to choose another one. And for a compiler to take data locality and artificial communication into consideration and manage the extended memory hierarchy of a multiprocessor is very difficult. However, even effective programmer-assisted compiler parallelization, keeping the programmer involvement to a minimum, would be perhaps the most significant software breakthrough in making parallel computing truly mainstream.

Whatever the future holds, it is certain that the continuing evolutionary advance will cross critical thresholds; significant walls will be encountered, but there are likely to be ways around them and unexpected breakthroughs will occur. Parallel computing will remain the place where exciting changes in computer technology and applications are first encountered in an ever evolving cycle of hardware/software interactions.

Appendix: Parallel Benchmark Suites

Despite the difficulty of identifying “representative” parallel applications and the immaturity of parallel software (languages and programming environments), the quantitative workload-driven evaluation of machines and architectural ideas must go on. To this end, several suites of parallel “benchmarks” have been developed and distributed. The term *benchmarks* is in quotations because of the difficulty of relying on a single suite of programs to make definitive performance claims in parallel computing. Benchmark suites for multiprocessors vary in the kinds of application domains and run-time characteristics they cover; whether they include toy programs, kernels, or real applications; the communication abstractions they are targeted for; and their philosophy toward benchmarking or architectural evaluation. What follows is a discussion of some of the most widely used, publicly available benchmark/application suites for parallel processing. Table A.1 shows how these suites can currently be obtained.

A.1 SCALAPACK

The ScaLapack suite (Dongarra and Walker 1995; Choi et al. 1992) consists of parallel, message-passing implementations of the LAPACK linear algebra kernels. The LAPACK suite includes routines to solve linear systems of equations, eigenvalue problems, singular value problems, matrix multiplications, matrix factorizations, and eigenvalue solvers. Information about the ScaLapack suite and the suite itself can be obtained from the Netlib repository maintained at the Oak Ridge National Laboratories (see Table A.1).

A.2 TPC

The Transaction Processing Performance Council (TPC), founded in 1988, has created a set of publicly available benchmarks, called TPC-A, TPC-B, TPC-C, and TPC-D that are representative of different kinds of inputs and queries to transaction processing and database system programs (Transaction Processing Council 1998). While database and transaction processing workloads are very important in actual usage of parallel machines, source codes for these programs are almost impossible to obtain because of their competitive value to their developers.

Table A.1 Obtaining Public Benchmark and Application Suites

Benchmark Suite	Communication Abstraction	Domain of Application	How to Obtain Code or Information
Scalapack	Message passing	Scientific	http://www.netlib.org
TPC	Either (not provided parallel)	Database/transaction processing	http://www.tpc.org
SPLASH-2	Shared address space (CC)	Scientific/engineering/graphics	http://www-flash.stanford.edu/apps/SPLASH
SPLASH-3	Shared address space (CC)	Varied	http://www.cs.princeton.edu/prism/splash-3
NAS	Either (paper and pencil)	Scientific	http://www.nas.nasa.gov
NPB2	Message passing	Scientific	http://science.nas.nasa.gov/Software/NPB
PARKBENCH	Message passing	Scientific	netlib@ornl.gov

TPC has a rigorous policy for reporting results. They use two metrics: (1) throughput in transactions per second, subject to a constraint that over 90% of the transactions have a response time less than a specified threshold; and (2) cost-performance in price per transaction per second, where price is the total system price and maintenance cost for five years. The benchmarks scale with the power of the system in order to measure it realistically.

The first TPC benchmark was TPC-A, consisting of a single, simple, update-intensive transaction. It was intended to provide a simple, repeatable unit of work designed to exercise the key features of an on-line transaction processing (OLTP) system, such as that of a bank's customer records or an airline reservation system. The chosen banking transaction consisted of reading 100 bytes from a terminal, updating the account, branch, and teller records, writing a history record, and finally writing 200 bytes to a terminal. TPC-A is no longer used.

TPC-B is a more centralized database (not OLTP) benchmark designed to exercise the system components necessary for update-intensive database transactions. It therefore has significant disk I/O but moderate system and application execution time, and it requires transaction integrity. Unlike OLTP, it does not require terminals or networking. These first two benchmarks were declared obsolete in 1995 and have been wholly replaced by TPC-C and now TPC-D.

TPC-C was approved in 1992. It is designed to be more realistic than TPC-A but to carry over many of its characteristics. TPC-C is a multiuser benchmark and requires a remote terminal emulator to emulate a population of users with their terminals. It models the activity of a wholesale supplier with a number of geographically distributed sales districts and supply warehouses, including customers placing orders, making payments, or making inquiries, as well as deliveries and inventory checks. The database size scales with the throughput of the system. TPC-C has a

more complex database structure compared to TPC-A, multiple transaction types of varying complexity, on-line and deferred execution modes, higher levels of contention on data access and update, patterns that simulate hot spots, access by primary as well as nonprimary keys, realistic requirements for full-screen terminal I/O and formatting, requirements for full transparency of data partitioning, and transaction rollbacks.

TPC-D is a decision support benchmark. According to the TPC, decision support describes a system's capability to support the formulation of business decisions through complex queries against a database. The queries access large portions of the database, not individual records only (as in OLTP) and include operations like multitable joins, extensive sorting, grouping and aggregations, and sequential scans. While OLTP (TPC-C) consists of small, mostly update transactions, decision support queries are large, individually time consuming, and read-only on the database. Decision support databases are updated only infrequently, either by periodic batch runs or by background "trickle" update activity. These update activities are also included in TPC-D. TPC-D models ad hoc queries like determining sales trends, as opposed to regular business operations in TPC-C, and has only a few concurrent users.

TPC plans to add more benchmarks, including an enterprise server benchmark, which provides concurrent OLTP and batch transactions as well as heavyweight read-only OLTP transactions with tighter response time constraints. A client-server benchmark is also under consideration. Information about all these benchmarks can be obtained by contacting the Transaction Processing Performance Council (see Table A.1).

A.3 SPLASH

The SPLASH (Stanford ParalleL Applications for Shared Memory) suite (Singh, Weber, and Gupta 1992) was originally developed at Stanford University to facilitate the evaluation of architectures that support a shared address space with coherent replication. It was replaced by the SPLASH-2 suite (Woo et al. 1995), which enhanced some applications and added several more, broadening the coverage of domains and characteristics substantially. The SPLASH-2 suite currently contains seven complete applications and five computational kernels. Some of the applications and kernels are provided in different versions, with different levels of optimization in the way data structures are designed and used (see the discussion of levels of optimization in Section 4.2.2). The programs represent various computational domains, mostly scientific and engineering applications and computer graphics. They are all written in C and use the Parmacs macro package from Argonne National Laboratories (Boyle et al. 1987) for parallelism constructs. Their characteristics together with methodological guidelines for using them can be found in (Woo et al. 1995). All of the parallel programs used for workload-driven evaluation of shared address space machines in this book come from the SPLASH-2 suite. The suite and its documentation can be obtained as described in Table A.1. The designers of the

SPLASH suites plan to add new shared address space programs from a variety of application domains in a new release called SPLASH-3.

A.4 NAS PARALLEL BENCHMARKS

The NAS benchmarks (Bailey et al. 1991, 1995) were developed by the Numerical Aerodynamic Simulation group at the National Aeronautic and Space Administration (NASA). They are a set of eight computations—five kernels and three pseudo-applications (not complete applications but more representative than the kernels of the kinds of data structures, data movement, and computation required in real aerophysics applications). The computations are each intended to focus on some important aspect of the types of highly parallel computations encountered in aerophysics applications. The kernels include an embarrassingly parallel computation, a multigrid equation solver, a conjugate gradient equation solver, a three-dimensional FFT equation solver, and an integer sorting program. Two different data sets, one small and one large, are provided for each benchmark. The benchmarks are intended to evaluate and compare real machines against one another, and an elaborate reporting and validation policy is in place.

The original NAS benchmarks take a different approach to benchmarking than the other suites described here. Those suites provide programs that are already written in a high-level language (such as Fortran or C, with constructs for parallelism). The NAS benchmarks, on the other hand, originally did not provide parallel implementations but were so-called paper-and-pencil benchmarks. They specified the problem to be solved in complete detail (the equation system and constraints, for example) and the high-level method to be used (multigrid or conjugate gradient method, for example) but did not provide the parallel program to be used. Instead, they left it up to the user to use the best parallel implementation for the machine at hand. The user is free to choose any language constructs for parallelism (though the language must be an extension of Fortran or C), data structures, communication abstractions and mechanisms, processor mapping, memory allocation and usage, and low-level optimizations (with some restrictions on the use of assembly language). The motivations for this approach to benchmarking are that since parallel architectures are diverse in their performance characteristics and the programming model toward which they are biased, and since no established dominant programming language or communication abstraction is most efficient on all architectures, a parallel program implementation that is best suited to one machine may not be appropriate for another. If we want to compare two machines using a given computation or benchmark, we should use the most appropriate implementation for each machine. This approach puts a greater burden on the user of the benchmarks but is more appropriate for comparing widely disparate machines. Providing the codes themselves, on the other hand, makes the user's task easier and may be a better approach for exploring architectural trade-offs among a well-defined class of similar architectures. While this philosophy remains, the NAS group provides message-passing implementations of the programs that can be used as starting points.

Because the NAS benchmarks, particularly the kernels, are relatively easy to implement and represent an interesting range of computations for scientific computing, they have been widely embraced by multiprocessor vendors. With the portability provided by the Message Passing Interface (MPI) standard, a recent follow-on effort called NPB2 utilizes fixed applications written in MPI rather than pencil-and-paper benchmarks, just like traditional benchmark suites. The benchmarks and their documentation can be obtained from NAS, as described in Table A.1.

A.5 PARKBENCH

The PARKBENCH (PARallel Kernels and BENCHmarks) effort (PARKBENCH Committee 1994) is a large-scale effort to develop a suite of microbenchmarks, kernels, and applications for benchmarking parallel machines, with at least an initial focus on explicit message-passing programs written in Fortran77. Versions of the programs in the High Performance Fortran (HPF) language (High Performance Fortran Forum 1993) are also furnished in order to provide portability across message-passing and shared address space platforms. The programs are taken from scientific computing.

PARKBENCH provides the different types of benchmarks we have discussed: low-level benchmarks or microbenchmarks, kernels, compact applications, and compiler benchmarks (the last two categories yet to be fully provided). The lowest-level microbenchmarks measure per-node performance. The purpose of these uniprocessor microbenchmarks is to characterize the performance of various aspects of the architecture and compiler system and to obtain some parameters that can be used to understand the performance of the kernels and compact applications. The uniprocessor microbenchmarks include timer calls, arithmetic operations, and memory bandwidth and latency stressing routines. There are also multiprocessor microbenchmarks that test communication latency and bandwidth—both point-to-point and all-to-all—as well as global barrier synchronization. Several of these multiprocessor microbenchmarks are taken from the earlier Genesis benchmark suite (Hey 1991).

The kernel benchmarks are divided into matrix kernels (multiplication, factorization, transposition, and tridiagonalization), Fourier transform kernels (a large 1D FFT and a large 3D FFT), partial differential equation kernels (a 3D successive-over-relaxation iterative solver and the multigrid kernel from the NAS suite), and others including the conjugate gradient, integer sort, and embarrassingly parallel kernels from the NAS suite and a paper-and-pencil I/O benchmark.

Compact applications (full but perhaps simplified applications) are intended in the areas of climate and meteorological modeling, computational fluid dynamics, financial modeling and portfolio optimization, molecular dynamics, plasma physics, quantum chemistry, quantum chromodynamics, and reservoir modeling, among others. Finally, the compiler benchmarks are intended for people developing High Performance Fortran compilers to test their compiler optimizations, not so much to evaluate architectures. The available PARKBENCH benchmarks can be obtained

from the Netlib repository at Oak Ridge National Laboratories, as described in Table A.1.

A.6 OTHER ONGOING EFFORTS

SPEC/HPCG: The developers of the widely used SPEC (Standard Performance Evaluation Corporation) suite for uniprocessor benchmarking (SPEC 1995) have teamed up with the developers of the Perfect (PERformance Evaluation for Cost-effective Transformations) Club benchmarks for traditional vector supercomputers (Berry et al. 1989) to form the SPEC/HPG (High Performance Group) to develop a suite of benchmarks (Eigenmann and Hassanzadeh 1996) measuring the performance of systems that “push the limits of computational technology,” notably multiprocessor systems. These, too, are focused on scientific computing.

Many other benchmarking efforts exist. As we can see, most of the suites so far are for message-passing computing, though some of these may be beginning to provide versions in High Performance Fortran that can be run on any of the major communication abstractions with the appropriate compiler support. We can expect the development of more shared address space benchmarks, such as in the SPLASH and SPLASH-2 suites, in the future. Many of the existing suites are also targeted toward scientific computing (the PARKBENCH and NAS efforts being the most large scale among these), though there is increasing interest in producing benchmarks for other classes of workloads (including commercial and general-purpose server workloads) for parallel machines. Almost all the benchmarks are designed to be a single application running on the machine at a time. Good benchmarks for multiprogrammed and other workloads that exercise the operating system do not yet exist (except for TPC); nor do we have well-established I/O-intensive benchmarks for parallel machines. In general, developing benchmarks and workloads that are representative of real-world parallel computing and are also effectively portable to real machines is a very difficult but very important problem (since the conclusions we draw about architectural trade-offs depend on the benchmarks used), and the preceding are all steps in the right direction.

References

- Abali, B., and C. Aykanat. 1994. Routing Algorithms for IBM SP1. *Lecture Notes in Computer Science*, Vol. 853. New York: Springer-Verlag, 161–175.
- Abdel-Shafii, H. A., J. Hall, S. V. Adve, and V. S. Adve. 1997. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. *Proc. Third Symposium on High Performance Computer Architecture* (February).
- Adve, S. V. 1993. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. Ph.D. diss., University of Wisconsin-Madison. Available as Tech. Report #1198, University of Wisconsin-Madison, Computer Science (December).
- Adve, S. V., and K. Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29(12):66–76.
- Adve, S. V., K. Gharachorloo, A. Gupta, J. L. Hennessy, and M. Hill. 1993. Sufficient Systems Requirements for Supporting the PLpc Memory Model. Tech. Report #1200, University of Wisconsin-Madison, Computer Science (December). Also available as Tech. Report #CSL-TR-93-595, Stanford University.
- Adve, S. V., and M. Hill. 1990a. Weak Ordering: A New Definition. 1990. *Proc. 17th Int'l Symposium on Computer Architecture* (May):2–14.
- . 1990b. Implementing Sequential Consistency in Cache-Based Systems. *Proc. 1990 Int'l Conference on Parallel Processing* (August):47–50.
- . 1993. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel and Distributed Systems* 4(6):613–624.
- Agarwal, A. 1991. Limit on Interconnection Performance. *IEEE Transactions on Parallel and Distributed Systems* 2(4):398–412.
- Agarwal, A., R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. 1995. The MIT Alewife Machine: Architecture and Performance. *Proc. 22nd Int'l Symposium on Computer Architecture* (May/June):2–13.
- Agarwal, A., and A. Gupta. 1988. Memory-Reference Characteristics of Multiprocessor Applications Under MACH. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May):215–225.
- Agarwal, A., B.-H. Lim, D. Kranz, and J. Kubiatowicz. 1990. (April): A Processor Architecture for Multiprocessing. *Proc. 17th Annual Int'l Symposium on Computer Architecture* (June):104–114.
- . 1991. LimitLESS Directories: A Scalable Cache Coherence Scheme. *Proc. Fourth Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (April):224–234.

- Agarwal, A., R. Simoni, J. Hennessy, and M. Horowitz. 1988. An Evaluation of Directory Schemes for Cache Coherence. *Proc. 15th Int'l Symposium on Computer Architecture* (June):280–289.
- Aiken, A., and A. Nicolau. 1988. Optimal Loop Parallelization. *Proc. SIGPLAN Conference on Programming Language Design and Implementation* (June):308–317. Also published in SIGPLAN Notices 23(7).
- Aimoto, Y., T. Kimura, Y. Yabe, H. Heiuchi, et al. 1996. A 7.68GIPS 3.84GB/s 1W Parallel Image-Processing RAM Integrating a 16Mb DRAM and 128 Processors. *International Solid-State Circuits Conference*, San Francisco (February):372–373.
- Alexander, T. B., K. G. Robertson, D. T. Lindsay, D. L. Rogers, J. R. Obermeyer, J. R. Keller, K. Y. Oka, and M. M. Jones II. 1994. Corporate Business Servers: An Alternative to Mainframes for Business Computing (HP K-Class). *Hewlett-Packard Journal* (June):8–33.
- Almasi, G. S., and A. Gottlieb. 1989. *Highly Parallel Computing*. Redwood City, CA: Benjamin/Cummings.
- Alverson, R., D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. 1990. The Tera Computer System. *Proc. 1990 Int'l Conference on Supercomputing* (June): 1–6.
- Amdahl, G. M. 1967. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS 1967 Spring Joint Computer Conference* 40:483–485.
- Anderson, J. P., S. A. Hoffman, J. Shifman, and R. Williams. 1962. D825-A Multiple-Computer System for Command and Control. *AFIP Proc. FJCC* 22:86–96.
- Anderson, J., and M. Lam. 1993. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. *Proc. SIGPLAN'93 Conference on Programming Language Design and Implementation* (June).
- Anderson, T. E., D. E. Culler, D. Patterson. 1995. A Case for NOW (Networks of Workstations). *IEEE Micro* 15(1):54–6.
- Anderson, T. E., S. S. Owicki, J. P. Saxe, and C. P. Thacker. 1992. High Speed Switch Scheduling for Local Area Networks. *Proc. ASPLOS V* (October):98–110.
- Archibald, J., and J.-L. Baer. 1986. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems* 4(4):273–298.
- Arnould, E. A., F. J. Bitz, E. C. Cooper, H. T. Kung, R. D. Sansom, and P. A. Steenkiste. 1989. The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers. *Proc. ASPLOS III* (April):205–216.
- Arpacı, R. H., D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yellick. 1995. Empirical Evaluation of the Cray-T3D: A Compiler Perspective. *Proc. 22nd Int'l Symposium on Computer Architecture* (June):320–331.
- Arvind, and D. E. Culler. 1986. Dataflow Architectures. *Annual Reviews in Computer Science* 1:225–253. Palo Alto, CA: Annual Reviews. Reprinted in *Dataflow and Reduction Architectures*. Edited by S. S. Thakkar. Los Alamitos, CA: IEEE Computer Society Press, 1987.
- Athas, W. C., and C. L. Seitz. 1988. Multicomputers: Message-Passing Concurrent Computers. *IEEE Computer* 21(8):9–24.
- August, M. C., G. M. Brost, C. C. Hsiung, and A. J. Schiffleger. 1989. Cray X-MP: The Birth of a Supercomputer. *Computer* 22(1):45–52.
- Baer, J.-L., and T.-F. Chen. 1991. An Efficient On-Chip Preloading Scheme to Reduce Data Access Penalty. *Proc. Supercomputing '91* (November):176–186.
- Baer, J.-L., and W.-H. Wang. 1988. On the Inclusion Properties for Multi-Level Cache Hierarchies. *Proc. 15th Annual Int'l Symposium on Computer Architecture* (May):73–80.
- Bailey, D. H. 1990. FFTs in External or Hierarchical Memory. *Journal of Supercomputing* 4(1):23–35. Also published in *Proc. Supercomputing '89* (November):234–242.

- . 1991. Twelve Ways to Fool the Masses When Giving Performance Results on Parallel Computers. *Supercomputing Review* (August):54–55.
- . 1993. Misleading Performance Reporting in the Supercomputing Field. *Scientific Programming* 1(2):141–151. Also published in *Proc. Supercomputing '93*.
- Bailey, D. H., E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. 1991. The NAS Parallel Benchmarks. *Intl. Journal of Supercomputer Applications* 5(3):66–73. Also published as Tech. Report RNR-94-007, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center (March 1994).
- Bailey, D. H., E. Barszcz, L. Dagum, and H. D. Simon. 1994. NAS Parallel Benchmark Results 3–94. *Proc. Scalable High-Performance Computing Conference*, Knoxville, TN (May):111–120.
- Bailey, D., T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. 1995. *The NAS Parallel Benchmarks 2.0*. Report NAS-95-020, Numerical Aerodynamic Simulation Facility, NASA Ames Research Center (December).
- Baker, W. E., R. W. Horst, D. P. Sonnier, and W. J. Watson. 1995. A Flexible ServerNet-Based Fault-Tolerant Architecture. *Proc. 25th Int'l Symposium on Fault-Tolerant Computing* (June). Los Alamitos, CA: IEEE Computer Society Press, 2–11.
- Bakoglu, H. B. 1990. *Circuits, Interconnection, and Packaging for VLSI*. Reading, MA: Addison-Wesley.
- Ball, J. R., R. C. Bollinger, T. A. Jeeves, R. C. McReynolds, D. H. Shaffer. 1962. On the Use of the Solomon Parallel-Processing Computer. *Proc. AFIPS Fall Joint Computer Conference* 22:137–146.
- Banks, D., and M. Prudence. 1993. A High Performance Network Architecture for a PA-RISC Workstation. *IEEE Journal on Selected Areas in Communication* 11(2):191–202.
- Barnes, J. E., and P. Hut. 1989. Error Analysis of a Tree Code. *Astrophysics Journal Supplement* 70(June):389–417.
- Barroso, L., and M. Dubois. 1993. The Performance of Cache-Coherent Ring-Based Multiprocessors. *Proc. 20th Annual Int'l Symposium on Computer Architectures (ISCA)* (May):268–277.
- . 1995. Performance Evaluation of the Slotted Ring Multiprocessors. *IEEE Transactions on Computers* 44(7):878–890.
- Barroso, L. A., S. Iman, J. Jeong, K. Oner, K. Ramamurthy, and M. Dubois. 1995. RPM: A Rapid Prototyping Engine for Multiprocessor Systems. *IEEE Computer* 28(2):26–34.
- Barszcz, E., Fatoohi, R., Venkatakrishnan, V., and Weeratunga, S. 1993. *Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors*. Tech. Report NAS RNR-93-007, NASA Ames Research Center, Moffett Field, CA (April).
- Barton, E., J. Crownie, and M. McLaren. 1994. Message Passing on the Meiko CS-2. *Parallel Computing* 20(4):497–507.
- Baskett, F., T. Jermoluk, and D. Solomon. 1988. The 4D-MP Graphics Superworkstation: Computing + Graphics = 40 MIPS + 40 MFLOPS and 100,000 Lighted Polygons per Second. *Proc. 33rd IEEE Computer Society Int'l Conference—COMPCON '88* (February):468–471.
- Batcher, K. E. 1974. Staran Parallel Processor System Hardware. *Proc. AFIPS National Computer Conference*, 405–410.
- . 1980. Design of a Massively Parallel Processor. *IEEE Transactions on Computers* C-29(9):836–840.
- Bell, C. G. 1985. Multis: A New Class of Multiprocessor Computers. *Science* 228:462–467.
- Benes, V. 1965. *Mathematical Theory of Connecting Networks and Telephone Traffic*. San Diego, CA: Academic Press.

- Bennett, J. E., and M. J. Flynn. 1996a. *Latency Tolerance for Dynamic Processors*. Tech. Report #CSL-TR-96-687, Computer Systems Laboratory, Stanford University.
- . 1996b. *Reducing Cache Miss Rates Using Prediction Caches*. Tech. Report #CSL-TR-96-707, Computer Systems Laboratory, Stanford University.
- Berry, M., D. Chen, P. Koss, et al. 1989. The PERFECT Club Benchmarks: Effective Performance Evaluation of Computers. *Int'l Journal of Supercomputer Applications* 3(3):5–40.
- Bershad, B. N., M. J. Zekauskas, and W. A. Sawdon. 1993. The Midway Distributed Shared Memory System. *Proc. COMPCON '93* (February).
- Bhatt, S. M. and C. E. Leiserson. 1983. How to Assemble Tree Machines. *ACM Symposium on Theory of Computing (STOC '82)*. New York: ACM Press.
- Biagioli, E., E. Cooper, and R. Sansom. 1993. Designing a Practical ATM LAN. *IEEE Network* (March).
- Bilas, A., L. Iftode, and J. P. Singh. 1998. Evaluation of Hardware Support for Next-Generation Shared Virtual Memory Clusters. *Proc. Int'l Conference on Supercomputing* (July).
- Bisiani, R., and M. Ravishankar. 1990. PLUS: A Distributed Shared-Memory System. *Proc. 17th Int'l Symposium on Computer Architecture* (May):115–124.
- Black, D., R. Rashid, D. Golub, C. Hill, R. Baron. 1989. Translation Lookaside Buffer Consistency: A Software Approach. *Proc. Third Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Boston (April):113–122.
- Blackford, L. S., J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra; S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. 1997. *ScalAPACK Users' Guide*. Philadelphia, PA: Society for Industrial and Applied Mathematics (SIAM).
- Blelloch, G. 1993. Prefix Sums and Their Applications. In *Synthesis of Parallel Algorithms*. Edited by J. Reif. San Francisco: Morgan Kaufmann, 35–60.
- Blelloch, G. E., C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. A. Zagha. 1991. Comparison of Sorting Algorithms for the Connection Machine CM-2. *Proc. Symposium on Parallel Algorithms and Architectures* (July):3–16.
- Blumrich, M. A., C. Dubnicki, E. W. Felten, K. Li, M.R. Mesarina. 1994. Two Virtual Memory Mapped Network Interface Designs. *Proc. Hot Interconnects II Symposium* (August).
- Blumrich, M., K. Li, R. Alpert, C. Dubnicki, E. Felten, and J. Sandberg. 1994. A Virtual Memory Mapped Network Interface for the Shrimp Multicomputer. *Proc. 21st Int'l Symposium on Computer Architecture* (April):142–153.
- Boden, N., D. Cohen, R. Felderman, A. Kulawik, C. Seitz, J. Seizovic, and W. Su. 1995. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro* 15(1):29–38.
- Bodin, F., P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr. 1993. Implementing a Parallel C++ Runtime System for Scalable Parallel Systems. *Proc. Supercomputing '93* (November):588–597. Also in *Scientific Programming* 2(3).
- Bolt Beranek and Newman Advanced Computers. 1989. *TC2000 Technical Product Summary*. Cambridge, MA: Bolt Beranek and Newman.
- Bomans, L., and D. Roose. 1989. Benchmarking the iPSC/2 Hypercube Multiprocessor. *Concurrency: Practice and Experience*, 1(1):3–18.
- Borkar, S., R. Cohn, G. Cox, T. Gross, H. T. Kung, M. Lam, M. Levine, B. Moore, W. Moore, C. Peterson, J. Susman, J. Sutton, J. Urbanski, and J. Webb. 1990. Supporting Systolic and Memory Communication in iWarp. *Proc. 17th Annual Int'l Symposium on Computer Architecture*, Seattle, WA (May):70–81. Revised version appears as Tech. Report #CMU-CS-90-197, Carnegie Mellon University.
- Bouknight, W. J., S. A Denenberg, D. E. McIntyre, J. M. Randall, A. H. Sameh, and D. L. Slotnick. 1972. The Illiac IV System. *Proc. IEEE* 60(4):369–388.

- Boyle, J., R. Butler, T. Disz, B. Glickfield, E. Lusk, W. R. Overbeek, J. Patterson, and R. Stevens. 1987. *Portable Programs for Parallel Processors*. New York: Holt, Rinehart and Winston.
- Brewer, E. A., F. T. Chong, F. T. Leighton. 1994. Scalable Expanders: Exploiting Hierarchical Random Wiring. *Proc. 1994 Symposium on the Theory of Computing*, Montreal, Canada (May):144–152.
- Brewer, E. A., F. T. Chong, L. T. Liu, J. Kubiatowicz, S. D. Sharma. 1995. Remote Queues: Exposing Network Queues for Atomicity and Optimization. *Proc. Seventh Annual Symposium on Parallel Algorithms and Architectures* (July):42–53.
- Brewer, E. A., and B. C. Kuszmaul. 1994. How to Get Good Performance from the CM-5 Data Network. *Proc. 1994 Int'l Parallel Processing Symposium*, Cancun, Mexico (April):858–867.
- Bruno, J., P. R. Cappello. 1988. Implementing the Beam and Warming Method on the Hypercube. *Proc. Third Conference on Hypercube Concurrent Computers and Applications*, Pasadena, CA, Jan 19–20.
- Burger, D. 1997. *System-Level Implications of Processor-Memory Integration*. Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. Presented at the Int'l Symposium on Computer Architecture (ISCA) '97 (June).
- Burger, D., J. Goodman, and A. Kagi. 1996. Memory Bandwidth Limitations in Future Microprocessors. *Proc. 23rd Annual Symposium on Computer Architecture* (May):78–89.
- Burkhardt, H., et al. 1992. *Overview of the KSR-1 Computer System*. Tech. Report KSR-TR-9202001, Kendall Square Research, Boston (February).
- Butler, M., T-Y. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. 1991. Single Instruction Stream Parallelism Is Greater Than Two. *Proc. Annual Int'l Symposium on Computer Architecture* (ISCA), 276–86.
- Callahan, T., and S. C. Goldstein. 1995. NIFDY: A Low Overhead, High Throughput Network Interface. *Proc. 22nd Annual Symposium on Computer Architecture* (June):230–241.
- Cardoza, W., F. Glover, and W. Snaman Jr. 1996. Design of a TruCluster Multicomputer System for the Digital UNIX Environment. *Digital Technical Journal* 8(1):5–17.
- Carter, J. B., J. K. Bennett, and W. Zwaenepoel. 1991. Implementation and Performance of Munin. *Proc. 13th Symposium on Operating Systems Principles* (October):152–164.
- . 1995. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions of Computer Systems* 13(3):205–244.
- Catanzaro, B. 1997. *Multiprocessor System Architectures: A Technical Survey of Multiprocessor/Multithreaded Systems Using SPARC, Multi-level Bus Architectures and Solaris (SunOS)*. Mountain View, CA: Sun Microsystems.
- Cekleov, M., D. Yen, P. Sindhu, J.-M. Frailong, et al. 1993. SPARCcenter 2000: Multiprocessing for the 90s, Digest of Papers. *Proc. COMPCON Spring '93*. Los Alamitos, CA: IEEE Computer Society Press, 345–353.
- Censier, L., and P. Feautrier. 1978. A New Solution to Cache Coherence Problems in Multiprocessor Systems. *IEEE Transaction on Computer Systems* C-27(12):1112–1118.
- Chan, K., et al. 1993. Multiprocessor Features of the HP Corporate Business Servers. *Proc. COMPCON* (Spring):330–337.
- Chandy, K. M., and J. Misra. 1988. *Parallel Program Design: A Foundation*. Reading, MA: Addison Wesley.
- Chang, P. P., S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu. 1991. IMPACT: An Architectural Framework for Multiple-Instruction Issue Processors. *Proc. 18th Int'l Symposium on Computer Architecture* (ISCA) 19(3):266–275.

- Chen, T.-F., and J.-L. Baer. 1992. Reducing Memory Latency via Non-Blocking and Prefetching Caches. *Proc. Fifth Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):51–61.
- . 1994. A Performance Study of Software and Hardware Data Prefetching Schemes. *Proc. 21st Annual Symposium on Computer Architecture* (April):223–232.
- Cheong, H., and A. Videnbaum. 1990. Compiler-directed Cache Management in Multiprocessors. *IEEE Computer* 23(6):39–47.
- Chien, A. A., and J. H. Kim. 1992. Planar-Adaptive Routing: Low-Cost Adaptive Networks for Multiprocessors. *Proc. 19th Annual International Symposium on Computer Architecture (ISCA)*, Gold Coast, Australia (May):268–277.
- Choi, J., J. J. Dongarra, R. Pozo, and D. W. Walker. 1992. ScaLAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers. *Proc. Fourth Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA. Los Alamitos, CA: IEEE Computer Society Press, 120–127.
- Chun, B. N., A. M. Mainwaring, and D. E. Culler. 1998. Virtual Network Transport Protocols for Myrinet. *IEEE Micro* (January):53–63.
- Clark, R., and K. Alnes. 1996. An SCI Chipset and Adapter. *Symposium Record, Hot Interconnects IV* (August):221–235.
- Cohen, D., G. Finn, R. Felderman, and A. DeSchon. 1993. ATOMIC: A Low-Cost, Very High-Speed, Local Communication Architecture. *Proc. 1993 Int. Conference on Parallel Processing*.
- Convex Computer Corporation. 1993. *Exemplar Architecture*. Richardson, TX: Convex Computer Corp.
- Corella, F., J. Stone, C. Barton. 1993. *A Formal Specification of the PowerPC Shared Memory Architecture*. Tech. Report Computer Science RC 18638 (81566), IBM Research Division, T.J. Watson Research Center (January).
- Cornell, J. A. 1972. Parallel Processing of Ballistic Missile Defense Radar Data with PEPE. *COMPCON 72*, 69–72.
- Cox, A., and R. Fowler. 1993. Adaptive Cache Coherency for Detecting Migratory Shared Data. *Proc. 20th Int'l Symposium on Computer Architecture* (May):98–108.
- Crowther, W., J. Goodhue, R. Gurwitz, R. Rettberg, and R. Thomas. 1985. The Butterfly Parallel Processor. *IEEE Computer Architecture Technical Newsletter*, 18–46.
- Culler, D. E. 1994. Multithreading: Fundamental Limits, Potential Gains, and Alternatives. In *Multithreaded Computer Architecture: A Summary of the State of the Art*. Edited by R. Ianuccci. Dordrecht, Germany; Norwell, MA: Kluwer Academic Publishers, 97–138.
- Culler, D. E., A. C. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yellick. 1993. Parallel Programming in Split-C. *Proc. Supercomputing '93* (November):262–273.
- Culler, D. E., A. C. Dusseau, R. P. Martin, and K. E. Schauser. 1993. Fast Parallel Sorting under LogP: From Theory to Practice. In *Portability and Performance for Parallel Processing*, Chapter 4. New York: John Wiley & Sons, 71–98.
- Culler, D. E., R. M. Karp, D. A., Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. 1993. LogP: Toward A Realistic Model of Parallel Computation. *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (May):1–12.
- . 1996. LogP: A Practical Model of Parallel Computation. *CACM* 39(11):78–85.
- Culler, D. E., A. Sah, K. E. Schauser, T. von Eicken, and J. Wawrzynek. 1991. Fine-Grain Parallelism with Minimal Hardware Support. *Proc. Fourth Int'l Symposium on Arch. Support for Programming Languages and Systems (ASPLOS)* (April):164–175.

- Culler, D. E., K. E. Schauser, and T. von Eicken. 1993. Two Fundamental Limits on Dataflow Multithreading. *Proc. IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, FL.
- Dahlgren, F. 1995. Boosting the Performance of Hybrid Snooping Cache Protocols. *Proc. 22nd Int'l Symposium on Computer Architecture* (June):60–69.
- Dahlgren, F., M. Dubois, and P. Stenstrom. 1994. Combined Performance Gains of Simple Cache Protocol Extensions. *Proc. 21st Int'l Symposium on Computer Architecture* (April):187–197.
- . 1995. Sequential Hardware Prefetching in Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 6(7).
- Dally, W. J. 1990a. Virtual-Channel Flow Control. *Proc. 17th Annual Int'l Symposium on Computer Architecture* (ISCA), Seattle, WA, (May):60–68.
- Dally, W. J. 1990b. Performance Analysis of k -ary n -cube Interconnection Networks. *IEEE-TOC* 39(6):775–85.
- Dally, W. J., A. Chien, S. Fiske, W. Horwat, J. Keen, J. Larivee, R. Lethin, P. Nuth, S. Willis. 1989. The J-Machine: A Fine-Grained Concurrent Computer. *Proc. IFIP 11th World Computer Congress, Information Processing '89*, 1147–1153.
- Dally, W. J., J. A. S. Fiske, J. S. Keen, R. A. Lethin, M. D. Noakes, and P. R. Nuth. 1992. The Message Driven Processor: A Multicomputer Processing Node with Efficient Mechanisms. *IEEE Micro* (April):23–39.
- Dally, W. J., J. S. Keen, M. D. Noakes. 1993. The J-Machine Architecture and Evaluation. *Digest of Papers. COMPCON Spring '93*, San Francisco, CA (February):183–188.
- Dally, W. J., and C. Seitz. 1987. Deadlock-Free Message Routing in Multiprocessor Interconnections Networks. *IEEE-TOC* C-36(5):547–553.
- Denning, P. J. 1968. The Working Set Model for Program Behavior. *Communications of the ACM* 11(5):323–333.
- Dennis, J. B. 1980. Dataflow Supercomputers. *IEEE Computer* 13(11):93–100.
- Digital Equipment Corporation. 1992. *Alpha Architecture Handbook*. Maynard, MA: Digital Equipment Corp.
- Dijkstra, E. W. 1965. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM* 8(9):569.
- Dijkstra, E. W., and C. S. Sholten. 1968. Termination Detection for Diffusing Computations. *Information Processing Letters* 1:1–4.
- Dongarra, J. J. 1990. *Performance of Various Computers Using Standard Linear Equations Software in a Fortran Environment*. Tech. Report CS-89-85, University of Tennessee, Computer Science Dept. (March).
- . 1994. *Performance of Various Computers Using Standard Linear Equation Software*. Tech. Report CS-89-85, University of Tennessee, Computer Science Dept. (November); current report available from netlib@ornl.gov
- Dongarra, J. J., J. Martin, and J. Wrolton. 1987. Computer Benchmarking: Paths and Pitfalls. *IEEE Spectrum* (July):38.
- Dongarra, J. J., and D. W. Walker. 1995. Software Libraries for Linear Algebra Computations on High performance Computers. *SIAM Review* 37:151–180.
- Dongarra, J. J., and W. Gentzsch, eds. 1993. *Computer Benchmarks*. Amsterdam: Elsevier Science B. V., North-Holland.
- Dubnicki, C., L. Iftode, E. W. Felten, K. Li. 1996. Software Support for Virtual Memory-Mapped Communication. *Tenth Int'l Parallel Processing Symposium* (April).

- Dubnicki, C., and T. LeBlanc. 1992. Adjustable Block Size Coherent Caches. *Proc. 19th Annual Int'l Symposium on Computer Architecture* (May):170–180.
- Dubois, M., and C. Scheurich. 1990. Memory Access Dependencies in Shared-Memory Multiprocessors. *IEEE Transactions on Software Engineering* 16(6):660–673.
- Dubois, M., C. Scheurich, and F. Briggs. 1986. Memory Access Buffering in Multiprocessors. *Proc. 13th Int'l Symposium on Computer Architecture* (June):434–442.
- Dubois, M., J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenstrom. 1993. The Detection and Elimination of Useless Misses in Multiprocessors. *Proc. 20th Int'l Symposium on Computer Architecture* (May):88–97.
- Dubois, M., J.-C. Wang, L. A. Barroso, K. Chen and Y.-S. Chen. 1991. Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs. *Proc. Supercomputing '91* (November):197–206.
- Dunigan, T. H. 1988. *Performance of a Second Generation Hypercube* Tech. Report ORNL/TM-10881, Oak Ridge National Lab. (November).
- Dunning, D., G. Regnier, G. McAlpine, D. Camaron, B. Shubert, F. Berry, A. M. Merritt, E. Gronke, and C. Dodd. 1998. The Virtual Interface Architecture. *IEEE Micro* 18(2).
- Dusseau, A. C., D. E. Culler, K. E. Schauser, and R. P. Martin. 1996. Fast Parallel Sorting under LogP: Experience with the CM-5. *IEEE Transactions on Parallel and Distributed Systems* 7(8):791–805.
- Dwarkadas, S., P. Keleher, A. L. Cox, and W. Zwaenepoel. 1993. Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology. *Proc. 20th Int'l Symposium on Computer Architecture* (May):144–155.
- Eggers, S., and R. Katz. 1988. A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation. *Proc. 15th Annual Int'l Symposium on Computer Architecture* (May):373–382.
- . 1989a. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. *Proc. Third Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (May):257–270.
- . 1989b. Evaluating the Performance of Four Snooping Cache Coherency Protocols. *Proc. 16th Annual Int'l Symposium on Computer Architecture* (May):2–15.
- Eigenmann, R., and S. Hassanzadeh. 1996. Benchmarking with Real Industrial Applications: The SPEC High Performance Group. *IEEE Computational Science and Engineering* (spring).
- Elliott, D. G., W. M. Snelgrove, and M. Stumm. 1992. Computational RAM: A Memory-SIMD Hybrid and Its Application to DSP. *Custom Integrated Circuits Conference*, Boston, MA (May):30.6.1–30.6.4.
- Elliott, D. G., M. Stumm, and W. M. Snelgrove. 1997. *Computational RAM: The Case for SIMD Computing in Memory*. Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. Presented at Annual International Symposium on Computer Architecture (ISCA) '97 (June).
- Erlichson, A., B. Nayfeh, J. P. Singh and Oyekunle Olukotun. 1995. The Benefits of Clustering in Cache-Coherent Multiprocessors: An Application-Driven Investigation. *Proc. Supercomputing '95* (November).
- Erlichson, A., N. Nuckolls, G. Chesson, and J. L. Hennessy. 1996. SoftFLASH: Analyzing the Performance of Clustered Distributed Virtual Shared Memory. *Proc. Seventh Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):210–220.
- Falsafi, B., A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus, A. Rogers, and D. A. Wood. 1994. Application-Specific Protocols for User-Level Shared Memory. *Proc. Supercomputing '94* (November):380–389.

- Falsafi, B. and D. A. Wood. 1997. Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA. *Proc. 24th Int'l Symposium on Computer Architecture* (June):229–240.
- Farkas, K., Z. Vranesic, and M. Stumm. 1992. Cache Consistency in Hierarchical Ring-Based Multiprocessors. *Proc. Supercomputing '92* (November).
- Feigel, C. P. 1994. TI Introduces Four-Processor DSP Chip. *Microprocessor Report* (March):28.
- Felderman, R., et al. 1994. Atomic: A High Speed Local Communication Architecture. *Journal of High Speed Networks* 3(1):1–29.
- Fenwick, D. M., D. J. Foley, W. B. Gist, S. R. VanDoren, and D. Wissell. 1995. The AlphaServer 8000 Series: High-End Server Platform Development. *Digital Technical Journal* 7(1):43–65.
- Flanagan, J. L. 1994. Technologies for Multimedia Communications. *IEEE Proceedings* 82(4):590–603.
- Flynn, M. J. 1972. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computing* C-21(September):948–960.
- Fortune, S., and J. Wyllie. 1978. Parallelism in Random Access Machines. *Proc. 10th ACM Symposium on Theory of Computing* (May).
- Fox, G., M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. 1988. *Solving Problems on Concurrent Processors*, vol. 1. Englewood Cliffs, NJ: Prentice Hall.
- Frailong, J.-L., et al. 1993. The Next Generation SPARC Multiprocessing System Architecture. *Proc. COMPCON* (spring):475–480.
- Frank, S., H. Burkhardt III, and J. Rothnie. 1993. The KSRI: Bridging the Gap between Shared Memory and MPPs. *Proc. COMPCON, Digest of Papers* (spring):285–294.
- Fu, J. W. C., and J. H. Patel. 1991. Data Prefetching in Multiprocessor Vector Cache Memories. *Proc. 18th Annual Symposium on Computer Architecture* (May):54–63.
- Fu, J. W. C., J. H. Patel., and B. L. Janssens. 1992. Stride Directed Prefetching in Scalar Processors. *Proc. 25th Annual Int'l Symposium on Microarchitecture* (December):102–110.
- Fuchs, H., G. Abram, and E. Grant. 1983. Near Real-Time Shaded Display of Rigid Objects. *Proc. SIGGRAPH*.
- Galles, M., and E. Williams. 1993. Performance Optimizations, Implementation, and Verification of the SGI Challenge Multiprocessor. *Proc. 27th Hawaii Int'l Conference on System Sciences Vol. I: Architecture* (January). Also in *SGI Challenge*. Edited by T. N. Mudge and B. D. Shriner. Los Alamitos, CA: IEEE Computer Society Press, 1994, 134–143.
- Geist, A., A. Beguelin, and J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. 1994. *PVM 3.0 Users' Guide and Reference Manual*. Tech. Report ORNL/TM-12187, Oak Ridge, TN: Oak Ridge National Laboratory (February). <http://www.eece.ksu.edu/pvm3/ug.ps>.
- Geist, A., A. Beguelin, J. Dongarra, R. Manchek, W. Jiang, and V. Sunderam. 1994. *PVM: A Users' Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: MIT Press.
- Geist, G. A., and V. S. Sunderam. 1992. Network Based Concurrent Computing on the PVM System. *Journal of Concurrency: Practice and Experience* 4(4):293–311.
- Gharachorloo, K. 1995. *Memory Consistency Models for Shared-Memory Multiprocessors*. Ph.D. diss., Computer Systems Laboratory, Stanford University (December). Also published as Tech. Report #CSL-TR-95-685.
- Gharachorloo, K., S. Adve, A. Gupta, M. Hill, and J. L. Hennessy. 1992. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing* 15(4):399–407.

- Gharachorloo, K., A. Gupta, and J. L. Hennessy. 1991a. Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors. *Proc. 4th Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (April):245-257.
- . 1991b. Two Techniques to Enhance the Performance of Memory Consistency Models. *Proc. Int'l Conference on Parallel Processing* (August): 1355-1364.
- . 1992. Hiding Memory Latency Using Dynamic Scheduling in Shared Memory Multiprocessors. *Proc. 19th Int'l Symposium on Computer Architecture* (May):22-33.
- Gharachorloo, K., D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. L. Hennessy. 1990. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. *Proc. 17th Int'l Symposium on Computer Architecture* (May):15-26.
- Gillett, R. 1996. Memory Channel Network for PCI. *IEEE Micro* 16(1):12-18.
- Gillett, R., M. Collins, and D. Pimm. 1996. Overview of Network Memory Channel for PCI. *Proc. IEEE Spring COMPCON '96* (February).
- Gillett, R., and R. Kaufmann. 1997. Using Memory Channel Network. *IEEE Micro* 17(1):19-25.
- Glass, C. J., and L. M. Ni. 1992. The Turn Model for Adaptive Routing. *Proc. Annual International Symposium on Computer Architecture (ISCA)* (May): 278-287.
- Godiwala, N. D., and B. A. Maskas. 1995. The Second-Generation Processor Module for AlphaServer 2100 Systems. *Digital Technical Journal* 7(1).
- Gokhale, M., B. Holmes, and K. Iobst. 1995. Processing in Memory: The Terasys Massively Parallel PIM Array. *IEEE Computer* 28(3):23-31.
- Goldschmidt, S. R. 1993. *Simulation of Multiprocessors: Speed and Accuracy*. Ph.D. diss., Stanford University (June).
- Golub, G., and C. Van Loan. 1997. *Matrix Computations 3e*. Baltimore, MD: Johns Hopkins University Press.
- Goodman, J. R. 1983. Using Cache Memory to Reduce Processor-Memory Traffic. *Proc. 10th Annual Int'l Symposium on Computer Architecture* (June):124-131.
- . 1987. Coherency for Multiprocessor Virtual Address Caches. *Proc. Second Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA (October):72-81.
- . 1989. *Cache Consistency and Sequential Consistency*. Tech. Report #1006, University of Wisconsin-Madison, Computer Science Dept. (February).
- Goodman, J. R., M. K. Vernon, P.J. Woest. 1989. Set of Efficient Synchronization Primitives for a Large-Scale Shared-Memory Multiprocessor. *Proc. Third Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (April):64-75.
- Gottlieb, A., R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. 1983. The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer. *IEEE Transactions on Computers* C-32(2):175-189.
- Gottlieb, A., and C. P. Kruskal. 1984. Complexity Results for Permuting Data and Other Computations on Parallel Processors. *Journal of the ACM* 31(April):193-209.
- Gottlieb, A., B. Lubachevsky, and L. Rudolph. 1983. Basic Techniques for the Efficient Coordination of Large Numbers of Cooperating Sequential Processes. *ACM Transactions on Programming Languages and Systems* 5(2).
- Grafe, V. G., and J. E. Hoch. 1990. The Epsilon-2 Hybrid Dataflow Architecture. *Proc. COMP-CON Spring '90*, San Francisco, CA (March):88-93.

- Grahn, H., and P. Stenstrom. 1996. Evaluation of a Competitive-Update Cache Coherence Protocol with Migratory Data Detection. *Journal of Parallel and Distributed Computing* 39(2):168–180.
- Grahn, H., P. Stenstrom, and M. Dubois. 1995. Implementation and Evaluation of Update-Based Protocols under Relaxed Memory Consistency Models. *Future Generation Computer Systems* 11(3):247–271.
- Granuke, G., and S. Thakkar. 1990. Synchronization Algorithms for Shared Memory Multiprocessors. *IEEE Computer* 23(6):60–69.
- Gray, J. 1991. *The Benchmark Handbook for Database and Transaction Processing Systems*. San Francisco: Morgan Kaufmann.
- Green, S. A., and D. J. Paddon. 1990. A Highly Flexible Multiprocessor Solution for Ray Tracing. *The Visual Computer* 6:62–73.
- Greenberg, R. I., and C. E. Leiserson. 1989. Randomized Routing on Fat-Trees. *Advances in Computing Research* 5:345–374.
- Greenwald, M., and D. R. Cheriton. 1996. The Synergy between Non-Blocking Synchronization and Operating System Structure. *Proc. Second Symposium on Operating System Design and Implementation*, USENIX, Seattle (October):123–136.
- Gropp, W., E. Lusk, and A. Skjellum. 1994. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. Cambridge, MA: MIT Press.
- Groscup, W. 1992. The Intel Paragon XP/S Supercomputer. *Proc. Fifth ECMWF Workshop on the Use of Parallel Processors in Meteorology* (November):262–273.
- Gunther, K. D. 1981. Prevention of Deadlocks in Packet-Switched Data Transport Systems. *IEEE Transactions on Communication* C-29(4):512–24.
- Gupta, A., J. L. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. 1991. Comparative Evaluation of Latency Reducing and Tolerating Techniques. *Proc. 18th Int'l Symposium on Computer Architecture* (May):254–263.
- Gupta, A., and W.-D. Weber. 1992. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers* 41(7):794–810.
- Gupta, A., W.-D. Weber, and T. Mowry. 1990. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache-Coherence Schemes. *Proc. Int'l Conference on Parallel Processing* I(August):312–321.
- Gurd, J. R., C. C. Kerkham, and I. Watson. 1985. The Manchester Prototype Dataflow Computer. *Communications of the ACM* 28(1):34–52.
- Gustafson, J. L. 1988. Reevaluating Amdahl's Law. *Communications of the ACM* 31(5):532–533.
- Gustafson, J. L., and Q. O. Snell. 1994. *HINT: A New Way to Measure Computer Performance*. Tech. Report, Ames Laboratory, U.S. Dept. of Energy, Ames, IA.
- Gustavson, D. 1992. The Scalable Coherence Interface and Related Standards Projects. *IEEE Micro* 12(1):10–22.
- Gwynnep, L. 1994a. Microprocessors Head Toward MP on a Chip. *Microprocessor Report* (May).
- . 1994b. PA-7200 Enables Inexpensive MP Systems. *Microprocessor Report* (March).
- Hagersten, E. 1992. *Toward Scalable Cache Only Memory Architectures*. Ph.D. diss., Swedish Institute of Computer Science (October).
- Hagersten, E., A. Landin, and S. Haridi. 1992. DDM—A Cache Only Memory Architecture. *IEEE Computer* 25(9):44–54.
- Hanrahan, P., D. Salzman, and L. A. Uppercle. 1991. A Rapid Hierarchical Radiosity Algorithm. *Proc. SIGGRAPH* (July).

- Hayashi, K., T. Doi, T. Horie, Y. Koyanagi, O. Shiraki; N. Immura, T. Shimizu, H. Ishihata, and T. Shindo. 1994. AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler. *ACM SIGPLAN Notices* 29(11):196.
- Heinlein, J., R. P. Boesch, Jr., K. Gharachorloo, M. Rosenblum, and A. Gupta. 1997. Coherent Block Data Transfer in the FLASH Multiprocessor. *Proc. 11th Int'l Parallel Processing Symposium* (April).
- Heinlein, J., K. Gharachorloo, S. Dresser, and A. Gupta. 1994. Integration of Message Passing and Shared Memory in the Stanford FLASH Multiprocessor. *Proc. 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):38–50.
- Heinrich, M., J. Kuskin, D. Ofelt, J. Heinlein, J. Baxter, J. P. Singh, R. Simoni, K. Gharachorloo, D. Nakahira, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. 1994. The Performance Impact of Flexibility on the Stanford FLASH Multiprocessor. *Proc. 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):274–285.
- Hennessy, J. L., and N. Jouppi. 1991. Computer Technology and Architecture: An Evolving Interaction. *IEEE Computer* 24(9):18–29.
- Hennessy, J. L., and D. A. Patterson. 1996. *Computer Architecture: A Quantitative Approach*. 2nd ed. San Francisco: Morgan Kaufmann.
- Herlihy, M. P. 1988. Impossibility and Universality Results for Wait-Free Synchronization. *Seventh ACM SIGACTS-SIGOPS Symposium on Principles of Distributed Computing* (August):276–290.
- . 1991. Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems* 13(1):124–149.
- . 1993. A Methodology for Implementing Highly Concurrent Data Objects. *ACM Transactions on Programming Languages and Systems* 15(5):745–770.
- Herlihy, M. P., and J. E. B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. *Proc. 20th Annual Symposium on Computer Architecture*, San Diego, CA (May):289–301.
- Herlihy, M. P., and J. Wing. 1987. Axioms for Concurrent Objects. *Proc. 14th ACM Symposium on Principles of Programming Languages* (January):13–26.
- Hernquist, L. 1987. Performance Characteristics of Tree Codes. *Astrophysics Journal Supplement* 64(August):715–734.
- Hey, A. J. G. 1991. The Genesis Distributed Memory Benchmarks. *Parallel Computing* 17:1111–1130.
- High Performance Fortran Forum. 1993. High Performance Fortran Language Specification. *Scientific Programming* 2(1):1–270.
- Hill, M. D., S. J. Eggers, J. R. Larus, G. S. Taylor, G. Adams, B. K. Bose, G. A. Gibson, P. M. Hansen, J. Keller, S. I. Kong, C. G. Lee, D. Lee, J. M. Pendleton, S. A. Ritchie, D. A. Wood, B. G. Zorn, P. N. Hilfinger, D. A Hodges, R. H. Katz, J. Ousterhout, and D. A. Patterson. 1986. Design Decisions in SPUR. *IEEE Computer* 19(10):8–22. Also in *Computers for Artificial Intelligence Processing*. Edited by B. W. Wah and C. V. Ramamoorthy. New York: John Wiley and Sons, 273–299
- Hill, M. D., and A. J. Smith. 1989. Evaluating Associativity in CPU Caches. *IEEE Transactions on Computers* C-38(12):1612–1630.
- Hillis, W. D. 1985. *The Connection Machine*. Cambridge, MA: MIT Press.
- Hillis, W. D., and G. L. Steele. 1986. Data Parallel Algorithms. *Communications of the ACM* 29(12):1170–1183.

- Hillis, W. D., and L. W. Tucker. 1993. The CM-5 Connection Machine: A Scalable Supercomputer. *Communications of the ACM* 36(11):31–40.
- Hirata, H., K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. 1992. An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads. *Proc. 19th Int'l Symposium on Computer Architecture* (May):136–145.
- Hoare, C. A. R. 1978. Communicating Sequential Processes. *Communications of the ACM* 21(8):666–667.
- Hockney, R. W., and C. R. Jesshope. 1988. *Parallel Computers 2*. London: Adam Hilger.
- Holt, C., M. Heinrich, J. P. Singh, E. Rothberg and J. L. Hennessy. 1995. *The Effects of Latency, Occupancy, and Bandwidth in Distributed Shared Memory Multiprocessors*. Tech. Report #CSL-TR-95-660, Computer Systems Laboratory, Stanford University (January).
- Homewood M., and M. McLaren. 1993. Meiko CS-2 Interconnect Elan—Elite Design. *Hot Interconnects* (August).
- Horiw, T., K. Hayashi, T. Shimizu, and H. Ishihata. 1993. Improving the AP1000 Parallel Computer Performance with Message Passing. *Proc. 20th Annual Int'l Symposium on Computer Architecture* (May):314–325
- Horowitz, M. 1997. Limits of Electrical Signalling. *Hot Interconnects Keynote* (August)
- Horst, R. 1995. TNet: A Reliable System Area Network. *IEEE Micro* 15(1):37–45.
- Horst, R. W., and T. C. K. Chou. 1985. An Architecture for High Volume Transaction Processing. *Proc. 12th Annual Int'l Symposium on Computer Architecture* (June):240–245, Boston-MA. (Tandem NonStop II)
- Horst, R. W., R. L. Harris, and R. L. Jardine. 1990. Multiple Instruction Issue in the NonStop Cyclone Processor. *Proc. Annual International Symposium on Computer Architecture (ISCA)*, 216–226.
- Hristea, C., D. Lenoski, and J. Keen. 1997. Measuring Memory Hierarchy Performance of Cache Coherent Multiprocessors Using Micro Benchmarks. *Proc. SC97* (November; all-Web conference proceeding).
- Hunt, D. 1996. *Advanced Features of the 64-Bit PA-8000*. Palo Alto, CA: Hewlett Packard Corp.
- IEEE Computer Society. 1993. *IEEE Standard for Scalable Coherent Interface (SCI)*. IEEE Standard 1596–1992. Washington, DC: IEEE Computer Society.
- . 1995. *IEEE Standard for Cache Optimization for Large Numbers of Processors Using the Scalable Coherent Interface (SCI)* Draft 0.35 (September). Washington, DC: IEEE Computer Society.
- Iftode, L., C. Dubnicki, E. W. Felten, and K. Li. 1996. Improving Release-Consistent Shared Virtual Memory Using Automatic Update. *Proc. Second Symposium on High Performance Computer Architecture* (February):14–25.
- Iftode, L., J. P. Singh, and K. Li. 1996a. Understanding Application Performance on Shared Virtual Memory Systems. *Proc. 23rd Int'l Symposium on Computer Architecture* (April):122–133.
- . 1996b. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. *Proc. Symposium on Parallel Algorithms and Architectures* (June).
- Intel Corporation. 1994. *I750, I860, I960 Processors and Related Products*. Santa Clara, CA: Intel Corp.
- . 1996. *Pentium® Pro Family Developer's Manual*. Santa Clara, CA: Intel Corp.
- Jeremiassen, T. E., and S. J. Eggers. Eliminating False Sharing. *Proc. 1991 Int'l Conference on Parallel Processing* (August):377–381.

- Jiang, D., H. Shan, and J. P. Singh. 1997. Application Restructuring and Performance Portability on Shared Virtual Memory and Hardware-Coherent Multiprocessors. *Proc. Sixth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (June):217–229.
- Jiang, D., and J. P. Singh. 1998. A Methodology and an Evaluation of the SGI Origin2000. *Proc. SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (June).
- Joe, T. 1995. *COMA-F: A Non-Hierarchical Cache Only Memory Architecture*. Ph.D. diss., Computer Systems Laboratory, Stanford University (March).
- Joe, T., and J. L. Hennessy. 1994. Evaluating the Memory Overhead Required for COMA Architectures. *Proc. 21st Int'l Symposium on Computer Architecture* (April):82–93.
- Joerg, C. F. 1994. *Design and Implementation of a Packet Switched Routing Chip*. Tech. Report MIT/LCS/TR-482, MIT Laboratory for Computer Science (August).
- Joerg, C. F., and A. Boughton. 1991. The Monsoon Interconnection Network. *Proc. ICCD* (October).
- Johnson, M. 1991. *Superscalar Microprocessor Design*. Englewood Cliffs, NJ: Prentice Hall.
- Jordan, H. F. 1985. HEP Architecture, Programming, and Performance. In *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*. Edited by J. S. Kowalik. Cambridge, MA: MIT Press, 8.
- Jouppi, N. P. 1990. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. *Proc. 17th Annual Symposium on Computer Architecture* (June):364–373.
- Jouppi, N. P., and P. Ranganathan. 1997. *The Relative Importance of Memory Latency, Bandwidth, and Branch Limits to Performance*. Workshop on Mixing Logic and DRAM: Chips that Compute and Remember. Presented at the Annual Int'l Symposium on Computer Architecture (ISCA) '97 (June).
- Jouppi, N. P., and D. Wall. 1989. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. *ASPLOS III*, 272–282.
- Kägi, A., D. Burger, and J. R. Goodman. 1997. Efficient Synchronization: Let Them Eat QOLB. *Proc. 24th Int'l Symposium on Computer Architecture* (ISCA) (June):170–180.
- Karlin, A. R., M. S. Manasse, L. Rudolph and D. D. Sleator. 1986. Competitive Snoopy Caching. *Proc. 27th Annual IEEE Symposium on Foundations of Computer Science*.
- Karol, M., M. Hluchyj, and S. Morgan. 1987. Input versus Output Queueing on a Space Division Packet Switch. *IEEE Transactions on Communications* 35(12):1347–1356.
- Karp, R., U. Vazirani, and V. Vazirani. 1990. An Optimal Algorithm for On-Line Bipartite Matching. *Proc. 22nd ACM Symposium on the Theory of Computing* (May):352–358.
- Kaxiras, S. 1996. Kiloprocessor Extensions to SCI. *Proc. 10th Int'l Parallel Processing Symposium*
- Kaxiras, S., and J. Goodman. The GLOW Cache Coherence Protocol Extensions for Widely Shared Data. *Proc. Int'l Conference on Supercomputing* (May):35–43.
- Keeton, K. K., T. E. Anderson, and D. A. Patterson. 1995. LogP Quantified: The Case for Low-Overhead Local Area Networks. *Hot Interconnects III: Symposium on High Performance Interconnects* (August).
- Keleher, P., A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. 1994. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. *Proc. Winter USENIX Conference* (January):15–132.
- Keleher, P., A. L. Cox, and W. Zwaenepoel. 1992. Lazy Consistency for Software Distributed Shared Memory. *Proc. 19th Int'l Symposium on Computer Architecture* (May):13–21.
- Kermani, P., and L. Kleinrock. 1979. Virtual Cut-Through: A New Computer Communication Switching Technique. *Computer Networks* 3(September):267–286.

- Kessler, R. E., and J. L. Schwarzmeier. 1993. Cray T3D: A New Dimension for Cray Research. *Proc. Papers, COMPCON Spring'93*, San Francisco (February):176–182.
- Knuth, D. E. 1966. Additional Comments on a Problem in Concurrent Programming Control. *Communications of the ACM* 9(5):321–322.
- Koebel, C., D. Loveman, R. Schreiber, G. Steele, and M. Zosel. 1994. *The High Performance Fortran Handbook*. Cambridge, MA: MIT Press.
- Koeninger, R. K., M. Furtney, and M. Walker. 1994. A Shared Memory MPP from Cray Research. *Digital Technical Journal* 6(2):8–21.
- Kogge, P.-M. 1994. EXECUBE—A New Architecture for Scalable MPPs. 1994 Int'l Conference on Parallel Processing (August):177–184.
- Kontothanassis, L. I., G. Hunt, R. Stets, N. Hardavellas, M. Cierniak, S. Parthasarathy, W. Meira, S. Dwarkadas, and M. Scott. 1997. VM-Based Shared Memory on Low-Latency, Remote-Memory-Access Networks. *Proc. 24th Int'l Symposium on Computer Architecture* (June).
- Kontothanassis, L. I., and M. L. Scott. 1996. Using Memory-Mapped Network Interfaces to Improve the Performance of Distributed Shared Memory. *Proc. Second Symposium on High Performance Computer Architecture* (February):166–177.
- Kostantinidou, S., and L. Snyder. 1991. Chaos Router: Architecture and Performance. *Proc. 18th Annual Symposium on Computer Architecture* (May):212–221.
- Krishnamurthy, A., K. E. Schauser, C. J. Scheiman, R. Y. Wang, D. E. Culler, and K. Yelick. 1996. Evaluation of Architectural Support for Global Address-Based Communication in Large-Scale Parallel Machines. *ACM SIGPLAN Notices* 31(9):37–48.
- Krishnamurthy, A., and K. A. Yelick. 1994. Optimizing Parallel SPMD Programs. *Seventh Annual Workshop on Languages and Compilers for Parallel Computing*. Ithaca, NY (August).
- . 1995. Optimizing Parallel Programs with Explicit Synchronization. *Programming Language Design and Implementation*, 196–204.
- . 1996. Analyses and Optimizations for Shared Address Space Programs. *JPDC* 38(2):130–144.
- Kroft, D. 1981. Lockup-Free Instruction Fetch/Prefetch Cache Organization. *Proc. Eighth Int'l Symposium on Computer Architecture* (May):81–87.
- Kronenberg, N. P., H. Levy, and W. D. Strecker. 1986. Vax Clusters: A Closely-Coupled Distributed System. *ACM Transactions on Computer Systems* 4(2):130–146.
- Kruskal, C. P., and M. Snir. 1983. The Performance of Multistage Interconnection Networks for Multiprocessors. *IEEE Transactions on Computers* C-32(12):1091–1098.
- Kubiatowicz, J., and A. Agarwal. 1993. The Anatomy of a Message in the Alewife Multiprocessor. *Proc. Int'l Conference on Supercomputing* (July):195–206.
- Kuehn, J. T., and B. J. Smith. 1988. The Horizon Supercomputing System: Architecture and Software. *Proc. Supercomputing '88* (November):28–34.
- Kumar, M. 1992. Unique Design Concepts in GF11 and Their Impact on Performance. *IBM Journal of Research and Development* 36(6):990–1000.
- Kumar, V., A. Grama, A. Gupta, and G. Karypis. 1994. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City, CA: Benjamin/Cummings Publishing Company.
- Kumar, V., and A. Gupta. 1991. Analysis of Scalability of Parallel Algorithms and Architectures: A Survey. *Proc. Int'l Conference on Supercomputing* (June):396–405.

- Kung, H. T., R. Sansom, S. Schlick, P. A. Steenkiste, M. Arnould, F. J. Bitz, F. Christianson, E. C. Cooper, O. Menzilcioglu, D. Ombres, and B. Zill. 1989. Network-Based Multicomputers: An Emerging Parallel Architecture. *Proc. Supercomputing '91 Conference* (November):664–673.
- Kurihara, K., D. Chaiken, and A. Agarwal. 1991. Latency Tolerance through Multithreading in Large-Scale Multiprocessors. *Proc. Int'l Symposium on Shared Memory Multiprocessing* (April):91–101.
- Kuskin, J., D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. 1994. The Stanford FLASH Multiprocessor. *Proc. 21st Int'l Symposium on Computer Architecture* (April):302–313.
- Lam, M. S., and R. P. Wilson. 1992. Limits on Control Flow on Parallelism. *Proc. 19th Annual Int'l Symposium on Computer Architecture* (May):46–57.
- Lamport, L. 1979. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers* C-28(9):690–691.
- Larus, J. R., B. Richards, and G. Viswanathan. 1996. Parallel Programming in C++: A Large-Grain Data-Parallel Programming Language. In *Parallel Programming Using C++*. Edited by G. V. Wilson and P. Lu. Cambridge, MA: MIT Press.
- Laudon, J. P. 1994. *Architectural and Implementation Tradeoffs in Multiple-Context Processors*. Ph.D. diss., Stanford University, Stanford, California. Also published as Tech. Report #CSL-TR-94-634, Computer Systems Laboratory, Stanford University (May).
- Laudon, J., A. Gupta, and M. Horowitz. 1994. *Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors*. In *Multithreaded Computer Architecture: A Summary of the State of the Art*. Edited by R. A. Iannucci. Dordrecht, Germany; Norwell, MA: Kluwer Academic Publishers, 167–200.
- Laudon, J. P., and D. Lenoski. 1997. The SGI Origin: A ccNUMA Highly Scalable Server. *Proc. 24th Int'l Symposium on Computer Architecture*.
- Lawton, J. V., J. J. Brosnan, M. P. Doyle, S. D. O'Riodain, and T. G. Reddin. 1996. Building a High-Performance Message-Passing System for MEMORY CHANNEL Clusters. *Digital Technical Journal* 8(2):96–116.
- Lee, C. G. 1989. Multi-Step Gradual Rounding. *IEEE Transactions on Computers* 38(4):595–600.
- Lee, R. L., A. Y. Kwok, and F. A. Briggs. 1991. The Floating Point Performance of a Superscalar SPARC Processor. *Proc. 4th Symposium on Architectural Support for Programming Languages and Operating Systems* (April):28–37.
- Leighton, F. T. 1992. *Introduction to Parallel Algorithms and Architectures*. San Francisco: Morgan Kaufmann.
- Leiserson, C. E. 1985. Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing. *IEEE Transactions on Computers* C-34(10):892–901.
- Leiserson, C. E., Z. S. Abuhamdeh, D. C. Douglas, C. R. Feynman, M. N. Ganmukhi, J. V. Hill, W. D. Hillis, B. C. Kuszmaul, M. A. St. Pierre, D. S. Wells, M. C. Wong, S. Yang, and R. Zak. 1996. The Network Architecture of the Connection Machine CM-5. *Journal of Parallel and Distributed Computing* 33(2):145–158. Also in *Proc. Fourth Symposium on Parallel Algorithms and Architectures '92* (June):272–285.
- Lenoski, D. 1992. *The Stanford DASH Multiprocessor*. Ph.D. diss., Computer Systems Laboratory, Stanford University.
- Lenoski, D., J. Laudon, K. Gharachorloo, A. Gupta, and J. L. Hennessy. 1990. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. *Proc. 17th Int'l Symposium on Computer Architecture* (May):148–159.

- Lenoski, D., J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. L. Hennessy. 1992. The DASH Prototype: Implementation and Performance. *Proc. 19th Int'l Symposium on Computer Architecture*, Gold Coast, Australia (May):92–103.
- . 1993. The DASH Prototype: Logic Overhead and Performance. *IEEE Transactions on Parallel and Distributed Systems* 4(1):41–61.
- Li, K., and P. Hudak. 1989. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems* 7(4):321–359.
- Li, S.-Y. 1988. Theory of Periodic Contention and Its Application to Packet Switching. *Proc. INFOCOM '88* (March):320–325.
- Lim, B.-H., and A. Agarwal. 1994. Reactive Synchronization Algorithms for Multiprocessors. *Proc. Sixth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 25–35.
- Linder, D., and J. Harden. 1991. An Adaptive Fault Tolerant Wormhole Strategy for k-ary n-cubes. *IEEE Transactions on Computer C-40(1)*:2–12.
- Lipton, R., and J. Sandberg. 1988. PRAM: A Scalable Shared Memory. Tech. Report #CS-TR-180-88, Computer Science Dept., Princeton University (September).
- Litzkow, M., M. Livny, and M. W. Mutka. 1988. Condor—A Hunter of Idle Workstations. *Proc. Eighth Int'l Conference on Distributed Computing Systems* (June):104–111.
- Lo, J. L., S. J. Eggers, J. S. Emer, H. M. Levy, R. L. Stamm, and D. M. Tullsen. 1997. Converting Thread-Level Parallelism into Instruction-Level Parallelism via Simultaneous Multithreading. *ACM Transactions on Computer Systems* (August).
- Lonergan, W., and P. King. 1961. Design of the B 5000 System. *Datamation* 7(5):28–32.
- Lovett, T., and R. Clapp. 1996. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. *Proc. 23rd Int'l Symposium on Computer Architecture* (May):308–317.
- Luk, C.-K., and T. C. Mowry. 1996. Compiler-Based Prefetching for Recursive Data Structures. *Proc. Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)* (October):222–233.
- Lukowsky, J., and S. Polit. 1997 (date accessed). IP Packet Switching on the GIGAswitch/FDDI System. <http://www.networks.digital.com:80/dr/techart/gsip-mn.html>.
- Mainwaring, A., B. Chun, S. Schleimer, and D. Wilkerson. 1997. System Area Network Mapping. *Proc. Ninth Annual ACM Symposium on Parallel Algorithms and Architecture*, Newport, RI (June):116–126.
- Mainwaring, A., and D. E. Culler. 1996. Active Message Applications Programming Interface and Communication Subsystem Organization. Tech. Report CSD-96-918, University of California at Berkeley.
- Martin, R. 1994. HPAM: An Active Message Layer of a Network of Workstations. Presented at *Hot Interconnects II* (August).
- Massalin, H., and C. Pu. 1991. A Lock-Free Multiprocessor OS Kernel. Tech. Report CUCS-005-01, Columbia University, Computer Science Dept. (October).
- Matelan, N. 1985. The FLEX/32 Multicomputer. *Proc. 12th Annual Int'l Symposium on Computer Architecture*, Boston, MA. (Flex) (June):209–213.
- May, C., E. Silha, R. Simpson, and H. Warren, eds. 1994. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. San Francisco: Morgan Kaufmann.
- McCreight, E. 1984. *The Dragon Computer System: An Early Overview*. Tech. Report, Xerox Corp. (September).
- Mellor-Crummey, J., and M. Scott. 1991. Algorithms for Scalable Synchronization on Shared Memory Multiprocessors. *ACM Transactions on Computer Systems* 9(1):21–65.

- Melvin, S., and Y. Patt. 1991. Exploiting Fine-Grained Parallelism through a Combination of Hardware and Software Techniques. *Proc. Annual Int'l Symposium on Computer Architecture (ISCA)*, 287–296.
- Michael, M., and M. Scott. 1996. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. *Proc. 15th Annual ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA (May): 267–276.
- Minnich, R., D. Burns, and F. Hady. 1995. The Memory-Integrated Network Interface. *IEEE Micro* 15(1):11–20.
- MIPS Technologies. 1991. *MIPS R4000 User's Manual*. Mountain View, CA: MIPS Technologies.
- . 1996. *R10000 Microprocessor User's Manual, Version 1.1* (January). Mountain View, CA: MIPS Technologies.
- Miyoshi, H.; M. Fukuda, T. Iwamiya, T. Nakamura, M. Tuchiya, M. Yoshida, K. Yamamoto, Y. Yamamoto, S. Ogawa, Y. Matsuo, T. Yamane, M. Takamura, M. Ikeda, S. Okada, Y. Sakamoto, T. Kitamura, H. Hatama, M. Kishimoto, M. Arnould, F. J. Bitz, E. C. Cooper, H. T. Kung, R. Sansom, S. Schlick, P. A. Steenkiste, and B. Zill. 1994. Development and Achievement of NAL Numerical Wind Tunnel (NWT) for CFD Computations. *Proc. Supercomputing '94*, Washington, DC (November):685–692.
- Mowry, T. C. 1994. *Tolerating Latency through Software-Controlled Data Prefetching*. Ph.D. diss., Computer Systems Laboratory, Stanford University. Also published as Tech. Report #CSL-TR-94-628, Computer Systems Laboratory, Stanford University (June).
- MPI Forum. 1993. *Document for a Standard Message-Passing Interface*. Tech. Report CS-93-214, University of Tennessee, Knoxville, Computer Science Dept. (November).
- . 1994. MPI: A Message Passing Interface. *Int'l Journal of Supercomputing Applications* 8(3/4). Special Issue on MPI. (updated 5/95). Also published in *Proc. Supercomputing '93 Conference* (May). Los Alamitos, CA: IEEE Computer Society Press, 878–883. Updated spec at <http://www.mcs.anl.gov/mpib/>.
- Mukherjee, S., and M. Hill. 1997. A Case for Making Network Interfaces Less Peripheral. *Hot Interconnects* (August).
- NAS Parallel Benchmarks. 1998 (date accessed). <http://science.nas.nasa.gov/Software/NPB/>.
- Nayfeh, B. A., L. Hammond, K. Olukoton. 1996. Evaluation of Design Alternatives for a Multiprocessor Microprocessor. *Proc. 23rd Annual Int'l Symposium on Computer Architecture* (May). New York: ACM Press, 67–77.
- Nestle, E., and A. Inselberg. 1985. The Synapse N+1 System: Architectural Characteristics and Performance Data of a Tightly-Coupled Multiprocessor System. *Proc. 12th Annual Int'l Symposium on Computer Architecture*, Boston, MA (Synapse) (June):233–239.
- Ngai, J., and C. Seitz. 1989. A Framework for Adaptive Routing in Multicomputer Networks. *Proc. 1989 Symposium on Parallel Algorithms and Architectures* (June):2–10.
- Nickolls, J. R. 1990. The Design of the MasPar MP-1: A Cost Effective Massively Parallel Computer. *COMPCON Spring '90, Digest of Papers*, San Francisco, CA (February/March):25–28.
- Nikhil, R. S., and Arvind. 1989. Can Dataflow Subsume von Neumann Computing? *Proc. 16th Annual Int'l Symposium on Computer Architecture* (May):262–72.
- Nikhil, R., G. Papadopoulos, and Arvind. 1993. *T: A Multithreaded Massively Parallel Architecture. *Proc. Annual Int'l Symposium on Computer Architecture (ISCA)* '93 (May):156–167.
- Noakes, M. D., D. A. Wallach, and W. J. Dally. 1993. The J-Machine Multicomputer: An Architectural Evaluation. *Proc. 20th Int'l Symposium on Computer Architecture* (May):224–235.

- Nuth, P., and W. J. Dally. 1992. The J-Machine Network. *Proc. Int'l Conference on Computer Design: VLSI in Computers and Processors* (October).
- . 1995. The Named-State Register File: Implementation and Performance. *Proc. First Int'l Symposium on High-Performance Computer Architecture* (January):4–13.
- O'Krafska, B., and A. Newton. 1990. An Empirical Evaluation of Two Memory-Efficient Directory Methods. *Proc. 17th Int'l Symposium on Computer Architecture* (May):138–147.
- Office of Science and Technology Policy. 1993. *Grand Challenges 1993: High Performance Computing and Communications, A Report by the Committee on Physical, Mathematical, and Engineering Sciences*. Washington, DC: Office of Science and Technology Policy.
- Ohara, M. 1996. *Producer-Oriented versus Consumer-Oriented Prefetching: A Comparison and Analysis of Parallel Application Programs*. Ph.D. diss., Computer Systems Laboratory, Stanford University. Available as Tech. Report #CSL-TR-96-695, Stanford University (June).
- Olukotun, K., B. A. Nayfeh, L. Hammond, K. Wilson and K. Chang. 1996. The Case for a Single-Chip Multiprocessor. *Proc. ASPLOS* (October):2–11.
- Omondi, A. R. 1994. Ideas for the Design of Multithreaded Pipelines. In *Multithreaded Computer Architecture: A Summary of the State of the Art*. Edited by R. Iannucci. Dordrecht, Germany; Norwell, MA: Kluwer Academic Publishers, 1994. See also A. R. Omondi, Design of a High Performance Instruction Pipeline. *Computer Systems Science and Engineering* 6(1):13–29 (1991).
- Pacheco, P. 1996. *Parallel Programming with MPI*. San Francisco: Morgan Kaufmann.
- Padegs, A. 1981. System/360 and Beyond. *IBM Journal of Research and Development* 25(5):377–390.
- Pai, V. S., P. Ranganathan, S. V. Adve, and T. Harton. 1996. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. *Proc. Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLÖS-VII)* (October):12–23.
- Papadimitriou, C. H. 1979. The Serializability of Concurrent Database Updates. *Journal of the ACM* 26(4):631–653.
- Papadopoulos, G. M., and D. E. Culler. 1990. Monsoon: An Explicit Token-Store Architecture. *Proc. 17th Annual Int'l Symposium on Computer Architecture*, Seattle, WA (May):82–91.
- Papamarcos, M., and J. Patel. 1984. A Low Overhead Coherence Solution for Multiprocessors with Private Cache Memories. *Proc. 11th Annual Int'l Symposium on Computer Architecture* (June):348–354.
- PARKBENCH Committee. 1994. Public International Benchmarks for Parallel Computers. *Scientific Programming* 3(2). Also published as Tech. Report CS93-213, University of Tennessee, Knoxville, Dept. of Computer Science (November).
- Patterson, D. A. 1995. Microprocessors in 2020. *Scientific American* (September).
- Patterson, D. A., T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, and K. Yellick. 1997. A Case for Intelligent RAM. *IEEE Micro* 17(2):34–44.
- Peterson, L., and B. Davie. 1996. *Computer Networks*. San Francisco: Morgan Kaufmann.
- Pfeiffer, W., S. Hotovy, N. Nystrom, D. Rudy, T. Sterling, M. Straka. 1995 (date accessed). JNNIE: The Joint NSF-NASA Initiative on Evaluation. <http://www.tc.cornell.edu/JNNIE/finrep/jnnie.html>.
- Pfister, G. F. 1995. *In Search of Clusters—The Coming Battle for Lowly Parallel Computing*. Englewood Cliffs, NJ: Prentice Hall.

- Pfister, G. F., W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliff, E. A. Melton, V. A. Norton, and J. Weiss. 1985. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. *Proc. Int'l Conference on Parallel Processing* (August):264–771.
- Pfister, G. F., and V. A. Norton. 1985. Hot Spot Contention and Combining Multistage Interconnection Networks. *IEEE Transactions on Computers* C-34(10).
- Pierce, P. 1988. The NX/2 Operating System. *Proc. Third Conference on Hypercube Concurrent Computers and Applications* (January):384–390.
- Pierce, P., and G. Regnier. 1994. The Paragon Implementation of the NX Message Passing Interface. *Proc. Scalable High-Performance Computing Conference* (May):184–90.
- Porter, R. E. 1960. *Datamation* 6(1):8–14.
- Przybylski, S., M. Horowitz, J. L. Hennessy. 1988. Performance Tradeoffs in Cache Design. *Proc. 15th Annual Symposium on Computer Architecture* (May):290–298.
- Ranganathan, P., V. S. Pai, H. Abdel-Shafii, and S. V. Adve. 1997. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. *Proc. 24th Int'l Symposium on Computer Architecture* (June).
- Ratner, J. 1985. Concurrent Processing: A New Direction in Scientific Computing. *Proc. 1985 National Computing Conference*, 835.
- Reddaway, S. F. 1973. DAP—A Distributed Array Processor. *First Annual Int'l Symposium on Computer Architecture* (December):61–65.
- Reinhardt, S. K., M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. 1993. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. *Proc. ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (May):48–60.
- Reinhardt, S. K., J. R. Larus, and D. A. Wood. 1994. Tempest and Typhoon: User-Level Shared Memory. *Proc. 21st Int'l Symposium on Computer Architecture* (April):325–337.
- Reinhardt, S. K., R. W. Pfile, and D. A. Wood. 1996. Decoupled Hardware Support for Distributed Shared Memory. *Proc. 23rd Int'l Symposium on Computer Architecture* (May):34–43.
- Rettberg, R., W. Crowther, P. Carvey, and R. Tomlinson. 1990. The Monarch Parallel Processor Hardware Design. *IEEE Computer* (April):18–30.
- Rettberg, R., and R. Thomas. 1986. Contention is No Obstacle to Shared-Memory Multiprocessing. *Communications of the ACM* 29(12):1202–1212.
- Rinard, M. C., D. J. Scales, and M. S. Lam. 1993. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer* 26(6).
- Rodgers, D. 1985. Improvements on Multiprocessor System Design. *Proc. 12th Annual Int'l Symposium on Computer Architecture*, Boston, MA (Sequent B8000) (June):225–231.
- Rosenblum, M., S. A. Herrod, E. Witchel, and A. Gupta. 1995. Complete Computer Simulation: The SimOS Approach. *IEEE Parallel and Distributed Technology* 3(4).
- Rosenburg, B. 1989. Low-Synchronization Translation Lookaside Buffer Consistency in Large-Scale Shared-Memory Multiprocessors. *Proc. Symposium on Operating Systems Principles* (December).
- Rothberg, E., J. P. Singh, and A. Gupta. 1993. Working Sets, Cache Sizes, and Node Granularity Issues for Large-Scale Multiprocessors. *Proc. 20th Int'l Symposium on Computer Architecture* (May):14–25.
- Russel, R. M. 1978. The CRAY-1 Computer System. *Communications of the ACM* 21(1):63–72.
- Saavedra-Barrera, R. H., D. E. Culler, T. von Eicken. 1990. Analysis of Multithreaded Architectures for Parallel Computing. *Second Annual ACM Symposium on Parallel Algorithms and Architectures* (July):169–178.

- Saavedra, R. H., R. S. Gaines, and M. J. Carlton. 1993. Micro Benchmark Analysis of the KSR1. *Proc. Supercomputing '93*, Portland, OR (November):202–213.
- Saavedra, R. H., and A. J. Smith. 1996. Analysis of Benchmark Characteristics and Benchmark Performance Prediction. *ACM Transactions on Computer Systems* 14(4):344–384.
- Sakai, S., Y. Kodama, and Y. Yamaguchi. 1991. Prototype Implementation of a Highly Parallel Dataflow Machine EM4. *Proc. Fifth Int'l Parallel Processing Symposium*, Anaheim, CA (April/May):278–286.
- Salmon, J. 1990. *Parallel Hierarchical N-body Methods*. Ph.D. diss., California Institute of Technology.
- Salmon, J. K., M. S. Warren, and G. S. Winckelmans. 1994. Fast Parallel Treecodes for Gravitational and Fluid Dynamical N-body Problems. *Intl. Journal of Supercomputer Applications* 8:129–142.
- Samanta, R., A. Bilas, L. Iftode, and J. P. Singh. 1998. Home-Based SVM Protocols for SMP Clusters: Design, Simulations, Implementation, and Performance. *Proc. 23rd Annual Int'l Symposium on Computer Architecture* (February).
- Saulsbury, A., F. Pong, and A. Nowatzky. 1996. Missing the Memory Wall: The Case for Processor/Memory Integration. *Proc. 23rd Annual Int'l Symposium on Computer Architecture* (May):90–101.
- Saulsbury, A., T. Wilkinson, J. Carter, and A. Landin. 1995. An Argument For Simple COMA. *Proc. First IEEE Symposium on High Performance Computer Architecture* (January):276–285.
- Savage, J. 1985. Parallel Processing as a Language Design Problem. *Proc. 12th Annual Int'l Symposium on Computer Architecture*, Boston, MA (Myrias 4000) (June):221–224.
- Scales, D. J., K. Gharachorloo, and C. A. Thekkath. 1996. Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory. *Proc. Seventh Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):174–185.
- Scales, D. J., and M. S. Lam. 1994. The Design and Evaluation of a Shared Object System for Distributed Memory Machines. *Proc. First Symposium on Operating System Design and Implementation* (November):101–114.
- Schanin, D.J. 1986. The Design and Development of a Very High Speed System Bus—The Encore Multimax Nanobus. In *Proc. Fall Joint Computer Conference (Encore)*, Dallas, TX (November). Edited by H. S. Stone. Los Alamitos: IEEE Computer Society Press, 410–418.
- Schauser, K. E., and C. J. Scheiman. 1995. Experience with Active Messages on the Meiko CS-2. *Proc. Ninth Int'l Symposium on Parallel Processing (IPPS'95)* (April):140–149.
- Scheurich, C. and M. Dubois. 1987. Correct Memory Operation of Cache-Based Multiprocessors. *Proc. 14th Int'l Symposium on Computer Architecture* (June):234–243.
- Schoinias, I., B. Falsafi, A. R. Lebeck, S. K. Reinhardt, J. R. Larus, and D. A. Wood. 1994. Fine-Grain Access Control for Distributed Shared Memory. *Proc. 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):297–306.
- Schroeder, M. D., A. D. Birrell, M. Burrows, H. Murray, R. M. Needham, T. L. Rodeheffer, E. H. Satterthwaite, and C. P. Thacker. 1991. Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links. *IEEE Journal on Selected Areas in Communications* 9(8):1318–1335.
- Schwiebert, L., and D..N. Jayasimha. 1995. A Universal Proof Technique for Deadlock-Free Routing in Interconnection Networks. *Symposium on Parallel Algorithms and Architecture* (July):175–184.
- Scott, S. 1991. A Cache-Coherence Mechanism for Scalable Shared-Memory Multiprocessors. *Proc. Int'l Symposium on Shared Memory Multiprocessing* (April):49–59.

- _____. 1996. Synchronization and Communication in the T3E Multiprocessor. *Proc. Seventh Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):26–36, Cambridge, MA.
- Scott, S., and J. R. Goodman. 1993. Performance of Pruning Cache Directories for Large-Scale Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 4(5):520–534.
- _____. 1994. The Impact of Pipelined Channels on k-ary n-Cube Networks. *IEEE Transactions on Parallel and Distributed Systems* 5(1):2–16.
- Scott, S., M. Vernon, and J. R. Goodman. 1992. Performance of the SCI Ring. *Proc. 19th Int'l Symposium on Computer Architecture* (May):403–414.
- Seitz, C. L. 1984. Concurrent VLSI Architectures. *IEEE Transactions on Computers* 33(12):1247–1265.
- _____. 1985. The Cosmic Cube. *Communications of the ACM* 28(1):22–33.
- Seitz, C. L., and W.-K. Su. 1993. A Family of Routing and Communication Chips Based on Moasic. *Proc. of Univ. of Washington Symposium on Integrated Systems*. Cambridge, MA: MIT Press, 320–337.
- Shah, G., J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. 1998. Performance and Experience with LAPI—A New High-Performance Communication Library for the IBM RS/6000 SP. *Twelfth Int'l Parallel Processing Symposium* (March):260–266.
- Shasha, D., and M. Snir. 1988. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Operating Systems* 10(2):282–312.
- Shimada, T., K. Hiraki, and K. Nishida. 1984. An Architecture of a Data Flow Machine and Its Evaluation. *Proc. COMPCON '84*, 486–90.
- Simoni, R., and M. Horowitz. 1991. Dynamic Pointer Allocation for Scalable Cache Coherence Directories. *Proc. Int'l Symposium on Shared Memory Multiprocessing* (April): 72–81.
- Sindhu, P., J.-M. Frailong, and M. Cekleov. 1991. *Formal Specification of Memory Models*. Tech. Report (PARC) CSL-91-11. Xerox Corp., Palo Alto Research Center, Palo Alto, CA.
- Sindhu, P., et al. 1993. XDBus: A High-Performance, Consistent, Packet Switched VLSI Bus. *Proc. COMPCON* (Spring): 338–344.
- Singh, J. P. 1993. *Parallel Hierarchical N-body Methods and Their Implications for Multiprocessors*. Ph.D. diss., Tech. Report #CSL-TR-93-565, Stanford University (March).
- _____. 1998. *Some Aspects of Controlling Scheduling in Hardware Control Prefetching*. To be published as Tech. Report, Princeton University, Computer Science Dept.
- Singh, J. P., A. Gupta, and M. Levoy. 1994. Parallel Visualization Algorithms: Performance and Architectural Implications. *IEEE Computer* 27(6).
- Singh, J. P., J. L. Hennessy, and A. Gupta. 1993. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer* 26(7):42–50.
- _____. 1995. Implications of Parallel Hierarchical N-body Applications for Multiprocessors. *ACM Transactions on Computer Systems* (May).
- Singh, J. P., C. Holt, T. Totsuka, A. Gupta, and J. L. Hennessy. 1995. Load Balancing and Data Locality in Hierarchical N-body Methods: Barnes-Hut, Fast Multipole and Radiosity. *Journal of Parallel and Distributed Computing* (June).
- Singh, J. P., T. Joe, A. Gupta, and J. L. Hennessy. 1993. An Empirical Comparison of the KSR-1 and DASH Multiprocessors. *Proc. Supercomputing '93* (November).
- Singh, J. P., E. Rothberg, and A. Gupta. 1994. Modeling Communication in Parallel Algorithms: A Fruitful Interaction between Theory and Systems? *Proc. 10th Annual ACM Symposium on Parallel Algorithms and Architectures*.

- Singh, J. P., W.-D. Weber, and A. Gupta. 1992. SPLASH: The Stanford Parallel Applications for SHared Memory. *Computer Architecture News* 20(1):5–44.
- Sites, R. L., ed. 1992. *Alpha Architecture Reference Manual*. Hudson, MA: Digital Press, Digital Equipment Corp.
- Slater, M. 1994. Intel Unveils Multiprocessor System Specification. *Microprocessor Report* (May):12–14.
- Slotnick, D. L. 1967. Unconventional Systems. *Proc. AFIPS Spring Joint Computer Conference* 30: 477–481.
- Slotnick, D. L., W. C. Borck, and R. C. McReynolds. 1962. The Solomon Computer. *Proc. AFIPS Fall Joint Computer Conference* 22: 97–107.
- Smith, A. J. 1982. Cache Memories. *ACM Computing Surveys* 14(3):473–530.
- Smith, B. J. 1981. Architecture and Applications of the HEP Multiprocessor Computer System. *Proc. SPIE: Real-Time Signal Processing IV* 298(August):241–248.
- . 1985. The Architecture of HEP. In *Parallel MIMD Computation: The HEP Supercomputer and Its Applications*. Edited by J.S. Kowalik. Cambridge, MA: MIT Press, 41–55.
- Smith, M. D., M. Johnson, and M. A. Horowitz. 1989. Limits on Multiple Instruction Issue. *Proc. Third Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, 290–302, Apr.
- Snir, M., S. Otto, S. H. Lederman, D. Walker, and J. Dongarra. 1995. *MPI: The Complete Reference*. Cambridge, MA: MIT Press.
- Sohi, G., S. Breach, and T. N. Vijaykumar. 1995. Multiscalar Processors. *Proc. 22nd Annual Int'l Symposium on Computer Architecture* (June):414–425.
- SPEC (Standard Performance Evaluation Corporation). 1995 (date accessed). <http://www.specbench.org/>. (SPEC Benchmark Suite Release 1.0., 1989)
- Spertus, E., S. C. Goldstein, K. E. Schauser, T. von Eicken, D. E. Culler, W. J. Dally. 1993. Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5. *Proc. 20th Annual Symposium on Computer Architecture* (May):302–313
- Stenstrom, P., T. Joe, and A. Gupta. 1992. Comparative Performance Evaluation of Cache-Coherent NUMA and COMA Architectures. *Proc. 19th Int'l Symposium on Computer Architecture* (May):80–91.
- Stets, R., S. Dwarkadas, N. Hardavellas, G. Hunt, L. Kontothanassis, S. Parthasarathy, and M. Scott. 1997. Cashmere-2L: Software Coherent Shared Memory on a Clustered Remote-Write Network. *Proc. 16th ACM Symposium on Operating Systems Principles* (October).
- Stone, H. S. 1970. A Logic-in-Memory Computer. *IEEE Transactions on Computers* C-19(1):73–78.
- Stunkel, C. B., D. G. Shea, D. G. Grice, P. H. Hochschild, and M. Tsao. 1994. The SP-1 High Performance Switch. *Proc. Scalable High Performance Computing Conference* (May):150–157 Knoxville, TN.
- Stunkel, C. B., et al. 1998 (date accessed). *The SP2 Communication Subsystem*. <http://ibm.tc.cornell.edu/ibm/pps/doc/css/css.ps>.
- SUN Microsystems. 1991. *The SPARC Architecture Manual*. #800-199-12, Version 8 (January). Mountain View, CA: SUN Microsystems.
- Sunderam, V. S. 1990. PVM: A Framework for Parallel Distributed Computing. *Concurrency: Practice and Experience* 2(4):315–339.
- Sunderam, V. S., J. Dongarra, A. Geist, and R. Manchek. 1994. The PVM Concurrent Computing System: Evolution, Experiences, and Trends. *Parallel Computing* 20(4):531–547.

- Swan, R. J., A. Bechtolsheim, K.-W. Lai, and J. K. Ousterhout. 1977. The Implementation of the CM* Multi-Microprocessor. *Proc. AFIPS Conference/National Computer Conference* (46):645–655.
- Swan, R. J., S. H. Fuller, and D. P. Siewiorek. 1977. CM*—A Modular, Multi-Microprocessor. *Proc. AFIPS Conference/National Computer Conference* (46):637–44.
- Sweazey, P., and A. J. Smith. 1986. A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus. *Proc. 13th Int'l Symposium on Computer Architecture* (May):414–423.
- Tamir, Y., and G. L. Frazier. 1988. High-Performance Multi-Queue Buffers for VLSI Communication Switches. *Proc. 15th Annual Int'l Symposium on Computer Architecture*, 343–354.
- Tanenbaum, A. S., and A. S. Woodhull. 1997. *Operating System Design and Implementation* 2nd ed. Englewood Cliffs, NJ: Prentice Hall.
- Tang, C. 1976. Cache Design in a Tightly Coupled Multiprocessor System. *Proc. AFIPS Conference* (June):749–753.
- Teller, P. 1990. Translation-Lookaside Buffer Consistency. *IEEE Computer* 23(6):26–36.
- Thacker, C., L. Stewart, and E. Satterthwaite, Jr. 1988. Firefly: A Multiprocessor Workstation. *IEEE Transactions on Computers* 37(8):909–20.
- Thapar, M., and B. Delagi. 1990. Stanford Distributed-Directory Protocol. *IEEE Computer* 23(6):78–80.
- Thekkath, R., A. P. Singh, J. P. Singh, J. Hennessy, and S. John. 1997. An Application-Driven Evaluation of the Convex Exemplar SP-1200. *Proc. Int'l Parallel Processing Symposium* (June).
- Thompson, M., J. Barton, T. Jermoluk, and J. Wagner. 1988. Translation Lookaside Buffer Synchronization in a Multiprocessor System. *Proc. USENIX Technical Conference* (February).
- Thornton, J. E. 1964. Parallel Operation in the Control Data 6600. *AFIPS Proc. Fall Joint Computer Conference*, Part 2 26:33–40. Reprinted in Siewiorek, Bell, and Newell. 1982. *Computer Structures: Principles and Examples*. New York: McGraw-Hill.
- Tomasulo, R. M. 1967. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development* 11(1):25–33.
- Torrellas, J., M. S. Lam, and J. L. Hennessy. 1994. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers* 43(6):651–663.
- Transaction Processing Council. 1998. <http://www.tpc.org>
- Traw, C., and J. Smith. 1991. A High-Performance Host Interface for ATM Networks. *Proc. ACM SIGCOMM Conference* (September):317–325.
- Traylor, R., and D. Dunning. 1992. Routing Chip Set for Intel Paragon Parallel Supercomputer. *Proc. Hot Chips '92 Symposium* (August).
- Tucker, L. W., and A. Mainwaring. 1994. CMMD: Active Messages on the CM-5. *Parallel Computing* 20(4):481–496.
- Tucker, L. W., and G. G. Robertson. 1988. Architecture and Applications of the Connection Machine. *IEEE Computer* 21(8):26–38.
- Tucker, S. 1986. The IBM 3090 System: An Overview. *IBM Systems Journal* 25(1):4–19.
- Tullsen, D. M., and S. J. Eggers. 1993. Limitations of Cache Prefetching on a Bus-Based Multiprocessor. *Proc. 20th Annual Symposium on Computer Architecture* (May):278–288.
- Tullsen, D. M., S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. 1996. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. *Proc. 23rd Int'l Symposium on Computer Architecture* (May):191–202.

- Tullsen, D. M., S. J. Eggers, and H. M. Levy. 1995. Simultaneous Multithreading: Maximizing On-Chip Parallelism. *Proc. 20th Annual Symposium on Computer Architecture* (June):278–288.
- Turner, J. S. 1988. Design of a Broadcast Packet Switching Network. *IEEE Transactions on Communication* 36(6):734–743.
- Valiant, L. G. 1990. A Bridging Model for Parallel Computation. *Communications of the ACM* 33(8): 103–111.
- Valois, J. 1995. Lock-Free Linked Lists Using Compare-and-Swap. *Proc. 14th Annual ACM Symposium on Principles of Distributed Computing*, Ottawa, Canada (August):214–222.
- Vick, C. R., and J. A. Cornell. 1978. PEPE Architecture—Present and Future. *Proc. AFIPS Conference* 47:981–1002.
- von Eicken, T., A. Basu, and V. Buch. 1995. Low-Latency Communication Over ATM Using Active Messages. *IEEE Micro* 15(1):46–53.
- von Eicken, T., D. E. Culler, S. C. Goldstein, and K. E. Schauser. 1992. Active Messages: A Mechanism for Integrated Communication and Computation. *Proc. 19th Annual Int'l Symposium on Computer Architecture*, Gold Coast, Australia (May):256–266.
- Vranesic, Z., M. Stumm, D. Lewis, and R. White. 1991. Hector: A Hierarchically Structured Shared Memory Multiprocessor. *IEEE Computer* 24(1):72–78.
- Wall, D. W. 1991. Limits of Instruction-Level Parallelism. *ASPLOS IV* (April):176–188.
- Wallach, D. A. 1992. PHD: A Hierarchical Cache Coherence Protocol. S.M. thesis, Massachusetts Institute of Technology. Also available as Tech. Report #1389, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Boston, MA (August).
- Wang, W.-H., J.-L. Baer and H. M. Levy. 1989. Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy. *Proc. 16th Annual Int'l Symposium on Computer Architecture* (June):140–148.
- Warren, M. S., and J. K. Salmon. 1993. A Parallel Hashed Oct-Tree N-body Algorithm. *Proc. Supercomputing '93*. Washington, DC: IEEE Computer Society, 12–21.
- Weaver, D., and T. Germond, eds. 1994. *The SPARC Architecture Manual*. SPARC International, Version 9. Englewood Cliffs, NJ: Prentice Hall.
- Weber, W.-D. 1993. *Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors*. Ph.D. diss., Computer Systems Laboratory, Stanford University (January). Also available as Tech. Report #CSL-TR-93-557, Stanford University.
- Weber, W.-D., S. Gold, P. Helland, T. Shimizu, T. Wicki, and W. Wilcke. 1997. The Mercury Interconnect Architecture: A Cost-Effective Infrastructure for High-Performance Servers. *Proc. 24th Int'l Symposium on Computer Architecture* (June):98–107.
- Weiss, S. and J. Smith. 1994. *Power and PowerPC*. San Francisco: Morgan Kaufmann.
- Widdoes, L., Jr., and S. Correll. 1980. The S-1 Project: Developing High Performance Computers. *Proc. COMPCON* (Spring):282–291.
- Wilson, A., Jr. 1987. Hierarchical Cache / Bus Architecture for Shared Memory Multiprocessors. *Proc. 14th Int'l Symposium on Computer Architecture* (June):244–252.
- Wolf, M. E., and M. S. Lam. 1991. A Data Locality Optimizing Algorithm. *Proc. ACM SIGPLAN'91 Conference on Programming Language Design and Implementation* (June):30–44.
- Wolfe, M. 1989. *Optimizing Supercompilers for Supercomputers*. Cambridge, MA: MIT Press.
- Woo, S. C., J. P. Singh, and J. L. Hennessy. 1994. The Performance Advantages of Integrating Block Data Transfer in Cache-Coherent Multiprocessors. *Proc. 6th Int'l Conference on Architectural Support for Programming Languages and Operating Systems* (October):219–229, San Jose, CA,

- Woo, S. C., M. Ohara, E. J. Torrie, J. P. Singh, and A. Gupta. 1995. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. 22nd Annual Int'l Symposium on Computer Architecture* (June):24–36.
- Wood, D. A., S. Chandra, B. Falsafi, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, S. S. Mukherjee, S. Palacharla, and S. K. Reinhardt. 1993. Mechanisms for Cooperative Shared Memory. *Proc. 20th Annual Symposium on Computer Architecture* (May):156–167.
- Wood, D. A., S. J. Eggers, G. Gibson, M. D. Hill, J. M. Pendleton, S. A. Ritchie, G. S. Taylor, R. H. Katz, and D. A. Patterson. 1986. An In-Cache Address Translation Mechanism. *Proc. 13th Annual Symposium on Computer Architecture* (June):358–365.
- Wood, D. A., and M. D. Hill. 1995. Cost-Effective Parallel Computing. *IEEE Computer* 28(2):69–72.
- Woodbury, P., A. Wilson, B. Shein, I. Gertner, P.Y. Chen, J. Bartlett, and Z. Aral. 1989. Shared Memory Multiprocessors: The Right Approach to Parallel Processing. *Proc. COMPCON* (Spring):72–80.
- Wulf, W., R. Levin, and C. Person. 1975. Overview of the Hydra Operating System Development. *Proc. 5th Symposium on Operating Systems Principles* (November):122–131.
- Yamashita, N., T. Kimura, Y. Fujita, Y. Aimoto, T. Manaba, S. Okazaki, K. Nakamura, and M. Yamashina. 1994. A 3.84GIPS Integrated Memory Array Processor LSI with 64 Processing Elements and 2Mb SRAM. *Int'l Solid-State Circuits Conference*, San Francisco (February):260–261.
- Zekauskas, M. J., W. A. Sawdon, and B. N. Bershad. 1994. Software Write Detection for a Distributed Shared Memory. *Proc. Operating Systems Design and Implementation Symposium* (November):87–100.
- Zhang, Z., and J. Torrellas. 1995. Speeding Up Irregular Applications in Shared-Memory Multiprocessors: Memory Binding and Group Prefetching. *Proc. 22nd Annual Symposium on Computer Architecture* (May):188–199.
- Zhou, Y., L. Iftode, and K. Li. 1996. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. *Proc. Operating Systems Design and Implementation Symposium* (October).

Index

A

absolute performance, 202, 228–229
measuring, 203
wall-clock time, 228–229
abstraction layers, 52
for bus-based SMPs, 270 (fig.)
parallel architecture, 27 (fig.)
access
data, 137–142, 254
faults, 558
memory, 310, 942
order with explicit synchronization, 285 (fig.)
order without synchronization, 285 (fig.)
user-level, 491–496
access control, 706
through code instrumentation, 707–708
with decoupled assist, 707
fine-grained, 707
through language/compiler support, 721–724
page-based, 709–721
access time
CRAY T3D, 510
DRAM, 831
reducing, 831
remote cache, 663
acquire-based consistency, 738–739
acquire method, 335
acquire operation, 692
incomplete, in RC, 871
single writer with consistency at, 734–737
Active Messages, 481–482, 490, 515
bulk transfers, 482
echo test, 522
incoming message notification, 482
one-way time, 523
primitives, 481
request/response transactions, 481
active threads, 898
number of, 898

support, 907
See also multithreading; threads
adaptive routing, 790, 799–801
fully adaptive, 800, 801
hot spots and, 817
nonminimal, 801
partially adaptive, 800, 801
See also routing
address home board, 416
address space
global physical, 55
identifiers (ASIDs), 440, 441
independent local physical, 55
private, 112
reflective memory, 519 (fig.)
See also shared address space;
shared physical address space
admission control, 818
Alewife project, 70, 915
algorithmic optimization, 219
algorithmic speedup, 220
measuring, 254
for parallel applications, 256 (fig.)
See also speedup
algorithms
Barnes-Hut, 79
barrier, 356–357, 542–547
Dekker's, 688
Gaussian elimination, 194
lock, 337, 342–343, 346–348,
538–541
routing, 752–753
scheduling, 808
sequential, 161, 166–169, 175,
179–180, 389
software queuing lock, 541 (fig.)
synchronization, performance,
649–651
waiting, 335
west-first, 797–798, 798–799
Alliant FX-8 machine, 435
all-to-all personalized communication, 547
example, 547
savings, 588
AMBER (Assisted Model Building through Energy Refinement), 8
Amdahl's Law, 84, 841
application of, 88
rewriting, 87
application miss rate, 319–324
breakdown for Barnes-Hut, LU,
Radiosity, 320 (fig.)
breakdown for Multiprog, 323
(fig.)
breakdown for Ocean, Radix,
Raytrace, 321 (fig.)
breakdown for 64-KB caches, 325
(fig.)
reducing, 325
See also cache miss rate
applications
performance, 429–433
performance improvement with
parallelism, 6
scaling, 431–433
speedups, 430–431
See also specific applications
application trends, 6–12
commercial, 9–12
scientific/engineering, 6–9
architectural trends, 14–21
microprocessor design, 15–19
system design, 19–21
array-based locks, 347–348
acquire method, 347
drawback, 348
latency, 347
performance, 350, 651
scalable problems, 539
See also locks
arrays
aligning, 365–366
alternative organizations, 365
(fig.)
4D vs. 2D performance impact,
363 (fig.)
organization determination,
364–365

arrays (*continued*)
 padding, 364
 smaller than 32 KB, 424
 artifactual communication, 137,
 139–142, 653
 avoiding, 257
 in extended memory hierarchy,
 139–140
 problem size and, 223
 Raytracing application, 176
 reducing, 142–150
 replication and, 140–142
 shared address space, 142, 146
 sources, 139–140
 ASCII Red machine, 502, 503, 771
 assignment, 83, 88–89
 Barnes-Hut application, 169–170
 block, 98
 changes, 107–108
 compile-time, 88
 cyclic, 98, 108
 Data Mining application, 83,
 180–181
 dynamic, 88, 126–129
 equation solver kernel, 98–99
 load-balanced, 127
 Ocean application, 161–163
 performance goals, 88
 Raytrace application, 175–176
 static, 88, 99, 126–129
See also parallelization process
assist occupancy
 contention impact on, 647
 effects on protocol trade-offs, 648
 impact on cache coherence
 protocols, 647 (fig.)
 latency tolerance and, 845
asynchronous links, 765
asynchronous message passing
 protocol, 479 (fig.)
 storage, 479
See also message passing
ATM (asynchronous transfer mode),
 646
 flow control, 813
 standard, 514
 switches, 514
atomic bus, 281
 lock-down, 391
 single-level caches with, 380–393
atomic primitive implementation,
 391–393, 651–652
atomic read-modify-write operation,
 334
attraction memories, 701
automatic update mechanism, 718,
 719 (fig.)
availability clusters, 514

B

backoff
 dynamic, 595
 exponential, 649, 651
 between failed operations, 393
 instructions, 920
 test&set locks with, 342
 values, 920
back pressure, 483, 816
back-to-back latency, 619
backward pointer, 569, 624
bandwidth, 59
 aggregate, 761, 762
 bisection, 761
 block data transfer and, 241, 242,
 857
 broadcast, 459
 bus-based organization and, 34
 channel, 752
 communication processor (CP),
 502
 across computer system, 951
 (fig.)
 data transfer operation, 60
 effective, 756
 HAL SI, 644
 hierarchical directory scheme,
 566
 individual, 761
 latency vs., 59, 787
 link, 760, 939
 link, limitations, 668
 matching, 807
 message-passing, 529
 message size vs., 531 (fig.)
 microprocessor bus, 19, 21
 network, 761–764, 846
 node-to-network, 846
 NUMA-Q, 641
 number of ports and, 462
 offered, 762
 on rings, 443, 444
 out-of-cache memory, 939
 point-to-point, 151, 845–846
 per-processor requirements, 312
 (fig.)
 protocol design and, 307–311
 scaling, 445–446, 457–459
 SGI Origin2000, peak hardware,
 618
 switch, 939
 total, 762
 uniprocessor design, 939
 vector memory transfer, 46
Banyan network, 803–804
Barnes-Hut algorithm, 79
Barnes-Hut application, 76–77,
 78–79, 166–174
assignment, 169–170
Barnes-Hut algorithm and, 79
barrier, 106
cache-to-cache sharing, 588
computation flow (fig.), 168
costzones, 170, 171 (fig.)
decomposition, 169–170
error and, 265
execution time, 174 (fig.), 214
force calculation, 166, 169
invalidation pattern, 575 (fig.)
invalidations, 577
latency in, 913
mapping, 173
miss rates, 320 (fig.)
naming, 184
n-body problem, 79, 80 (fig.)
ORB, 170, 171 (fig.)
orchestration, 170–173
particles, number of, 265
partitioning, 169, 171 (fig.)
scaling, 432
scaling of speedups on SGI
 Origin2000, 622 (fig.)
sequential algorithm, 166–169
spatial locality, 170–172, 322
speedup, 431
speedup on SGI Origin2000, 621
 (fig.)
star force computation, 79
summary, 173–174
synchronization, 172–173
tasks, 82
temporal locality, 172
three-dimensional space calculation, 167
traffic vs. local cache, 582 (fig.)
traffic vs. number of processors,
 580 (fig.)
tree traversals in, 893
two-dimensional particle distribution, 168 (fig.)
working sets, 172
See also case studies
barrier algorithms, 356–357,
 542–547
all-to-all personalized communication, 547
global synchronization, 356–357
parallel prefix, 546–547
scalable multiprocessor synchronization, 542–547
software combining trees,
 542–543
tree barriers with local spinning,
 543–545
barriers
 centralized, 353–356
 fairness, 356

- hardware, 358
 latency, 356
 memory (MB), 693
 performance goals, 356
 performance on SGI Challenge,
 357 (fig.)
 scalability, 356
 static binary tree, 544
 storage cost, 356
 traffic, 356
 write memory (WMB), 693
- barrier synchronization, 252,
 353–358
 barrier algorithms, 356–357
 centralized barrier with sense
 removal, 354–356
 centralized software barrier,
 353–354
 hardware barrier, 358
 hardware primitives, 357–358
 performance, 356
See also synchronization
 baseline communication structure,
 834
 BBN Butterfly machine, 760
 benchmarks
 BT, 532–537
 kernel, 967
 LAPACK, 963
 LINPACK, 13, 209
 low-level, 967
 LU, 532, 533 (fig.), 537–538
 microbenchmarks, 215–216, 967
 NAS, 966–967
 ongoing efforts, 968
 PARKBENCH, 967–968
 Perfect Club, 968
 Scalapack, 963
 SPEC, 13, 199
 SPLASH, 307, 965–966
 TPC, 642, 643, 963–965
 workloads, 201
 Benes network, 776, 777 (fig.)
 binary trees, 772
 illustrated, 773 (fig.)
 space partitioning (BSP), 250
 binding prefetch, 880
 bisection bandwidth, 761
 bisection scaling rule, 788
 bit-level parallelism, 15
 bit-serial design, 46
 Blizzard-S, 745
 block data transfer, 187–188, 837
 advantages, 188, 838, 857
 amortized per-message overhead,
 857
 bandwidth and, 241, 242
 bandwidth waste, 857
 communication assist, 854
 contention, 858
- disadvantages, 858
 effect, 839 (fig.)
 explicit, 853
 facility, 238
 in FFT program, 861
 interaction with cache-coherent
 shared address space,
 855–856
 mechanisms, 853–854
 message passing, 848
 in near-neighbor equation solver,
 859 (fig.)
 in Ocean application, 861 (fig.)
 overhead, 242
 overhead per transfer, 858
 path, 854 (fig.)
 performance benefits, 856–863
 pipelined transfer, 857
 pipeline stage time, 862
 policy issues, 854–856
 read-write cache-coherent
 communication vs., 862–863
 relative performance improve-
 ment with, 860 (fig.)
 at row-oriented partition
 boundaries, 860
 shared address space, 853–863
 synchronization bundling, 857
 system buffering/copying,
 853–854
 techniques, 853–854
 trade-offs, 854–856, 859
 transferred data placement and,
 856
 transferred data replication, 857
See also latency tolerance
 blocked multithreading, 898–902
 context switch, 916–917
 control, 916–917
 disadvantages, 912
 EPC, 915–916
 implementation issues, 914–917
 interleaved scheme vs., 911
 latency tolerance, 903 (fig.)
 PC bus, 919 (fig.)
 processor utilization vs. number
 of threads, 899 (fig.)
 requirements, 914
 speedup, 912 (fig.)
 state replication, 914–915
See also multithreading
 blocking, 144, 195
 asynchronous SEND operation,
 115
 benefits, 245
 busy-waiting trade-offs, 335
 caches, 414, 877
 data reuse and, 246
 to exploit temporal locality, 144
 (fig.)
- head-of-line, 805, 806
 reads, 864, 877
 block prefetch, 880
 blocks
 B-by-B, 246
 busy state, 590
 contiguous data, 247
 directory information, 562
 directory state, 558
 directory tree, 566
 head pointer, 569
 interleaving, 246
 local, 560
 remote, 560
 size of, 246
 block transfer engine (BLT), 819
 board-level integration, 465–466
 bounded buffer problem, 117
 branch target buffer (BTB), 918
 broadcast
 bandwidth, 459
 bit, 655
 media, hierarchy of, 555
 scheme, 655
 snooping mechanism, 559
 version, 119
 BSP model, 191
 BT benchmark, 532–537
 application performance, 534
 (fig.)
 communication characteristics,
 535 (fig.)
 message profile over time, 536
 (fig.)
 speedups, 533
 temporal communication
 behavior, 536
See also benchmarks
 buffer deadlock, 412
 causes, 412
 solutions, 412, 594
See also deadlock
 buffering
 at home, 590–591
 input, 804–806
 memory operation, 863
 output, 806
 at requestors, 591
 switch, 759, 768
 virtual channel, 807–808
 buffers
 branch target (BTB), 918
 channel, 804–808
 FIFO, 409, 616, 804
 flit, 816
 input, 472, 482–483
 intervention request (IRBs), 615
 lookahead, 870
 output, 472
 prefetch, 878

buffers (*continued*)
 read request (RRBs), 615
 reorder, 414, 870, 876
 stream, 882
 TLB, 67, 223
 write, 413, 865, 867, 874
 write-back, 385
 write request (WRBs), 615

bus-based systems
 hierarchical, 677
 organization, 32–34
 scaling data in, 445
 snoop bandwidth in, 445–446

buses, 555
 PCI, 635, 636, 637
 sharing, 588, 589
 snooping, 277–283, 589
 split-transaction, 398–415
 SysAD, 609, 612, 613

bus order, 281–282
 program order and, 282
 writes/reads serialized in, 291

bus traffic
 block size impact on, 324–327
 cache block size for Multiprog, 327 (fig.)
 cache block size with 1-MB caches, 326 (fig.)
 cache block size with 64-KB caches, 328
 test-and-test&set locks and, 343

bus transactions
 arbitration medium, 470
 completion of, 470
 destination, 470
 information format, 470
 input registers, 470
 ordering properties, 473
 output registers, 470

bus upgrades, 295

busy-overhead, 158

busy states, 590
 blocks in, 590
 SGI Origin2000, 597, 600

busy time, 897

busy-useful, 157, 210

busy-waiting, 106–107, 335–336
 cost, 336
 trade-offs, 335

butterflies, 774–777
 back-to-back, 776
 d-dimensional indirect, 774
 d-dimensional k-ary, 775
 fault tolerance, 775
 forward-going, 777
 illustrated, 775 (fig.)
 links, 775–776
 routing, 778
 2 x 2, 774
 with wormhole routing, 808

See also network topologies

C

cache associativity, 241

cache-based directory scheme, 566, 568–570
 latency reduction, 587 (fig.)
 NUMA-Q, 622–645
 See also directory protocols; directory schemes

cache blocks
 address tag, 329
 dirty, 406, 409
 exclusive copy of, 292
 false sharing of, 711
 granularity, 706
 invalid, writing into, 294, 296
 large, alleviating drawbacks of, 328–329
 lifetime of, 316
 modified state, 292
 multiword, 319
 old, invalidating, 389–390
 overhead, 242
 owner, 292
 promoted, 294
 replacement, 296
 shared, writing into, 295, 296
 sharing state, 402
 states, 279
 state transition diagram, 279–280
 state transitions, 279, 367
 write-back protocols and, 296

cache block size, 241
 bus traffic and, 324–327
 effects on proceeding past writes, 869 (fig.)
 miss rate and, 318–319
 trade-offs, 313–329

cache coherence, 272, 273–283
 with bus snooping, 277–283
 directory-based, 553–677
 example problem, 274 (fig.)
 extending, 433–446
 as hardware design issue, 275
 hardware support for, 669
 home memory and, 590
 I/O buses and, 515
 memory bus, 508
 NUMA-Q, 624–632
 problem, 273–277
 processor overhead and, 645
 protocol, 305
 scalable, 558–559
 SGI Origin2000, 597–604
 snoop-based, on rings, 441–444
 snooping, 272

cache-coherent multiprocessors
 block data transfer vs., 862–863
 bus-based, 441
 coherence misses, 314–315
 hardwired assist, 706

latency tolerance, 926–927
 snoop-based, 589

cache-coherent, nonuniform memory access (CC-NUMA)
 architectures, 558, 725
 flexibility, 730–732
 hardware cost, 705–706
 main memory management, 653
 performance trade-offs, 703 (fig.)
 physical address space limitations, 732
 two-processor system, 674

cache-coherent shared address space, 879–891
 hardware-controlled prefetching, 881–883
 hardware-controlled vs. software-controlled prefetching, 888–890
 interactions with multiprocessor coherence protocol, 887–888
 prefetching concepts, 880–881
 prefetching with single processor, 884–887
 sender-initiated precommunication, 890–891
 software-controlled prefetching, 884
 See also shared address space

cache conflicts, 224

cache controllers, 62, 389
 in bus-based system, 652
 design, 381–382
 hardwired assist, 706
 implementation requirements, 663
 lowest-level, 397
 request table, 402
 snooping, 277
 uniprocessor, 381–382

cache misses, 17, 140–141
 application structure and, 319–324
 capacity, 141, 223, 313
 classification of, 316–318
 coherence, 314–315
 cold-start, 140
 compulsory, 313
 conflict, 141, 164, 314, 362–363
 “context available” signal and, 920
 cost of, 324
 data transfer, 58
 decomposition of, 318
 detection mechanism, 558
 false-sharing, 315–316
 latency of, 17, 324
 path of, 404–406
 process migration and, 324
 read, 662, 744

- reducing, 141
 time, 943, 944
 TLB, 223, 429
 true-sharing, 315, 316
 upgrades, 318
 write, 666
 cache miss rate, 943
 Barnes-Hut application, 320 (fig.)
 block size and, 318–319
 hybrid protocols, 332 (fig.)
 invalidation-based protocols, 332 (fig.)
 LU application, 320 (fig.)
 Multiprog application, 323 (fig.)
 Ocean application, 321 (fig.)
 Radiosity application, 320 (fig.)
 Radix application, 321 (fig.)
 Raytrace application, 321 (fig.)
 update-based protocols, 332 (fig.)
 cache-only memory architecture
 (COMA), 680, 701–705
 access latency, 702
 attraction memories, 701
 communication latencies, 729
 design options, 703–705
 explicit migration, 730
 fine-grained replication, 729
 flat directory scheme, 703–704
 hardware-intensity, 726
 hardware/software trade-offs,
 701–702
 hardware support, 702, 739
 hierarchical directory scheme,
 703–705
 high-capacity miss rates and, 702
 memory blocks, 701
 performance trade-offs, 702–703
 read operation path, 705
 Simple, 681, 726–728
 workloads, 729–730
 caches, 17, 46
 allocation granularity, 277
 blocking, 414, 877
 block size, 226
 commercial use of, 67
 CRAY T3E, 36
 direct-mapped, 241, 247, 362 (fig.)
 filter bandwidth, 68
 hardware, 185
 infinite, 572
 L_1 , 394–395, 396
 local, 139
 lockup-free, 414, 922–926, 930
 LRU, 259
 mapping conflicts, 362 (fig.)
 microprocessor transistors
 devoted to, 948 (fig.)
 multilevel, 393–398
 organization of, 313
 per-processor, 272
 remote, 623–624, 660, 662
 replacement, 185
 scaling, 236–237
 shared, 434–437
 single-level, 281, 380–393
 snooping, 383, 386 (fig.), 398 (fig.)
 software, fixed-size, 186
 SRAM, 304
 tertiary, 700–701
 uniprocessor, 381
 virtually indexed, 437–439
 write-back, 274, 283, 291
 write-no-allocate, 281 (fig.)
 write-through, 278, 279, 281 (fig.), 291
 See also cache controllers; cache misses; cache miss rate
 cache sizes, 142 (fig.), 236
 choosing, 239–240
 miss rate vs., 224 (fig.)
 for scaled-down problem, 237 (fig.)
 cache tags
 design of, 381–382
 dual, 397
 looking up, 401
 nonmatching, 396
 cache-to-cache handshake, 383
 cache-to-cache sharing, 300, 589
 Barnes-Hut application, 589
 likelihood of, 597
 capacity limitations, 700–705
 COMA, 701–705
 tertiary cache, 700–701
 capacity misses, 141, 223, 314
 block size and, 318
 COMA machines and, 702
 occurrence of, 313
 Ocean application, 324
 Raytrace application, 324
 reducing, 314
 spatial locality in, 324
 See also cache misses
 case studies
 Barnes-Hut application, 76–77, 78–79
 CM-5, 493–494
 CRAY T3D, 508–511, 818
 CRAY T3E, 512–513
 Data Mining application, 77, 80–81
 IBM SP-1/SP-2, 820–822
 Intel Paragon, 499–503
 LU, 244–248
 Meiko CS-2, 503–506
 Multiprog, 244, 252
 Myricom network, 826–827
 Myrinet SBUS Lanai, 516–518
 nCUBE/2, 488–490
 network design, 818–827
 NUMA-Q, 622–645
 Ocean application, 76, 77–78
 parallel application, 76–81, 160–182
 PCI Memory Channel, 518–521
 Radiosity application, 244, 249–252
 Radix application, 244, 248–249, 267
 Raytrace application, 77, 79–80
 SCI, 822–825
 SGI Challenge, 415, 417–424
 SGI Origin2000, 596–622, 825–826
 Sun Enterprise 6000, 415–416, 424–429
 workload, 244–253
 CDC
 CDC 7600, 67
 STAR-100, 67, 953
 centralized barriers, 353–356
 with sense removal, 354–356
 sense reversal method, 355
 shared counter, 353
 See also barriers
 centralized directory schemes, 564
 centralized queue, 128
 channel buffers, 804–808
 input, 804–806
 output, 806
 shared pool, 807
 virtual channel, 807–808
 See also buffers
 channels, 751, 752
 bandwidth, 752
 dependence graph, 793, 794 (fig.)
 occupancy, 755
 ordering, 794 (fig.)
 packet occupation, 756
 resource allocation, 792
 as shared resource, 791
 time, 460
 virtual, 613, 795–796
 width, 784, 787
 chip-level integration, 463–464
 circuit switching, 756–757
 CISC microprocessors, 19
 clocking
 asynchronous, 765
 synchronous, 765
 closed systems, 760
 clusters, 12
 availability, 514
 hardware primitives, 515
 as parallel machines, 514
 potential of, 514
 CM*, 68
 CM-5, 70, 465–466, 953

- C.mmp, 68
 CM-1, 69, 952
 CM-2, 69, 952
 case study, 493–494
 communication assist, 493
 communication latency, 525
 “control” network, 542
 data networks, 494
 latency, 493
 machine organization, 466 (fig.)
 message interleaving, 495
 network, 465
 receive overhead, 525
 send overhead, 523–524
 shared address read, 527
 user-level network support, 493–494
 coarse-grained tasks, 83
 coarse vector overflow method, 656, 657 (fig.)
 coarse vector representation, 609, 656 (fig.)
 code instrumentation, 707–708
 protocol cost, 708
 run-time cost, 707
 coherence, 272
 cache, 272, 273–283, 433–446, 553–677
 compiler-based, 723
 global, 855–856
 granularity of, 146, 724, 725 (fig.)
 hierarchical, 659–669
 local, 855
 management, 724
 memory system, 276
 MSI protocol and, 297
 multilevel cache hierarchies and, 393
 object-based, 721–723
 properties, 275, 277
 serialization to location for, 589–591, 604–607, 632–633
 software SVM, 721
 TLB, 439–441
 for virtually indexed caches, 437–439
 coherence controllers, 562
 contention at, 654
 NUMA-Q, 731
 Stanford FLASH, 731
 coherence misses, 314–315
 false-sharing, 315
 improving locality on, 328
 occurrence of, 314–315
 true-sharing, 315
See also cache misses
 cold misses, 140, 313
 block size and, 318
 occurrence of, 313
 reducing, 314
 spatial locality in, 324
See also cache misses
 commercial computing, 9–12
 communication, 25
 all-to-all personalized, 547
 architecture implications and, 135
 artificial, 137, 139–142, 653
 available, operations, 26–27
 baseline, structure, 834
 collective, 55
 with communication, 837
 computation with, 837
 cost, 62–63, 122, 151, 156
 endpoints, 518
 expense of, 138
 explicit, 116, 187, 189
 framework for understanding, 26
 frequency of, 62
 granularity, 186–187, 724, 725 (fig.)
 hardware support for, 515
 impact of, 132
 implicit, 189
 inherent, 140
 interprocess, 58, 131
 Meiko CS-2, 505 (fig.)
 message-passing, 38, 111
 microbenchmarks, 216
 misses, 141
 in multimemory system, 137–142
 network as pipeline for, 836 (fig.)
 overhead, 186–187
 overhead, reducing, 487
 overlapping, 63, 155–156
 performance, 755–764
 pipeline, 836
 between processes, 59
 processes timelines, 835 (fig.)
 queues, 498
 read-write, 852
 receiver-initiated, 833
 reducing, 123, 131–135
 redundant, data, 140
 replication and, 58–59
 sender-initiated, 833, 879
 shared address space, 473, 474 (fig.)
 structure, 137, 848, 852
 structuring, to reduce cost, 150–156
 SVM, 712 (fig.)
 true, 58
 communication abstraction, 26, 52, 56
 “contract,” 53
 implementing, 488
 interface, 53
 in large-scale machines, 469
 message passing, 39 (fig.), 488
 role of, 53
 communication architecture, 25–28
 design space, 485–486
 facets, 25
 communication assists, 50–51, 156
 blind physical DMA, 487 (fig.)
 block data transfer, 854
 CM-5, 493
 CRAY T3D, 511
 design, 51
 HAL S1 multiprocessor, 644
 Memory Channel, 518
 Myricom network, 826
 network transaction formatting, 485
 in protocol processing, 645
 shared memory design and, 51
 shared physical address space, 506, 507
 for user-level handlers, 495 (fig.)
 for user-level network ports, 492 (fig.)
 communication latency, 522–523, 833
 comparison, 525–526
 components, hiding, 842
 message breakdown, 523 (fig.)
 speedup, 842
See also latency; latency tolerance
 communication processor (CP), 455, 496
 bandwidth, 502
 computer processor and, 498
 concurrency intensive function, 499
 cooperation, 496
 dedicated, 497
 Meiko CS-2, 503
 message delivery, 499
 polling, 497
 synchronization operations, 496
 communication-to-computation ratio, 132
 computing, 132
 control of, 132
 data set size and, 258
 growth rates of, 258–259
 MC scaling, 433
 measuring, 257
 n/p ratio and, 226
 processor count vs., 258
 scaling models, 212
 TC scaling, 433
 in three dimensions, 132
 in two dimensions, 132
 compare&swap instruction, 334, 340, 649
 atomic, 540
 implementing, 392
 queuing lock, 540
 competing operations, 695

- compiler-based coherence, 723
 compilers, 959
 - advanced optimizations, 289–290
 - architecture evolution and, 75
 - parallelizing, 961
 - technology, 959
 - uniprocessor, 289
 complete applications, 217–218
 compulsory traffic, 140
 Computational RAM, 951
 computer architecture
 - change and, 4
 - generations, 15
 - history of, 945 (fig.)
 computer systems
 - bandwidths across, 951 (fig.)
 - 500 fastest, 24 (fig.)
 concurrency
 - degree of, 98
 - exposing, 85
 - extra, 155
 - finding with program structure, 93–94
 - Gaussian elimination, 194–195
 - in Gauss-Seidel equation solver computation, 95 (fig.)
 - identifying, 124–126
 - limited, 84, 86
 - loop nests and, 93
 - managing, 126–129
 - reducing, 99, 101
 - scaling models, 212
 - workload, 254–257
 concurrency profiles, 85
 - area under curve, 86
 - distributed time, discrete-event logic simulator, 87
 - x-axis points, 87
 conflicting operations, 695
 conflict misses, 141, 314, 876
 - across grids, 164
 - increase of, 319
 - occurrence of, 314
 - predicting, 885
 - reducing, 314
 - See also* cache misses
 conflict order, 696
 constraints
 - resource-oriented, 207
 - user-oriented, 207
 contention
 - assist occupancy and, 647
 - block data transfer and, 858
 - cause of, 154, 654
 - at coherence controllers, 654
 - endpoint, 154
 - at endpoints, 761
 - latency and, 759, 786, 787 (fig.)
 - lock algorithms and, 675
 - network, 154
 orchestration strategies and, 654
 - prefetching and, 895
 - preserving, 235
 - problem, 153–154
 - reducing, 153–154
 - resource, 62
 context status word (CSW), 915
 context switches, 897, 901
 - in blocked scheme, 916–917
 - overhead, 901
 - See also* switches
 Convex Exemplar, 570, 700
 - crossbar, 940
 - SCI specification, 822
 correctness, 377
 - classes, 589
 - deadlock, 594–595
 - livelock, 595
 - NUMA-Q, 632–634
 - requirements, 378–380
 - serialization across locations for sequential consistency, 592–593
 - serialization to location of coherence, 589–591
 - SGI Origin2000, 604–609
 - starvation, 595–596
 cost, 59
 - amortizing, 588
 - busy-waiting, 336
 - cache miss, 324
 - communication, 62–63, 122, 151, 156
 - data access, 138
 - delay, 152
 - design, 188, 679
 - fixed, 461
 - hardware, 188, 705–724
 - implementation, 679
 - message, 151
 - network, 782
 - parallelism, 537
 - performance trade-off, 2, 4, 15
 - pipelining, 900
 - reducing over integrated/specialized assist, 708
 - reducing through communication structure, 150–156
 - scaling, 461–462
 - start-up, 60
 - switch, 768, 898, 900
 - unready thread, 909–910
 costup, 462
 costzones, 170, 171 (fig.)
 CRAY
 - C90 vector processor, 8
 - CRAY-1, 45, 67, 953
 - CS6400, 410, 445
 - SCX I/O network, 822
 - 3/SSS, 952
 vector supercomputers, 8, 22
 Xmp, 337
 CRAY T3D, 8, 70, 186, 508–511
 - access time, 510
 - Alpha architecture, 509
 - Alpha 21064, 509–510, 511
 - annex registers, 510
 - BT benchmark, 533
 - bulk transfer engine, 511
 - communication assist, 511
 - design, 508–509
 - DTB Annex, 510
 - flow control, 820, 942
 - hardware support for barriers, 542
 - links, 819, 820
 - LU benchmark, 532
 - machine organization, 509 (fig.)
 - network, case study, 818–820
 - nonblocking stores, 511
 - nonblocking writes, 513
 - packet buffers, 820
 - packet formats, 819 (fig.)
 - prefetch buffer, 878
 - prefetch instruction, 511
 - routing distance, 820
 - scaling, 509
 - shared address read, 528
 - synchronous link design, 765
 - three-dimensional bidirectional torus, 818
 - user-level messaging support, 526
 - virtual-to-physical translation, 510
 - write buffer, 511
 CRAY T3E, 36 (fig.), 186, 512–513
 - Alpha 21164 processor, 512
 - asynchronous link design, 766
 - block transfer engine utility, 512
 - design, 512
 - E-registers, 512
 - improvements, 512–513
 - nonblocking writes, 513
 - prefetch buffer, 878
 - prefetch queue utility, 512
 critical section, 105
 crossbar, 803–804
 - Convex Exemplar, 940
 - implementations, 803 (fig.)
 - memory as, 802
 - size increase, 808
 - tristate drivers and, 809
 crossbar switch, 30, 31 (fig.), 34
 cube-connected cycles, 811
 cut-through routing, 757 (fig.), 782
 - deadlock, 792
 - distance, 779
 - See also* routing
 Cyber 835 mainframe, 924–925
 cycles per instruction (CPI), 138

cyclic assignment, 98, 108
 cyclic redundancy checks (CRCs),
 608, 633
 SP networks and, 822
 trailer word, 766

D

dancehall multiprocessor organization, 459 (fig.)
 data access
 cost, 138
 in multimemory system, 137–142
 workload, 254
 dataflow architecture, 47–49
 division, 48
 dynamic scheduling, 51
 execution pipeline, 48 (fig.)
 graph, 48 (fig.)
 operation naming, 48–49
 tagged-token, 48
 tags, 47–48
 tokens, 47
 See also parallel architectures
 Data General, 570
 data-local, 158
 data mining, 80–81
 for associations, 80–81
 data queries vs., 80
 Data Mining application, 77, 80–81,
 178–182
 assignment, 83, 180–181
 barrier, 106
 decomposition, 180–181
 disk access, 182
 equivalence classes, 180, 181, 182
 itemsets, 180, 278–279
 lexicographic sorting, 182
 load balance, 180–181
 mapping, 182
 orchestration, 181–182
 sequential algorithm, 179–180
 spatial locality, 182
 summary, 182
 synchronization, 182
 tasks, 82–83
 temporal locality, 182
 data parallelism, 124, 125
 data parallel processing, 26, 44–47
 equation solver pseudocode, 100
 (fig.)
 languages, 47
 multiple cooperating microprocessors, 46
 orchestration under, 99–101
 “virtual processor” operations, 49
 data-race-free models, 695
 data races, 696
 data-remote, 158
 data reuse, 246
 data set size, 206

data structuring optimization, 219
 data traffic components, 360
 data transfers, 58
 bandwidth, 60
 granularity, 145
 occurrence of, 58, 59
 time for, 60–61
 within machines, 59
 See also block data transfer
 deadlock, 378–379, 791
 avoidance assumption, breaking,
 594
 buffer, 412, 594
 in computer system, 379 (fig.)
 cut-through routing, 792
 DASH system and, 594
 detection methods, 594
 examples, 792 (fig.)
 fetch, 390, 412, 483–485
 freedom, 791–795
 “head on,” 791
 NACK solutions, 596
 network transactions and, 473
 NUMA-Q, 633
 potential, 378, 390
 prevention, 379, 412
 SGI Origin2000, 595, 608
 store-and-forward routing, 792
 at traffic intersection, 379 (fig.)
 See also livelock; starvation
 deadlock-free networks, 483
 DEC Alpha, 12, 693, 699
 DEC Memory Channel, 520
 hardware organization, 521 (fig.)
 transmit regions, 520
 decomposition, 83, 84–88
 Barnes-Hut application, 169–170
 of cache misses, 318
 changes to, 107–108
 Data Mining application,
 180–181
 domain, 132, 134 (fig.)
 equation solver kernel, 93–98
 goal of, 84
 Ocean application, 161–163
 Raytrace application, 175–176
 row-based, 98
 scatter, 176, 246
 See also parallelization process
 DECOMP statement, 100–101
 decoupled hardware approach, 707
 dedicated message processing,
 496–506
 machine organization with em-
 bedded processor, 498 (fig.)
 machine organization with sym-
 metric processor, 497 (fig.)
 Dekker’s algorithm, 284, 688
 delay
 cost, 152
 network, 61, 62, 646–647
 network transit, 152
 propagation, 815
 reducing, 152–153
 routing, 460, 758, 761, 781 (fig.)
 switching, 756
 delayed-exclusive replies, 698
 delta network, 804
 dependence order, 54
 dependences, 124
 data, 94
 in Gauss-Seidel equation solver
 computation, 95 (fig.)
 Ocean application, among grid
 computations, 162
 deterministic routing, 790–791
 algorithms, 791
 multiple routes, 791
 routing path conflicts, 801 (fig.)
 See also routing
 diffs, 717–718, 737
 application, 718
 computation, 718
 correct order of, 738
 ERC, 745
 LRC, 717, 746
 processing, 718
 storage, reducing, 739
 digits, 248
 dimension order routing, 789, 800
 directed acyclic graph (DAG), 797
 direct-mapped caches, 241, 247, 362
 (fig.)
 direct memory access (DMA), 455
 controllers, 490, 495
 descriptor, 493
 devices, 486
 engine, 486, 499–502, 517
 full-cache-block write, 422
 HIO interface chips and, 422
 IBM SP-2 engine, 41
 input channels, 488, 489
 I/O transfers by, 274
 nonblocking sends and, 40
 output channels, 488
 partial-block write, 422, 423
 performance advantages, 493
 physical, 486–491
 read requests, 422
 read responses, 422
 SGI Origin2000 support, 610
 transfers, 40, 43, 481, 487, 491,
 493
 direct networks, 489
 directories
 basic operation of, 561 (fig.)
 block information, 562
 distributed, doubly linked list,
 569 (fig.)
 height, 568

- height reduction, 659
 hierarchical, 660, 664–667
 limited pointer, 568
 on read miss, 560
 organization of, 565–571
 sparse, 659
 state diagrams, 670 (fig.), 671 (fig.)
 storage overhead, reducing, 655–659
 value of, 564
 width, 568
 width reduction, 655–658
- directory-based cache coherence, 553–677
 approaches, 559–571
 complexity, 669
 hierarchical coherence, 659–669
 parallel software implications, 652–655
 performance parameters, 645–648
 storage overhead reduction, 655–659
 synchronization, 648–652
 two-level organizations, 557 (fig.)
See also cache coherence
- directory-based multiprocessors, 553–677
- directory controllers, 562, 639
- directory lookup, 586, 599
- directory protocols, 556
 assessing, 571–579
 assist occupancy impact, 646–647
 cache block size effects, 579
 correctness, 589–596
 definitions, 560
 design challenges, 579–596
 directory state, 558
 dirty node, 560, 562
 exclusive node, 560
 home node, 560
 local node, 560
 local vs. remote traffic, 578–579
 machine organization and, 587–589
 network delay impact, 646–647
 NUMA-Q, 624
 operation, 560–564
 optimizations, 585–587
 outer protocol as, 556
 overview, 559–571
 owner node, 560
 performance, 584–589, 645–648
 scaling, 564–565
 SGI Origin, 588
- directory schemes
 alternatives, 567 (fig.)
 cache-based, 566, 568–570
 centralized, 564
- data sharing patterns, 571–577
 flat, 565, 567–571
 hierarchical, 565–567
 memory-based, 566, 567–568
 operation, 560–564
 summary, 571
- directory states
 busy, 603–604
 exclusive, 603
- directory trees, 566
- dirty bit, 560, 563
- dirty node, 560, 562, 591
- distributed, doubly linked list, 569
- distributed-memory organization
 consistency model, 508
 generic, 458 (fig.)
- distributed queues, 128
- distributed shared memory (DSM) systems, 558
- domain decomposition, 132
 load balanced, 133–134
 for simple nearest-neighbor computation, 134
See also decomposition
- downgrade, 600
- Dragon write-back update protocol, 301–305, 374
 bus transactions, 302
 exclusive-clean (E) state, 302
 lower-level design choices, 304–305
 modified (M) state, 302
 for processor actions, 305 (fig.)
 read miss, 303
 replacement, 304
 shared-clean (Sc) state, 302
 shared-modified (Sm) state, 302
 SRAM caches, 304
 state transition diagram, 303 (fig.)
 state transitions, 303–304
 write, 303–304
See also update-based protocols
- DRAM chips, 14
 access time, 831
 bit array, 950
 buffer caches, 954
 capacity, 14, 949, 950
 cycle times, 938
 designs, 949
 engineering requirements for, 949
 enhanced, 949
 gigabit, 946
 integrating logic into, 949
 interfaces, 950
 packages, 949
 remote cache, 663, 700
 vector organization, 953 (fig.)
- D-tags, 427, 429
- dual-ported RAM, 382
- dynamic assignment, 88
- load balancing and, 128
 static assignment vs., 126–129
- dynamic backoff, 595
- dynamic partitioning, 126, 127 (fig.)
- dynamic scheduling, 870
 latency and, 938
 processors, 895, 922
 Radix, 875 (fig.)
- dynamic tasking, 126–128
 advantages, 127–128
 example, 128
 task pool, 127, 129 (fig.)
 techniques, 128–129
- Dyna3D, 956–957
- E**
- eager-exclusive replies, 698–699
- eager release consistency (ERC), 712
 diffs, 745
 lazy vs., 713
 non-null pointer value, 715–716
 software, 715
- echo, 824
 negative, 824
 positive, 824
 test, 522
- e-cube routing, 778, 790
- effective bandwidth, 756
- efficiency-constrained scaling, 231
- EM-4, 494, 495
- encapsulation, 754
- Encore Gigamax
 block diagram, 664 (fig.)
 system configuration, 663
See also hierarchical snooping
- Encore Multimax, 435
- endpoint contention, 154
- end-to-end flow control, 494, 763, 813, 816–818
 admission control, 818
 global communication operations, 817–818
 hot spots, 817
See also flow control
- entry consistency model, 722
- equation solver kernel, 92–116
 accumulation and convergence determination in, 114 (fig.)
 assignment, 98–99
 data parallel, pseudocode, 100 (fig.)
- decomposition, 93–98
 with decomposition into grid points, 98 (fig.)
- finite differencing method, 92
- ghost rows, 109, 111
- nearest-neighbor update, 93 (fig.)
- orchestration under data parallel model, 99–101

equation solver kernel (*continued*)
 orchestration under message-passing model, 108–116
 orchestration under shared address space model, 101–108
 picking cache sizes with, 240 (fig.)
 pseudocode, 94 (fig.)
 red-black ordering, 97 (fig.)
 row-based, cyclic assignment of, 108 (fig.)
 scaling model impact on, 211–213
 shared address space model
 pseudocode, 104–105 (fig.)
 spatial locality in, 149 (fig.)
 temporal locality in, 143–144
 error correction codes (ECCs), 608, 633
 error handling
 NUMA-Q, 633–634
 SGI Origin2000, 608–609
 Ethernets
 flow control, 812
 switched gigabit, 548, 940
 event-driven simulation, 233, 234 (fig.)
 event synchronization, 57, 103, 283, 887
 acquire method, 335
 components, 335–336
 flags for, 106, 107 (fig.)
 global, 106, 113, 353–358
 group, 107
 identifying, 695
 between pairs/groups of processes, 106
 point-to-point, 107 (fig.), 117, 352–353
 release method, 335
 requirements of, through flags, 285 (fig.)
 waiting algorithm, 335
 See also synchronization
 exception program counter (EPC), 915–916, 918–919
 exclusive node, 560
 exclusive pending (EP) state, 925
 exclusive reply, 607
 execution time
 Barnes-Hut application, 174 (fig.), 214
 busy-overhead, 158
 busy-useful, 157
 components, 157 (fig.)
 data-local, 158
 data-remote, 158
 mapping, 159 (fig.)
 MC scaling, 211

multithreading, breakdowns, 913 (fig.)
 Ocean application, 166 (fig.)
 perfect-memory, 210
 as performance metric, 203
 Raytrace application, 179
 synchronization, 158
 explicit communication, 116
 advantages of, 189
 flexibility, 187
See also communication
 explicit synchronization, 285 (fig.)
 exponential backoff, 649, 651
 extended memory hierarchy, 138–140
 artifactual communication in, 139–140
 improving architecture of, 139
 in multiprocessors, 138–139, 271 (fig.)

F

false sharing, 146, 226, 322, 360
 of cache blocks, 711
 cause of, 329, 360
 cross-particle, 365
 effects of, 710
 page fault from, 711
 protocol operations and, 710
 reducing, 328, 329, 361 (fig.)
 SVM communication from, 712 (fig.)
 write-write, 410
 false-sharing misses, 315–316
 block size and, 318
 increase in, 319
 likelihood of, 316
 reducing, 316
 Fast Fourier Transform (FFT), 196
 block data transfer benefits in, 861 (fig.)
 implementation, 549
 kernel performance with consistency models, 874 (fig.)
 radix-2, 548
 sweet spot, 859
 fat cube, 613
 fat-trees, 774
 d-dimensional k-ary, 776
 illustrated, 777 (fig.)
 routing, 778
 fetch&add instruction, 340, 372
 fetch&decrement instruction, 340
 fetch&increment instruction, 340, 346–347
 fetch&op instruction, 611, 613
 at-memory, 651
 four-entry LRU, 617
 fetch&setups instruction, 340
 fetch&store instruction, 540
 fetch deadlock, 390, 412, 483–485
 problem cause, 484
 solution, 484
See also deadlock
 FiberChannel, 637
 Arbitrated Loop, 769
 switch, 637
 FIFO
 buffers, 409, 616, 804
 input/output, 493
 physical, 615
 request-response ordering, 409
 transmit/receive, 500
 virtual, 615
 fine-grained tasks, 83
 finite differencing method, 92
 finite element method, 957
 Firefly update protocol, 374
 fixed-size machine evaluation, 221–226
 inherent behavioral characteristics, 221–222
 problem size range determination, 221
 spatial locality, 224–226
 temporal locality and working sets, 222–224
 flat directory schemes, 565, 567–571
 cache-based, 566, 568–570
 COMA, 704
 full bit vector organization, 567–568
 latency reduction, 586 (fig.), 587 (fig.)
 memory-based, 566, 567–568
 overflow strategy, 568
See also directory schemes
 floating-point operations, 101, 209
 FLOPs, 230, 257
 flow control, 399, 404, 811–818
 ATM, 813
 CRAY T3D network, 820
 end-to-end, 494, 763, 813, 816–818
 in Ethernet network, 812
 link-by-link, 489
 link-level, 813–816
 main memory, 404
 in ring-based LANs, 812
 SGI Challenge, 424
 split-transaction bus, 404
 TCP, 813
 in wide area networks, 812–813
 Flynn's taxonomy, 44
 fork-in approach, 136
 forwarding, 594
 to dirty node, 591
 intervention, 585, 586
 reply, 585, 586, 597
 forward pointer, 569

four-state invalidation protocol, 299–301
 four-state update protocol, 301–305
 fragmentation, 360, 754
 full bit vector organization, 567–568
 full-empty bit, 352–353
 flexibility and, 353
 set to empty, 352
 full-word operation, 15
 function parallelism, 124
 availability, 125
 exploiting, 125
 types of, 124–125

G

galaxy evolution case study. *See*
Barnes-Hut application
 Gaussian elimination, 118
 algorithms, 194
 concurrency, 194–195
 parallelizing, 119
 sequential, pseudocode, 119 (fig.)
 Gauss-Seidel method, 92
 concurrency, 95 (fig.)
 dependencies, 95 (fig.)

generic parallel architecture, 50–52
 communication assist, 50–51
 convergence toward, 50
 organization, 51 (fig.)
See also parallel architectures
 ghost rows, 109, 111
 Gigaplane system bus, 424–427
 centerplane design, 424
 collision-based speculative arbitration, 425
 invalidations and, 427
 outstanding transaction support, 425
 signals, 425
 signal timing, 426 (fig.)
See also Sun Enterprise 6000

global coherence, 855–856
 global synchronization, 95, 96
 barrier algorithms, 356–357
 centralized barrier with sense removal, 354–356
 centralized software barrier, 353–354
 event, 106, 113, 353–358
 between grid sweeps, 96
 hardware barriers, 358
 hardware primitives, 357–358
 performance, 356
 point, 156
See also synchronization
 global trees, 957
 Goodyear MPP, 952
 GO symbol, 815
 Grand Challenge
 application requirements, 7 (fig.)

example, 8
 granularity, 183
 of allocation, 146, 237, 724, 725 (fig.)
 cache block, 706
 coarse, 724
 of coherence, 146, 724, 725 (fig.)
 communication, 186–187, 724, 725 (fig.)
 data transfer, 145
 exploiting, 145
 fine, 724, 725
 memory consistency model and, 681
 page, 725
 Raytrace application, 177
 task, 122–131
 grid computations
 iterative nearest-neighbor, 264
 Ocean application, 162 (fig.), 163, 164.

group event synchronization, 107
 guided self-scheduling, 128

H

half-power point, 60
 HAL S1 multiprocessor, 643–645
 bandwidths, 644
 communication assist, 644
 DMA engine, 644
 MCU, 644
 hardware-controlled prefetching, 880, 881–883
 advantages, 888, 896
 coverage, 889
 disadvantages, 889
 effectiveness, 889
 goal, 881
 history table, 882
 LA-PC, 883
 OBL schemes, 882
 scheduling, 883
 schemes, 881
 software-controlled vs., 888–890
 unnecessary prefetch reduction, 889
See also prefetching
 hardware cost, 188
 access control through code instrumentation and, 707–708
 access control through language/compiler support and, 721–724
 access control with decoupled assist and, 707
 CC-NUMA, 705–706
 page-based access control and, 709–721
 reducing, 705–724

hardware locks, 337–338
 around instruction sequence, 339
See also locks
 hardware/software trade-offs, 679–747
 capacity for replication, 679, 680
 COMA, 701–702
 design/implementation cost, 679, 680
 parallel software implications, 729–730
 waiting time at memory operation, 679, 680
 header, 754, 792
 head node, 624
 rolling out scenario, 631
 of sharing list, 628
 write, 630
 head-of-line blocking, 805
 avoiding with switch design, 807 (fig.)
 impact of, 806
 head pointer, 569
 HEP, 905, 906, 907
 hierarchical coherence, 659–669
 hierarchical directories
 branching factors, 669
 latency problem, 668
 multirooted, 667 (fig.)
 organization, 666 (fig.)
 storage overhead, 667
See also directories
 hierarchical directory scheme, 565–567
 advantages, 566
 bandwidth characteristics, 566
 COMA, 703–704
 latency characteristics, 566
 location information, 566
 processing nodes, 565, 566
See also directory schemes
 hierarchical parallelism, 193
 hierarchical ring-based multiprocessor, 665 (fig.)
 hierarchical snooping, 559, 660–664
 bus hierarchy, 660
 bus transactions, 661
 Encore Gigamax, 663–664
 interconnection network, 666
 processor caches, 660
 remote caches, 660
 write serialization, 663
See also snooping
 Hierarchical Uniform Grid (HUG), 174
 High Performance Fortran (HPF), 723, 967
 hit rate, 943
 home-based protocols, 717
 LRC, 746
 performance advantage, 717–718

home node, 560
 need for local serialization at, 606
 (fig.)
 page, 717
 Horizon design, 905
 cycle, 906
 lookahead, 906
 hot spots
 adaptive routing and, 817
 end-to-end flow control, 817
 network, 760
 HP PA-8000 processor, 684, 868, 940
 hybrid protocols, 330–332
 competitive scheme, 331
 miss rates for, 322 (fig.)
 mixed approaches, 331
 upgrade/update rates for, 333
 (fig.)
See also protocols
 hypercubes, 38, 778
 links, 782
 ports, 778
 uses, 778
See also network topologies

I
 IBM PowerPC, 693, 699
 G30, 940
 systems, 12
 IBM SP-1/SP-2, 40, 41 (fig.), 466
 BT benchmark, 533
 LU benchmark, 533
 network case study, 820–822
 output port, 822
 packets, 821
 switch design, 823
 switch packaging, 821 (fig.)
 workstation as node support, 467
 idle time, 898
 Illiac IV, 67
 implicit communication, 189
 inclusion, 393–394
 fall-out, 394
 maintaining, 394–396
 requirements, 394
 IncPIO requests, 424
 indefinite postponement, 791
 infinite caches, 572
 inherent behavioral characteristics, 221–222
 inherent communication, 140
 inner protocol, 556
 input buffers, 472
 associated with incoming channels, 490
 full, 484
 management of, 482
 overflow, 482–483
 scalability and, 484
 space availability, 484

state of, 483
See also buffers
 input-output microbenchmarks, 215
 instruction-level parallelism, 15–17
 distribution of, 18
 increasing amount of, 17
 superscalar execution, 17
 integration
 board-level, 465–466
 chip-level, 463–464
 system-level, 466–467
 Intel Paragon, 8, 41, 42 (fig.), 68
 in ASCI Red machine, 502, 503
 case study, 499–503
 communication latency, 525
 CP elimination, 525
 DMA engines, 499–502
 DMA transfers, 500, 501
 flow control, 942
 i860XP, 500, 501
 machine organization, 500 (fig.)
 NI chip, 499, 500
 node operating systems, 526
 nodes, 499
 output buffer, 550
 processing elements per node, 740
 send overhead, 524
 shared address read, 527–528
 Intel Pentium Pro, 868, 935
 cache coherence memory bus, 935
 four-processor “quad-pack,” 32 (fig.), 33 (fig.)
 processor consistency model, 698
 processor module, 641
 interconnection network
 design, 749–830
 generic parallel machine, 751 (fig.)
 scalable, 764
See also networks
 interleaved multithreading, 902–910
 advantages, 902, 911
 basic, 905
 blocked scheme vs., 911
 context availability, 918
 control, 920
 disadvantages, 912
 evolution, 904
 implementation issues, 917–920
 latency tolerance, 904 (fig.), 911 (fig.)
 long-latency events and, 908
 overhead, 909
 PC bus, 919 (fig.)
 PC unit, 918–919
 pipeline use, 905–908
 requirements, 914
 short-latency events and, 909
 single-thread pipelined support, 908–910
 speedup, 912 (fig.)
 state replication, 917–918
 Tera, 906–908
See also multithreading
 intervention, 585
 forwarding, 585, 586
 request buffers (IRBs), 615
 invalidation acknowledgments, 607, 699
 invalidation-based protocols, 278
 lock transfers, 675
 MESI (four-state), 299–301
 misses and, 887
 miss rates, 332 (fig.)
 MOESI (five-state), 300
 MSI (three-state), 293–299
 update-based protocols combined with, 330–332
 update-based protocols vs., 329–330
 upgrade/update rates for, 333 (fig.)
 write atomicity, 592–593
 write-back, 293–299, 391
 write-through, 280, 283 (fig.)
 invalidation frequency, 572
 invalidation patterns
 with default data sets, 574–575 (fig.)
 irregular read-write, 573
 migratory, 573
 producer-consumer, 573
 read-only, 572
 invalidation size distribution, 572
 invalid pending (IP) state, 925
 I/O buses, 515
 IQ-Link board, 622, 623, 643
 block diagram, 636 (fig.)
 efficiency, 643
 implementation, 639–641
 IRIX, 415
 itemsets, 81
 iWARP, 69, 494, 495

J
 Jade programming language, 723
 J-machine, 494, 495, 810
 execution contexts, 495–496
 3D torus, 771

K
 kernels, 92, 216
 benchmarks, 967
 data structure sharing, 218
 equation solver, 92–116
 examples, 216
 interaction among, 217

message-passing abstraction
 support, 488
performance-relevant
 characteristics, 217
software, 488
 user linkages, 488
keys, 248

L

LAN
 controllers, 490–491
 flow control, 812
 interfaces, 490–491
 ring, 514
 scalable high-performance, 503
 shared bus, 514
LAPACK suite, 963
latency, 17, 59, 151
 array-based lock, 347
 avoidance, 938
 back-to-back, 619, 620 (fig.)
 bandwidth vs., 59, 787
 Barnes-Hut, 913
 barrier, 356
 of cache misses, 17, 324
 CM-5, 493
 COMA, 729
 communication, 522–523,
 525–526, 833
 contention and, 759, 786, 787
 (fig.)
 dynamic scheduling and, 938
 flat cache-based directory
 protocol, 587 (fig.)
 flat memory-based directory
 protocol, 586 (fig.)
 hiding, 155, 842, 849 (fig.), 870,
 871, 877, 914
 hierarchical directory, 566, 668
 LL-SC lock, 346
 lock, 343
 lock acquisition, 337
 memory, 833
 network, 755–761
 network transaction, 475
 NUMA-Q, 641–642
 performance degradation from,
 17
 phits vs., 788 (fig.)
 pipelined, 618, 911
 reducing, 831–832
 scaling, 460–461
 shared memory approach and, 37
 sharing list purge, 629
 snoop, 383
 test-and-test&set lock, 343
 trade-offs, 784
 true unloaded, 619
 unloaded, 755, 756, 780–785

latency tolerance, 832–951
 application limitations, 843
 approaches, 837–840
 assist occupancy and, 845
 benefit bounds, 843 (fig.)
 benefits, 841–843
 block data transfer, 837, 838, 848
 in blocked multithreading, 903
 (fig.)
 cache-coherent multiprocessor,
 926–927
 communication architecture
 limitations, 843–846
 communication pipeline and,
 836–837
 goals, 836
 in interleaved multithreading,
 804 (fig.)
 interleaved scheme, 911 (fig.)
 limitations, 843–847
 in message passing, 847–851
 multithreading, 840, 850–851
 network capacity and, 846
 no, 834
 overhead and, 845
 overview, 834–847
 point-to-point bandwidth and,
 845–846
 precommunication, 838–839,
 848–850
 proceeding past communication
 in same thread, 839–840, 850
 processor limitations, 846–847
 requirements, 841
 in shared address space, 851–852
 techniques, 647, 926–927
 timelines for different forms, 844
 (fig.)
 write, 876
latency under load, 785–788
 comparison, 786
 with routing delay, 787 (fig.)
latency wall, 940–942
 network interface, 941–942
parallel architecture evolution
 and, 941
speed-of-light, 943
lazy release consistency (LRC), 713
 acquire-based, 738
 complexity, 715
 diffs, 717, 746
 eager vs., 713 (fig.)
 home-based, 746
 implementations, 715
 write notices, 717
 See also release consistency (RC)
least recently used (LRU)
 caches, 259
 replacement, 395
limited pointer directories, 568
LimitLESS scheme, 658
linear arrays, 769, 770 (fig.)
linear scaling property, 209
link-by-link flow control, 489
link-level flow control, 813–816
 handshake, 815 (fig.)
 as host-switch links, 816
 illustrated, 814 (fig.)
 implementation, 813
 problem with, 817
 short-narrow link, 814
 short-wide link, 814
 See also flow control
link-level protocol, 751
links, 751, 764–767
 advancement of, 939
 asynchronous, 765
 bandwidth, 760, 939
 butterfly, 775–776
 CRAY T3D network, 819, 820
 framing, 765
 host-switch, 816
 hypercube, 782
 long, 764, 815 (fig.)
 narrow, 764–765, 815
 propagation delays, 815
 SCI, 766
 SGI Origin network, 825
 short, 764
 synchronous, 765
 torus, 775
 tree, 775
 wide, 764–765
 width, 781 (fig.)
LINPACK benchmark, 13, 503
matrix factorization, 209
microprocessor-based systems on,
 22
supercomputer/MPP performance
 on, 24 (fig.)
uses, 23
 See also benchmarks
Little's Law, 944
livelock, 379, 390, 791
 avoiding, 390
 directory protocols and, 595
 LL-SC implementation and, 392
 NACK solutions, 596
 NUMA-Q, 633
 potential, 379, 390, 392
 problem solution, 380
 SGI Origin2000, 608
 See also deadlock; starvation
load balancing, 88, 123–131
 dynamic partitioning for, 127
 (fig.)
 goal of, 124
 process, 124

- load balancing (*continued*)
 as software responsibility, 131
 static assignment and, 126
 workload, 254–257
- load-locked (LL) instruction, 344, 345
- load-locked, store-conditional
 (LL-SC) locks, 344–346
- exponential backoff and, 649, 651
 guarantees, 345
 guidelines, 345
 implementing, 392, 652
 for implementing atomic
 operations, 344
 latency, 346
 livelock and, 392
 performance, 348–349, 350, 392
- R10000 processor instructions, 611
- spin-lock built with, 346
- storage, 346
See also locks
- local blocks, 560
- local caches, 139
- local coherence, 855
- locality
 characteristics, changing, 141
 exploiting, 57, 139, 142, 143–150
 parallelism and, 14
 spatial, 142, 145–150
 temporal, 142, 143–145
 in tree network, 668
- local memory microbenchmarks, 215
- local node, 560
- local spinning, 543–545
- local state monitor, 661
- lock algorithm, 337, 538–541
 advanced, 346–348
 enhancements, 342–343
 high-contention/low-contention
 and, 675
- lock-free synchronization, 351
- locks
 accessing, 336
 acquiring, 106
 acquisition latency, 337, 343
 array-based, 347–348, 539, 651
 expense of, 106
 fairness, 343
 hardware, 337–338
 LL-SC, 344–346
 performance, 340–342, 348–350
 performance goals, 343–344
 QOLB, 651
 queuing, 651
 scalability, 343
 on SGI Origin2000, 650 (fig.)
 software, 338–340
 software queuing, 539
 storage cost, 343
 test&set, 340, 341 (fig.), 342
- test-and-test&set, 342–343
 ticket, 346–347
 traffic, 343
- lockup-free caches, 414
 Cyber 835 mainframe, 924
 design, 922–926
 design questions, 923–924
 at memory system level, 930
 options, 924
- LogP model, 191
- L₁ caches
 set-associative, 394–395
 write-back, 396
 write-through, 396
- long links, 764
- lookahead buffer, 870
- lookahead program counter (LA-PC), 883
- lookup-free cache design, 922–926
- loop nests, 93, 95
 loops
 analysis of, 93
 bounds, 113
 parallelizing, 124
 parallel, self-scheduling of, 128
 sequential, 93
- low-voltage differential signal (LVDS), 766
- LU benchmark, 532
 application performance, 533
 (fig.)
 communication characteristics, 551 (fig.)
 speedups, 532
 working set curves, 537 (fig.)
See also benchmarks
- LU factorization, 244–248
 computational complexity of, 244
 Gaussian elimination and, 244
 invalidation pattern, 574 (fig.)
 invalidations, 576
 load imbalance, 256
 locks and, 248
 miss rates, 320 (fig.)
 parallel blocked, 247 (fig.)
 sequential blocked, 245 (fig.)
 spatial locality, 320, 322
 speedup, 431
 speedup on SGI Origin2000, 621 (fig.)
 traffic vs. local cache, 582 (fig.)
 traffic vs. number of processors, 580 (fig.)
 writing block in, 576
See also workload case studies
- M**
- machine size, 206
- main memory
 flow control, 404
- out-of-order responses and, 409–410
- SGI Challenge, 415, 420
- subsystem, 405
- Manchester Dataflow Machine, 494
- mapping, 83, 89–90
 Barnes-Hut application, 173
 cumulative performance, 531
 Data Mining application, 182
 execution time, 159 (fig.)
 Ocean application, 165
 parallel algorithms, 153
 Raytrace application, 178
See also parallelization process
- MasPar, 952
- massively parallel processors (MPPs), 23
 dedicated proprietary network, 454
 packaging, 454
 performance, 24 (fig.)
- matrix transposition, 876
 process, 675–677
 sender-initiated, 676 (fig.), 929 (fig.)
- MC (memory-constrained) scaling, 207–208, 431, 433
 communication-to-computation ratio, 212, 433
 concurrency, 212
 execution time, 211
 memory requirements, 212
 naive, 433
 realistic, 433
 scaled speedup, 210
 SGI Origin2000, 621–622
 spatial locality, 213
 speedup, 211, 213
 synchronization, 213
 temporal locality, 213
 work, 211
 working set, 227
See also scaling; scaling models
- media arbitration, 472
- Meiko CS-2, 503–506, 953
 asymmetric CP, 503
 case study, 503–506
 communication assist, 505 (fig.)
 communication latency, 525–526
 conceptual structure, 504 (fig.)
 design shortcoming, 506
 DMA processor, 505, 506
 elan, 504
 flow control, 942
 machine organization, 505 (fig.)
 MBUS, 503
 network transactions, 503, 505
 node architecture, 503
 node operating systems, 526
 page table, 506

- send overhead, 524–525
 shared address read, 528
- memory**
 attraction, 701
 main, 404, 405
 mainframe, 30
 reflective, 515, 518
 shared memory parallel program, 30
 shared virtual, 709–724
- memory accesses**
 average, 942
 frequency of, 310
- memory barrier (MEMBAR),** 688–689, 692–693
- memory-based directory schemes,** 566, 567–568
 latency reduction in, 586 (fig.)
SGI Origin, 596–622
See also directory schemes
- memory consistency, 283–291**
 at acquire, 734–737, 738–739
 at release, 733–734, 737–738
 sequential, 286–291
- memory consistency models, 285**
 cost and, 681
 granularity and, 681
 microprocessor, 699
NUMA-Q, 633
 in real multiprocessor systems, 698–700
 relaxed, 681–700
SGI Origin2000, 607–608
 shared address space, 681
- memory/directory interface (MI), 617**
- memory latency, 833**
 hiding, 870
 two contexts and, 913
See also latency; latency tolerance
- memory management unit (MMU), 507**
- memory operations, 275–276**
 for execution with write-through
 invalidation protocol, 283 (fig.)
 incomplete, in SC, 871
 parallel case and, 276
 reordering, 290
 with respect to processors, 275–276
 sequential consistency, 286
 within instructions, 275
- memory system**
 design, 305
 multiprocessor, 276
SGI Challenge, 424
Sun Enterprise 6000, 429
- memory usage, 206**
- meshes, 38, 770**
 with dimension order routing, 800
- two-dimensional, 771
 uniform wire density, 784
- MESI write-back invalidation protocol, 299–301**
 bandwidth requirement, 307–311
 bus actions, 311
 bus transactions, 299
 cache-to-cache sharing, 300
 exclusive clean state, 299
 lower-level design choices, 300–301
 owned state, 300
SGI Challenge, 422
 shared signal (S), 300
 stable states, 387
 states, 299
 state transition diagram, 301 (fig.), 388 (fig.)
 state transitions, 299–300
 variants, 299
See also MSI write-back invalidation protocol
- message passing, 26, 37–42**
 abstraction, 39 (fig.), 488
 asymptotic bandwidths, 529 (fig.)
 asynchronous, 479 (fig.)
 bandwidth vs. message size, 531 (fig.)
 collective communication operations, 55
 communication, 111
 convergence, 42–43
 equation solver pseudocode, 110–111 (fig.)
 implementation, 43
 libraries, 43, 52
 local references, 187
 machines, 38, 39 (fig.)
 mutual exclusion, 538
 naming, 56
 operations, 43, 56
 operation time vs. message size, 530 (fig.)
 orchestration under, 108–116
 ordering, 56
 parallel software operations, 528–531
 point-to-point events, 538
 primitives, 109
 replication, 185
 research efforts, 68
 send/receive pair, 476, 481
 start-up costs, 60, 529 (fig.)
 store-and-forward, 40
 synchronization, 111, 113
 synchronous, 39, 478 (fig.)
 three-phase protocol, 480
 user communication, 38
 user-level, 476, 477, 490
 uses, 38
- See also* shared address space
message-passing architectures, 37
 node packaging, 37
 processors in, 42
- Message Passing Interface (MPI),** 112, 477, 957, 967
- message passing latency tolerance, 847–851**
 block data transfer, 848
 multithreading, 850–851
 precommunication, 848–850
 precommunication latency hiding, 849 (fig.)
 proceeding past communication in same thread, 850
 structure of communication, 848
See also latency tolerance
- messages, 751**
 flow control, 753
 making larger, 838, 839 (fig.)
 overhead, 755
 size of, 762
- MFLOPS (millions of floating-point operations per second), 230, 231**
- microbenchmarks, 215–216, 967**
 communication, 216
 echo test, 522
 experiment results, 217 (fig.)
 implementation, 216
 input-output, 215
 local memory, 215
 processing, 215
 role of, 216
SGI Origin2000 characterization with, 618–620
synchronization, 216
See also benchmarks
- microprocessors, 63–64**
 architecture design trends, 15–19
 bit-level parallelism, 15
 bus bandwidth, 19, 21
 bus-based shared memory, 20
CISC, 19
 clock frequency improvement, 13
 engineering requirements for, 949
 era, 6
 fraction devoted to memory, 947 (fig.)
 growth curve, 15
 logic density improvement, 13
 markets, 19
 memory consistency models, 699
 parameters, 71 (fig.)
Pentium, 20
 performance, 4, 13
RISC, 19
 transistors, 946, 948 (fig.)
 user-level interrupts and, 491

- MIPS (millions of instructions per second), 230, 231, 257
 MIPS R4000 processor, 910
 MIPS R10000 processor, 597, 684, 868
 misses. *See* cache misses
 miss state holding registers (MSHRs), 924–925
 access, 924
 contents, 924
 state entries, 925 (fig.)
 MOESI protocol, 300
 Monsoon, 494, 495, 496
 MSI write-back invalidation protocol, 293–299
 bus read exclusive, 294
 coherence satisfaction, 297
 illustrated, 295 (fig.)
 invalid state, 293
 lower-level design choices, 298–299
 modified state, 293
 for processor transactions, 297 (fig.)
 sequential consistency
 satisfaction, 297–298
 shared state, 293
 state transitions, 294–296
 transactions and, 293–294
 write to unmodified blocks, 296
See also invalidation-based protocols
 multicast, 120
 multidimensional meshes, 770
 multilevel cache hierarchies, 393–398
 coherence and, 393
 dual tags and, 397
 handling, 393
 inclusion property, 393–394
 internal queues, 411 (fig.)
 intrahierarchy protocol, 397
 propagating transactions in, 396–398
 split-transaction bus with, 410–413
 two-level, 394 (fig.), 398 (fig.)
 multimemory system, 137–142
 multipath routing, 800
 multiple-context processing, 897
 multiple-instruction-multiple-data (MIMD), 44
 multiple outstanding references, 413–414
 exploiting, 414
 semantic implications of, 414
 write buffer example, 413
 multiple writer protocols, 716–718
 with acquire-based consistency, 738–739
 automatic update mechanism, 719 (fig.)
 home-based, 717
 problem, 717 (fig.)
 with release-based consistency, 737–738
 software method, 717
 TreadMark SVM, 716
 multiprocessors, 12
 architectures, 23
 board-level, 948
 cache-coherent, 314, 926–927
 chip-level, 948
 dancehall organization, 459 (fig.)
 directory-based, 553–577
 distributed-memory organization, 458 (fig.)
 extended memory hierarchies, 138–139, 271 (fig.)
 FLASH, 640, 648
 HAL S1, 643–645
 hierarchical bus-based, 661 (fig.)
 hierarchical ring-based, 665 (fig.)
 as memory hierarchy, 140
 performance measures, 202–204
 ring-connected, 442, 443 (fig.)
 scalable, 453–551
 shared memory, 23, 28, 29,
 269–376
 simulations with, 200, 233–234
 snooping cache-coherent, 278 (fig.)
 symmetric, 32–34, 269–271
 use goal for, 121
 Multiprog, 244, 252
 miss rates, 323 (fig.)
 spatial locality, 323–324
 statistic gathering, 254
 traffic as function of cache block size for, 327 (fig.)
 workload, 313, 323
 multiprogrammed workloads, 218
 multistage interconnect, 30–31, 34
 multithreading, 155
 blocked, 898–902, 914–917
 execution time breakdowns, 913 (fig.)
 future of, 939
 hardware-supported, 896–897
 integrating with multiple-issue processors, 920–922
 interleaving, 902–908, 917–920
 latency hiding, 914
 message passing, 850–851
 parallelism, 840
 performance benefits, 910–914
 relaxed memory consistency and, 914
 shared address space, 896–922
 simultaneous, 920–922
 support, 851
 techniques, 898–910
See also latency tolerance; threads
 Munin system, 738
 mutual exclusion, 57, 103, 113, 124, 188, 337–351
 algorithm, 688
 implementation of, 335
 message-passing model, 538
 operation implementation, 337
 shared address space, 538
See also synchronization
 Myricom network, 826–827
 communication assist, 826
 packets, 827
 switches, 827
 Myrinet, 514–515
 case study, 516–518
 communication endpoints, 518
 DMA engines, 517
 Lanai processor, 516
 NIC, 516–518
 NOW organization, 517 (fig.)

N

- naming, 55–56
 Barnes-Hut application, 184
 message-passing model, 56
 Ocean application, 184
 Raytrace application, 184
 shared address model, 55, 184
 narrow links, 764–765
 NAS benchmarks, 966–967
 BT, 532–537
 kernels, 967
 LU, 532, 533, 537–538
 paper-and-pencil, 966
 parallelism costs, 537
See also benchmarks
 n-body problems, 76, 80 (fig.)
 nCUBE, 68, 463
 nCUBE/2, 463–464
 case study, 488–490
 DMA controllers of, 490
 input channels, 489
 machine organization, 464 (fig.)
 network interface organization, 489 (fig.)
 node chip, 463
 nearest-neighbor sharing, 588
 negative acknowledgment (NACK), 591
 input buffer space availability, 484
 lines, 400
 NUMA-Q and, 632
 SGI Challenge, 404
 SGI Origin2000, 597
 network design, 749–830
 case studies, 818–827
 trade-offs, 749–750, 779–788

- network interface card (NIC)
 IBM SP-2, 41
 Memory Channel, 519–520
 Myrinet, 516–518
- network interface (NI), 493, 525, 751, 768
 input/output ports, 768
 Intel Paragon, 499, 500
 packet formatting, 768
 SGI Origin2000, 618
- network(s)
 back pressure and, 483
 bandwidth, 761–764
 Benes, 776, 777 (fig.)
 buffering, 760
 capacity, 846
 closed system, 760
 communication performance, 755–764
 contention, 154
 cost, 782
 deadlock-free, 483
 definitions, 750–755
 delay, 61, 62
 diameter, 753
 direct, 489
 flow control mechanism, 753
 fully connected, 768–769
 “hot spots,” 760
 latency, 755–761
 open system, 763
 organizational structure, 764–768
 performance modeling, 763–764
 routing algorithm, 752–753
 routing distance, 753
 saturation point, 762
 switches, 457
 switching strategy, 753
 wiring complexity, 764
See also network topologies
- networks of workstations (NOWs), 513–521
 communication latency, 526
 convergence, 513
 hardware primitives, 515
 Myrinet SBUS Lanai case study, 516–518
 node operating systems, 526
 organization using Myrinet, 517 (fig.)
 PCI Memory Channel case study, 518–521
 receive overhead, 525
 send overhead, 525
 speedup, 532
- network topologies, 752, 768–778
 butterfly, 774–777
 choice of, 761
 fully connected, 768–769
 hypercube, 778
- latency and, 756
 linear array, 769
 multidimensional mesh, 769–772
 ring, 769
 trade-offs, 779–788
 tree, 772–774
- network transactions, 455, 468
 action, 472
 bus transactions vs., 469
 completion detection, 472
 deadlock avoidance, 473
 delivery guarantees, 473
 destination name and routing, 472
 effects, 469
 flow with symmetric communication, 499 (fig.)
 format, 471, 485
 input buffering, 472
 interpreting, 490
 latencies, 475
 media arbitration, 472
 Meiko CS-2, 503, 505
 Monsoon design, 496
 output buffering, 472
 overhead, 584
 parallel software and, 522–526
 performance, 522–526
 point-to-point, 555
 primitive, 469 (fig.), 470–473
 processing in large-scale architecture, 485 (fig.)
 protection, 471
 transaction ordering, 473
 uncoupled source/destination, 471
 user/system flag, 491
- next PC (NPC), 918–919
- node-level protocols, 752
- nodes, 457
 dirty, 560, 562
 exclusive, 560
 head, 624
 home, 560
 Intel Paragon, 499
 linear array of, 769
 local, 560
 NUMA-Q, 623
 outer protocol, 556
 owner, 560
 pending lists and, 633
 ring of, 769
 routing distance between, 460
 SGI Origin2000, 597
 SMP, 467, 503, 721
 tail, 624
 workstations as, 467
- node-to-network interfaces, 156, 454
 physical DMA, 486–488
 user-level access, 491–493
- nonadaptive routing. *See* deterministic routing
- nonbinding prefetch, 880
- nonblocking asynchronous SEND operation, 115
- nonblocking synchronization, 351
- nonuniform memory access (NUMA)
 approach, 34, 549
 scalable shared memory multiprocessor, 35 (fig.)
 shared memory context, 70
 use of, 36
- NUMA-Q, 556, 622–645, 700
 average remote miss latency components, 643 (fig.)
 bandwidth, 641
 case study, 622–645
 coherence controller, 731
 comparison case study, 643–645
 correctness issues, 632–634
 DataPump, 641, 642
 deadlock, 633
 directory protocol, 624
 directory state diagrams, 670 (fig.)
 directory structure, 624
 error handling, 633–634
 hardware overview, 635–637
 interquad I/O transfers, 637
 I/O subsystem, 637 (fig.)
 IQ-Link board, 622, 623, 643
 IQ-Link board block diagram, 636 (fig.)
 IQ-Link implementation, 639–641
 latency, 641–642
 livelock, 633
 memory consistency model, 633
 microbenchmark characteristics, 642 (fig.)
 multiprocessor block diagram, 623 (fig.)
 NACKs and, 632
 performance characteristics, 641–643
 processing nodes, 623
 processor consistency model, 698
 protocol extensions, 634–635
 protocol interactions with SMP node, 637–639
 read request handling, 626–628
 remote cache, 623–624, 700
 SCI ring, 824
 SCI specification, 822
 SCLIC, 635, 638, 639–640
 serialization of operations to given location, 632–633
 serialization of writes, 673–674
 sharing list, 624, 625 (fig.), 628
 SMPs, 623

NUMA-Q (*continued*)

- SMPs as nodes, 588
 - starvation, 633
 - states, 624–626
 - third-party commodity hardware, 622
 - TPC-B benchmark, 642, 643
 - TPC-D benchmark, 642, 643
 - workload characteristics, 642 (fig.)
 - write-back request handling, 630–632
 - write request handling, 628–630
- See also* SGI Origin2000

O

object-based coherence, 721–723

- advantages/disadvantages, 722

- synchronization events, 723

- See also* coherence

- occupancy, 61

- Ocean application, 76, 77–78, 161–166

- assignment, 83, 161–163

- block transfer benefits in, 861 (fig.)

- capacity misses, 324

- concurrency, 78

- decomposition, 161–163

- dependences among grid computations, 162 (fig.)

- equation solver kernel, 92–116

- execution time breakdown, 166 (fig.)

- grid arrays, 90

- grid computations, 162 (fig.), 163, 164

- horizontal cross sections, 78 (fig.)

- invalidation pattern, 574 (fig.)

- invalidations, 576

- iterative nearest-neighbor grid computations, 891

- mapping, 165

- miss rates, 321 (fig.)

- models, 77

- naming, 184

- nearest-neighbor sharing, 588

- on Origin2000 machine, 149

- orchestration, 163–165

- owner computes arrangement, 90

- prefetching remote data in, 892 (fig.)

- process streams, 261

- scaling, 432

- sequential algorithm, 161

- spatial locality, 163–164, 320, 321

- speedup, 431

- speedup on SGI Origin2000, 621 (fig.)

- summary, 165

- synchronization, 165

- tasks, 82

- temporal locality, 164

- time-step phases, 162 (fig.)

- traffic vs. local cache, 582 (fig.), 583 (fig.)

- traffic vs. number of processors, 580 (fig.), 581 (fig.)

- two-dimensional grids, 77

- working sets, 164

- See also* case studies

- offered bandwidth, 762

- one-block lookahead (OBL) schemes, 882

- on-line bipartite matching, 810

- on-line transaction processing (OLTP) benchmarks, 10

- open systems, 763

- operations

- atomic, implementing, 391–393, 652

- competing, 695

- conflicting, 695

- critical section, 105

- floating-point, 101, 209

- global communication, 817–818

- interleaving, prohibiting, 105

- lock-based, 351

- message passing, 43, 56

- out-of-order processing of, 290

- parallel prefix, 546–547

- programming model, 56

- RECEIVE, 112, 114–115

- release, 692, 733–734

- SEND, 112, 114–115

- serial order of, 276

- shared address model, 56

- optimization, 219–220

- algorithmic, 219

- data layout, distribution, alignment, 220

- data structuring, 219

- orchestration, 220

- orchestration, 83, 89

- Barnes-Hut application, 170–173

- choices, 89

- Data Mining application, 181–182

- under data parallel model, 99–101

- under message-passing model, 108–116

- Ocean application, 163–165

- optimization, 220

- for performance, 142–156

- performance goals, 89

- Raytrace application, 176–177

- under shared address space model, 101–108

- See also* parallelization process

- ordering, 56–57

- bus transaction properties, 473

- channel, 794 (fig.)

- message-passing model, 56

- network transaction, 473

- partial store (PSO), 686, 689, 865

- red-black, 96, 97 (fig.)

- relaxed memory (RMO), 686,

- 690, 693, 699, 866

- request-response, 409

- shared address model, 57

- static, 445

- synchronization and, 57

- total store (TSO), 686, 865, 866, 867

- weak (WO), 686–687, 690–691,

- 699, 866, 867

- Orion bus interface controller (OBIC), 635, 638, 639

- orthogonal recursive bisection (ORB), 170

- Barnes-Hut, 171 (fig.)

- hypercube technology mapping, 173

- outer protocol, 556

- output array, 248

- output scheduling, 808–810

- algorithm, 808

- control structure, 809 (fig.)

- overflow

- broadcast scheme, 655

- coarse vector scheme, 656–657

- dynamic pointers scheme, 658

- methods for reducing directory width, 655–658

- no broadcast scheme, 655–656

- pointers, 658

- software scheme, 657–658

- overhead, 61

- block transfer, 242

- cache block, 242

- communication, 186–187, 487

- context switch, 901

- directory memory, 667

- endpoint, 183, 846

- hierarchical directory storage, 667

- interleaved scheme, 909

- latency tolerance and, 845

- message, 755

- network transaction, 584

- per-message, amortized, 857

- receive, 522, 523, 525

- reducing, 151–152, 659

- send, 522, 523–525

- single-thread support scheme, 909

- thread-switching, 898–899

- unready thread cost, 909 (fig.)

- overlap, 63

- overlapping transactions, 586

owner computes arrangement, 90
owner node, 560

P

packet buffers, 820
packets, 751
 CRAY T3D formats, 819 (fig.)
 crossing channels, 756
 envelope, 756
 format, 754 (fig.)
 formatting, 768
 fragmentation, 755
 full buffers and, 759
 header, 754, 792
 IBM SP network, 821
 Myricom network, 827
 parts, 754
 payload, 754
 SCI, formats, 825 (fig.)
 trailer, 754, 766
packet switching, 758, 759
page-based access control, 709–721
page fault handlers, 709
page granularity, 725
page migration
 automatic, 729
 explicit, 729
 SGI Origin2000 and, 610–611,
 729
page table entries (PTEs)
 locking, 440
 modifying, 439
pairwise sharing, 624
PAL code, 526
parallel architectures
 abstraction layers, 27 (fig.), 52
 commercial computing and, 9–10
 convergence of, 25–52
 cost-performance trade-offs and,
 4
 dataflow, 47–49
 data parallel processing, 44–47
 efficiency, 230
 engineering component, 64
 evolutionary scenario, 937–940
 evolution of, 2
 form and function elements, 1
 future of, 935–961
 generic, 50–52
 layers of, 469 (fig.)
 learning process, 2–3
 message passing, 37–44
 potential breakthroughs, 944–955
 reasons for, 4–25
 role in information processing, 2
 scaling in, 467–468
 scientific computing and, 8
 shared address space, 28–37
 systolic, 49–50

technology and, 936–955
parallel computers, 1
 cost-performance trade-offs, 2
 hardware/software
 responsibilities in, 2
 market pyramid, 937 (fig.)
 operating systems, 959
 performance characteristics, 202
 programming model realization,
 468
parallelism, 14
 attractiveness of, 5
 bit-level, 15
 computer architecture and, 1
 costs, 537
 data, 124, 125
 exploiting, 57
 freedom of, 75–76
 function, 124
 hierarchical, 193
 importance of, 12
 instruction-level, 15–17
 learning curve, 9
 levels of, 193, 194 (fig.)
 locality and, 14
 multithreading, 840
 1960s innovations, 67
 performance improvement and, 6
 pipeline, 119, 120, 124, 125
 process-level, 19
 thread-level, 17–19
parallelization process, 81–92
 data, 90–91
 example program, 92–116
 goals of, 82, 91–92
 gradual, 366
 steps in, 82–90
parallel languages, 958
parallel prefix operation, 546–547
 downward sweep, 547 (fig.)
 performance, 546
 upward sweep, 546 (fig.)
parallel programming
 languages, immaturity of, 200
 models, 26
 transition to, 12
parallel programs, 75–120
 algorithm designers and, 75
 architects and, 75
 case studies, 76–81
 debugging, 960
 deterministic, 96
 example parallelization of,
 92–116
 immaturity of, 200
 load imbalanced, 88
 programmers and, 75
 speedup of, 122
 statistics about, 255
Parallel Random Access Memory
 (PRAM) model, 136–137,
 191, 718, 908
algorithm development for, 137
speedups on, 254
usefulness of, 191
parallel software, 522–538
 application-level performance,
 531–538
 categories, 955
 directory-based cache coherence
 implications, 652–655
 evolutionary scenario, 955–960
 hardware/software trade-offs and,
 729–730
 message-passing operations,
 528–531
 network transaction performance,
 522–526
 potential breakthroughs, 961
 shared address space operations,
 527–528
 walls, 960–961
parallel vector processors (PVPs), 23,
 24 (fig.)
parallel virtual machine (PVM), 52
parameter space, 238–243
 associativity, 241
 cache block size, 241
 cache/replication size, 239–240
 goals, 238
 number of processors, 238–239
 performance parameters of com-
 munication architecture, 242
 problem size, 238–239
PARKBENCH, 967–968
partial store ordering (PSO), 686,
 689, 865
partitioning, 83, 123–137
 Barnes-Hut application, 169, 171
 (fig.)
 dynamic, 126, 127
 goal of, 132, 135–136
 image plan, 177 (fig.)
 into contiguous subdomain, 135
 principle, 359
 profiling-based semistatic, 169
 repartitioning, 134
 specifying, 101
 static block, 176
 techniques, 136
 trade-offs, 135
patches, 249–250
 interaction list, 250, 253 (fig.)
 light transfer between, 250
payload, 754
PCI Memory Channel, 518–521
 communication assist, 518
DMA engine, 520

- PCI Memory Channel (*continued*)
 interface, 719
 NIC, 519–520
 page control table (PCT), 520
 shared physical address space, 518
 write doubling mechanism, 718
- PC (problem-constrained) scaling, 207
 communication-to-computation ratio, 212
 concurrency, 212
 memory requirements, 212
 problem size, 208
 spatial locality, 213
 speedup, 208, 213
 synchronization, 213
 temporal locality, 213
 working set, 227
See also scaling; scaling models
- pending lists
 distributed, 634
 SCI, 629 (fig.)
 total number for nodes, 633
- pending states, 590
- Pentium. *See* Intel Pentium Pro
- Perfect Club benchmarks, 968
- perfect-memory execution time, 210
- performance, 59–63, 121–197
 absolute, 202, 203, 228–229
 application parameter impact on, 202
 array-based lock, 350, 651
 block data transfer benefits, 856–863
 COMA, trade-offs, 702–703
 communication, 755–764
 components, 59
 cost trade-off and, 4
 data transfer time, 60–61
 directory protocol, 584–589, 645–648
 example, 59–60
 factors, 156–159
 floating-point, 13
 hierarchical coherence and, 668–669
 isolation with microbenchmarks, 215–216
 lock, 340–342, 348–350
 microprocessor, 4, 5 (fig.), 13
 modeling, 189, 763–764
 MPPs, 24
 multiprocessor, measures, 202–204
 multithreading, 910–914
 NUMA-Q, 641–643
 occupancy and, 61
 orchestration for, 142–156
 overhead and, 61
- partitioning for, 123–137
 percentage improvement in, 231
 processor, growth, 4, 5 (fig.)
 programming, 121–197
 relaxed consistency models and, 700
 SGI Origin2000, 618–622
 supercomputer, 24
 SVM, 720–721
 synchronization algorithm, 649–651
 trade-offs, 122, 123
 uniprocessor, supercomputer, 22 (fig.)
 workstation, 71 (fig.)
- performance metrics, 228–231
 absolute performance, 228–229
 choosing, 228–231
 execution time, 157–159, 179, 203
 percentage improvement, 231
 problem size, 230–231
 processing rate, 230
 rate-based, 262
 speedup. *See* speedup
 utilization, 230, 262
- perimeter-to-area ratio, 132, 133 (fig.)
- peripheral component interface (PCI) buses, 635, 636, 637
- per-processor heaps, 363
- phits, 752
 flits vs., 753
 latency vs., 788 (fig.)
- physical DMA, 486–491
 blind, communication assist for, 487 (fig.)
 communication abstraction implementation, 488
 LAN interfaces, 490–491
 nCUBE2, 488–490
 node-to-network interface, 486–488
See also direct memory access (DMA)
- physical protocol, 751
- physical scaling, 462–468
 board-level integration, 465–466
 chip-level integration, 463–464
 system-level integration, 466–467
See also scaling
- pipelined latency, 618, 911
- pipelined parallelism, 119, 120, 124–125
 availability, 125
 example, 125
- pipeline stalls, 901
- pipelining
 cost, 900
 late miss detection in, 901 (fig.)
- multiple memory operations, 863
 strategy, 400–401
 of writes, 866
- pivot element, 118
- pivot row, 118, 119
- “platform pyramid,” 6
- PLUS system, 718
- P-node trees, 544
- point-to-point bandwidth, 151, 845–846
- point-to-point events, 538
- point-to-point network transactions, 555
- point-to-point synchronization, 96, 172
 data dependences and, 130
 event, 107 (fig.), 117, 352–353
 full-empty bits and, 352–353
 hardware support, 352–353
 interrupts, 353
 software algorithms, 352
- polling, 482
- PowerChannel-2 I/O subsystem, 422–424
 HIO interface chips, 422
 organization, 423 (fig.)
See also SGI Challenge
- Powerpath-2 system bus, 417–420
 acknowledge cycle, 419
 address/data buses, 418
 data resource ID line, 419
 distributed arbitration scheme, 418–419
 four-cycle sequence, 420
 signals, 417
 state transition diagram, 418 (fig.)
 timing diagram, 419 (fig.)
 urgent requests, 418
See also SGI Challenge
- precommunication, 155, 838–839
 latency hiding through, 849 (fig.)
 long-latency events and, 839
 message passing, 848–850
 with multiple-issue, dramatically scheduled processors, 895
 performance benefits, 891–896
 prefetching, 850, 878
 in receiver-initiated communication, 839
 relaxed consistency comparison, 895–896
 sender-initiated, 890–891
 shared address space, 877–896
 with single-issue, statically scheduled processors, 891–894
 summary, 896
 without caching of shared data, 877–879
See also latency tolerance
- predicted PC, 919

- prefetch buffers, 878
 prefetching, 434, 850
 analysis, 880, 881
 binding, 880
 block, 880
 compiler-generated, performance
 benefits, 893 (fig.)
 concepts, 880–881
 contention and, 895
 coverage, 881
 effectiveness, 895
 effects for SC, 874
 hardware-controlled, 880,
 881–883
 hardware-controlled vs. software
 controlled, 888–890
 implementation issues, 896
 ineffectiveness, 872
 nonbinding, 880
 on local accesses, 891
 with ownerships, 888 (fig.)
 precommunication, 850, 878
 relaxed memory consistency
 comparison, 895–896
 remote data, 891
 remote data, performance
 benefits, 892 (fig.)
 reorder buffer operations, 876
 scheduling, 880, 883
 selective, benefits of, 894 (fig.)
 in shared address space, 879 (fig.)
 with single processor, 884–887
 software-controlled, 880, 884
 timing, 881
 unnecessary, 881, 884
 presence bits, 496, 560
 Princeton SHRIMP designs, 521
 private address space, 112
 problem sizes, 206
 artificial communication and,
 223
 choosing, 238–239
 choosing based on working sets,
 224 (fig.)
 effect of, 227 (fig.)
 impact on spatial locality
 behavior, 225 (fig.)
 metric, 230–231
 PC scaling, 208
 range determination, 221
 TC scaling, 208
 proceeding past communication in
 same thread, 839–840
 buffering and, 863
 message passing, 850
 pipelining and, 863
 reads, 868–876
 in receiver-initiated
 communication, 840
 shared address space, 863–877
 summary, 876–877
 writes, 864–868
 See also latency tolerance
 proceeding past reads, 868–876
 dynamic scheduling, 870
 enhancing, 871–872
 performance impact, 873–876
 release consistency, 871
 sequential consistency, 871
 speculative execution, 870–871
 speculative reads, 872
 proceeding past writes, 864–868
 cache size/cache block size effects
 on, 869 (fig.)
 performance benefits, 867 (fig.)
 performance impact, 865–868
 write buffer and, 867–868
 processes, 29, 83
 communication between, 59
 descheduling, 233
 execution intervals, 736
 pinning, 89
 relationships of, 84 (fig.)
 serialization of, 124
 shared memory machine, 49
 processing elements (PEs), 44–45
 condition flag, 45
 systolic architecture, 49
 “virtualizing,” 46
 processing microbenchmarks, 215
 processing rate, 230
 process-level parallelism, 19
 processor-and-memory (PAM), 946,
 950, 954, 955
 processor-cache handshake, 388
 processor consistency (PC), 686, 866,
 867
 comparison, 688 (fig.)
 model, 698
 write ordering preservation, 687
 See also system specification
 processor-in-memory (PIM), 951,
 952 (fig.)
 processor interface (PI), 615–617
 physical FIFO, 615
 request numbers, 617
 shielding, 617
 See also SGI Origin2000
 processors
 arrays, development of, 45
 building blocks, 4
 bus, assist, interface sharing, 588
 in bus-based shared memory
 microprocessors, 20
 dynamically scheduled, 895, 922
 HP PA-8000, 684
 initiator, 440
 massively parallel (MPPs), 23
 message-passing machine, 42
 MIPS R10000, 597, 684, 868
 number, choosing, 238–239
 number, effect of, 227 (fig.)
 parallel vector (PVPs), 23, 24
 (fig.)
 performance growth, 4, 5 (fig.)
 relationships of, 84 (fig.)
 statically scheduled, 891–894
 status word, 915
 superscalar, 895
 trust between, 453
 utilization, 230
 vector, 21–22
 program counters (PCs), 915–916,
 918–919
 blocked multithreading, 915–916
 bus, 919 (fig.)
 chain, 915, 918
 exception (EPCs), 915–916,
 918–919
 interleaved multithreading,
 918–919
 managing, 916
 next (NPC), 918–919
 predicted, 919
 programmer’s interface, 686
 access labels, 697
 consistency model, 694
 relaxed models at, 699
 synchronization event
 identification, 695
 See also relaxed memory
 consistency models
 programming languages, 682
 High Performance Fortran, 723
 Jade, 723
 labeling support, 729
 parallel, immaturity of, 200
 Split-C, 724
 variable declaration attribute, 697
 programming models, 26
 abstraction, 28
 Active Messages, 481–482
 challenges, 482–485
 data parallel, 44–47
 differences in, 183
 implementation of, 183
 implications for, 182–190
 message passing, 37–44, 476–481
 naming, 55–56
 operations, 56
 ordering, 56–57
 requirements, 53–57
 shared address space, 37, 473–476
 shared memory processor, 29
 programming performance, 121–197,
 960
 program orders, 687–694
 relaxing all, 689–694
 write-to-read, relaxing, 687–689
 write-to-write, relaxing, 689

properly labeled model, 695
 protocol design trade-offs, 305–334
 bandwidth requirement, 307–311
 in cache block size, 313–319
 decisions, 306
 impact of, 311–313
 methodology, 306–307
 per-processor bandwidth
 requirements, 312 (fig.)
 update-based vs. invalidation-based and, 329–334

protocols
 home-based, 717
 hybrid, 330–332
 inner, 556
 invalidation-based, 278, 280, 283 (fig.)
 multiple writer, 716–718
 outer, 556
 processing, 708
 space of, for write-back caches, 283
 update-based, 278, 292

Q

queue-on-lock-bit (QOLB)
 synchronization, 626, 651

queues
 centralized, 128
 communication, 498
 distributed, 128
 inside multilevel cache hierarchy, 411 (fig.)
 structures, 413
 task, 128, 129 (fig.)
 queuing locks, 651

R

Radiosity, 244, 249–252
 barrier synchronization, 252
 BSP tree, 250
 interaction list, 250, 253 (fig.)
 invalidation pattern, 575 (fig.)
 invalidations, 577
 load imbalance, 256
 miss rates, 320 (fig.)
 patches, 249–250
 quadtrees, 251 (fig.)
 spatial locality, 322
 speedup, 431
 speedup on SGI Origin2000, 621 (fig.)
 traffic vs. local cache, 582 (fig.)
 traffic vs. number of processors, 580 (fig.)
 See also workload case studies

Radix, 244, 248–249, 267
 bandwidth requirements, 327
 digits, 248

dynamic scheduling, 875 (fig.)
 false-sharing effect in, 323
 invalidation pattern, 574 (fig.)
 invalidations, 576
 kernel sorting, 430
 keys, 248
 local keys, 248
 miss rates, 321 (fig.)
 permutation step, 249 (fig.)
 process streams, 261
 sharing behavior, 323
 sorting, 249 (fig.)
 sorting kernel, 576
 speculative execution, 875 (fig.)
 speedup, 256, 431
 speedup on SGI Origin2000, 621 (fig.)
 traffic vs. local cache, 582 (fig.), 583 (fig.)
 traffic vs. number of processors, 580 (fig.), 581 (fig.)
 See also workload case studies

RAM
 address map, 423
 arrays, 635
 Computational, 951
 dual-ported, 382

ray-oriented approach, 175
 Raytrace application, 77, 79–80, 174–178
 artifactual communication, 176
 assignment, 175–176
 capacity misses, 324
 decomposition, 175–176
 dynamic tasking, 127
 execution time breakdown for, 179
 granularity, 177
 HUG, 174
 image plane partitioning, 177 (fig.)
 invalidation pattern, 574 (fig.)
 invalidations, 577
 mapping, 178
 miss rates, 321 (fig.)
 naming, 184
 orchestration, 176–177
 ray-oriented approach, 175
 rays, 174, 175, 176
 scene-oriented approach, 175
 sequential algorithm, 175
 serialization, 256
 spatial locality, 176–177
 speedup, 431
 speedup on SGI Origin2000, 621 (fig.)
 subspace, 175
 summary, 178
 synchronization, 177
 tasks, 82

temporal locality, 177
 traffic vs. local cache, 582 (fig.), 583 (fig.)
 traffic vs. number of processors, 580 (fig.), 581 (fig.)
 working sets, 177

ray tracing, 79–80
 read exclusive, 292
 bus transactions, 292
 cache coherence and, 294
 writing cache and, 297

read misses, 662, 744
 blocking, 864, 877
 proceeding past, 868–876

read-only sharing, 572, 588

read requests
 buffers (RRBs), 615
 NUMA-Q, 626–628
 SGI Origin2000, 599–601

read-to-own. See read exclusive

read-write communication, 852

ready threads, 898

realistic scaling, 266

RECEIVE operation, 112
 blocking asynchronous, 115
 nonblocking asynchronous, 115
 semantics, 114–115
 synchronous form, 112, 114

receive overhead, 522
 comparison, 525
 message breakdown, 523 (fig.)

red-black ordering, 96
 advantages, 96
 of equation solver, 97 (fig.)

REDUCE statement, 100 (fig.), 101

reducing
 application miss rate, 325
 artifactual communication, 142–150
 capacity misses, 314
 cold misses, 314
 communication, 123, 131–135
 concurrency, 99, 101
 conflict misses, 314
 contention, 153–154
 delay, 152–153
 extra work, 123, 135–137
 false sharing, 316, 328, 329, 361 (fig.)
 overhead, 151–152
 serialization, 130–131
 spatial interleaving, 360–362
 true-sharing misses, 316

reference generators, 233

reflective memory, 515
 address space organization, 519 (fig.)

receive region, 518
 transmit region, 518

register insertion rings, 443, 444

- relaxed memory consistency models, 681–700
 intuition behind, 684, 685 (fig.)
 multithreading and, 914
 performance and, 700
 prefetching comparison, 895–896
 programmer's interface, 686,
 694–697
 in real multiprocessor systems,
 698–700
 single writer with consistency at
 acquire, 734–737
 single writer with consistency at
 release, 733–734
 software implementation,
 732–739
 solution components, 685–686
 system specification, 685–694
 translation mechanism, 686, 698
 using, 711–716
See also memory consistency
 models
- relaxed memory ordering (RMO),
 686, 690, 693, 699, 866
- release-based consistency, 737–738
- release consistency (RC), 687, 690,
 691–692, 699, 866, 867
 acquire, 692
 conservative, 722 (fig.)
 eager, 712, 713 (fig.)
 hardware implementations, 715
 incomplete acquire operation in,
 871
 interface, 692
 latency hiding under, 871
 lazy, 713 (fig.), 715, 717, 738, 746
 non-null pointer value, 715–716
 at programmer's interface, 699
 release, 692
 weak ordering vs., 691 (fig.)
See also system specification
- release method, 335
- release operation, 692, 733–734
- remote blocks, 560
- remote caches, 623–624, 660, 662
 access time, minimizing, 663
 DRAM, 663, 700
- renaming, 18
- reorder buffers, 414, 870, 876
- repartitioning, 134
- replication, 184–186
 artifactual communication and,
 140–142
 communication and, 58–59
 fine-grained, 729
 finite capacity, 140
 limited capacity, 679, 680
 local, 140
 management, 185–186, 724
 message-passing model, 185
- shared address space, 185
 shared address space without,
 723–724
 size, 239–240
 state, 914–915, 917–918
- reply forwarding, 585
 outstanding requests and, 586
 SGI Origin2000, 597, 602
- requests, 398
 conflicting, 398–399, 402–403
 IncPIO, 424
 outstanding, 399
 urgent, 418
 writes and, 403
See also split transaction bus
- request tables, 402
 fully associative, 409
 requests entered into, 405
- resource-oriented properties, 207
- resources
 contention for, 62
 hot spot, 154
 occupancy, 153
 resource, 154
- responders, 382
- response transaction, 398
- ring access control mechanism,
 442–443
- ring-connected multiprocessors, 442,
 443 (fig.)
- ringlets, 822
- rings, 441–444, 555, 769
 advantages, 441–442
 bandwidth on, 443, 444
 broadcast media, 442
 disadvantages, 442
 illustrated, 770 (fig.)
 interface, 442
 misses on, 444
 register insertion, 443, 444
 sequential consistency on, 441,
 444
- serialization on, 441, 444
 slotted, 443
- snooping cache coherence on,
 441
- token-passing, 442–443
- uses, 769
See also network topologies
- RISC microprocessors, 19, 53, 68
- routes, 752
- routing, 789–801
 adaptive, 790, 799–801
 bits, 790
 butterfly, 778
 cut-through, 757 (fig.), 782
 deadlock, 791–792
 deadlock-free, 793
 delay, 460, 758, 761, 781 (fig.)
 deterministic, 790–791
- dimension order, 789, 800
 e-cube, 778, 790
 fat-tree, 778
 mechanisms, 789–790
 multipath, 800
 operations at switches, 789
 source-based, 790, 805
 store-and-forward, 756, 757
 (fig.), 758
- table-driven, 790
 turn-model, 797–799
 up*–down*, 796–797, 799
- wormhole, 759, 794
- routing algorithms, 752–753
 adaptive, 790
 classes of, 789
 deadlock-free, 793, 794
 deterministic, 791
 minimal, 791
 multipath, 800
 nonminimal, 791
 properties of, 789
 west-first, 797–798, 799 (fig.)
- routing distance, 753, 756, 779
 average, 753
 CRAY T3D network, 820
 cut-through, 779
 store-and-forward, 779
- run length, 899
- S**
- saturation point, 762, 763 (fig.)
- scalability, 455, 456–468
 input buffering and, 484
 limitations overcome by, 456–457
- scalable cache coherence, 558–559
- Scalable Coherent Interface (SCI)
 case study, 822–825
 interconnection across quads, 635
 links, 766
 packet formats, 825 (fig.)
 pending lists, 629 (fig.)
 purging sharing list in, 630 (fig.)
 read miss, 627 (fig.)
- ringlets, 822
- rings, 636, 643
- serialization, 639
- sharing list, 625 (fig.)
- standard, 570, 635
- transactions, 824
- transport layer, 636
- scalable machines
 abstract view, 458 (fig.)
 configuration support, 461
 dancehall organization, 459 (fig.)
 distributed-memory organization,
 458 (fig.)
 sequential consistency in, 475
- scalable multiprocessors, 453–551
 array-based lock problems, 539

- scalable multiprocessors (*continued*)
 dedicated message processing, 496–506
 with directories, 555 (fig.)
 parallel software implications, 522–538
 synchronization, 538–547, 655
 user-level access, 491–496
- Scalapack suite, 963
- scaled speedup, 210
- scaling, 202–214
 bandwidth, 457–459
 bisection, rule, 788
 caches, 236–237
 cost, 461–462
 data in bus-based systems, 445–446
 directory protocol, 564–565
 efficiency-constrained, 231
 error and, 265
 in generic parallel architecture, 467–468
 importance of, 204–206
 key issues in, 206–207
 latency, 460–461
 linear, property, 209
 machines, 206
 physical, 462–467
 questions, 207
 realistic, 266
 relationships, preserving, 235
 SGI Origin2000, 621–622
 snoop bandwidth, 445–446
 unloaded latency, 780 (fig.)
 VLSI, 802
 workload parameters, 213–214
- scaling down problem, 234–237
 cache size for, 237 (fig.)
 confidence and, 237
- scaling models, 205, 207–213
 impact on equation solver kernel, 211–213
 memory-constrained (MC), 207–208, 210–211, 431, 433, 621–622
 problem-constrained (PC), 207–208, 212–213, 227
 time-constrained (TC), 207, 208–210, 431, 433
- scatter decomposition, 176
- scene-oriented approach, 175
- scheduling
 algorithm, 808
 based on latency, 885
 dynamic, 870
 output, 808–810
 prefetch, 880, 883
- scientific/engineering computing, 6–9, 218–219
- SCI link interface controller (SCLIC), 635, 638, 639
 chip block diagram, 641 (fig.)
 coherence controller, 640
 directory controller, 639
 engine, 640
 memory-mapped counters, 641
 scope consistency, 723
 segmented register file, 900
 for multithreaded processor, 915
 (fig.)
 statically, 914
- self-scheduling, 128
- sender-initiated communication, 833, 879
 precommunication, 890–891
 software-controlled, 890
- sender-initiated matrix transposition, 676 (fig.), 929 (fig.)
- SEND operation, 112
 blocking asynchronous, 115
 nonblocking asynchronous, 115
 semantics, 114–115
 synchronous form, 112, 114
- send overhead, 522
 comparison, 523–524
 message breakdown, 523 (fig.)
- sense reversal, 355
- Sequent Computer Systems.
See NUMA-Q
- sequential algorithms
 Barnes-Hut application, 166–169
 Data Mining application, 179–180
 Ocean application, 161
 providing, 389
 Raytrace application, 175
- sequential consistency (SC), 286–291, 865
 comparison, 688 (fig.)
 execution model, 695
 implementing, 287
 incomplete memory operation in, 871
 interleaving, 288, 695
 invalidations and, 710
 latency hiding under, 871
 memory operations, 286
 model, 682
 MSI invalidation protocol and, 297–298
 performance limitations, 684
 prefetching effects for, 874
 preservation conditions, 289–291
 at programmer's interface, 683
 register allocation by compiler
 and, 683 (fig.)
 in rings, 441, 444
 in scalable machines, 475
 semantics, 685, 688, 697
- sequentially consistent execution, 288
- sequentially consistent system, 288
- serialization and, 289, 592–593
- split-transaction bus, 406–409
 for SVM, 711 (fig.)
- in two-level hierarchical bus
 design, 677
 write atomicity, 288, 289 (fig.)
 write buffers, 865, 874
- sequential loops, 93
- sequential program order, 53, 54
- serialization, 388–390
 across locations for sequential consistency, 592–593
 buffer at home, 590–591
 buffer at requestor, 591
 for bus-based machines, 291
 forward to dirty node, 591
 globally consistent, 590
 to location for coherence, 589–591, 604–607, 632–633
 minimizing, 124
 NACK and retry, 591
- of operations at different locations, 406
- Raytrace application, 256
- reducing, 130–131
- in rings, 441, 444
- SCI protocol, 639
- sequential consistency and, 289
- split-transaction bus, 406–409
- write, 277, 288, 389, 589, 663
- SGI Challenge, 311
- application performance, 429–433
- application speedups, 430–431
- barrier performance on, 357 (fig.)
- bus architecture, 400
- bus interface chips, 420
- cache-to-cache transfers, 384
- case study, 415, 417–424
- chip types, 420
- design, 415
- flow control, 424
- I/O subsystem, 422–424
- lock performance on, 348–350
- main memory, 415, 420
- memory subsystem, 420–422
- memory system performance, 424
- MESI protocol, 422
- NACK lines, 404
- operating system, 415
- outstanding transactions, 449
- Powerpath-2 system bus, 417–420
- processor board chip partitioning, 421 (fig.)
- processor subsystem, 420–422

- read microbenchmark results, 425 (fig.)
 scaling results, 432
 system organization, 416 (fig.)
See also Sun Enterprise 6000
 SGI Origin2000, 36–37, 150, 160, 556, 596–622
 access protection rights, 608–609
 application speedups, 620–621
 arbitrator, 614
 automatic page migration support, 610–611
 back-to-back latency, 619, 620 (fig.)
 bandwidths, 618
 Barnes-Hut application on, 174
 busy states, 597, 600
 cache coherence protocol, 597–604
 case study, 596–622
 characterization with microbenchmarks, 618–620
 correctness issues, 604–609
 deadlock, 608
 deadlock detection, 595
 directory lookup, 599
 directory protocol, 588
 directory state diagrams, 670 (fig.)
 directory structure, 598–599, 609
 DMA operation support, 610
 error handling, 608–609
 fat cube, 613
 hardware, 612–614
 input/output operation support, 610
 interconnection network, 613
 I/O configuration, 614 (fig.)
 lazy TLB invalidation mechanism, 611
 links, 825
 livelock, 608
 local serialization at home node, 606 (fig.)
 local serialization at requestor, 606 (fig.)
 lock performance on, 650
 memory consistency model, 607–608
 MIPS R10000 processors, 597, 684, 868
 multiprocessor block diagram, 598 (fig.)
 NACKs, 597
 network case study, 825–826
 network topology, 826 (fig.)
 node board, 613 (fig.)
 Ocean application on, 150
 performance characteristics, 618–622
 pipelined latency, 618
 processing nodes, 597
 protocol actions in response to requests, 601 (fig.)
 protocol extensions, 610–612
 protocol states, 598–599
 Raytrace application on, 179
 read-exclusive handling, 602
 read request handling, 599–601
 reply forwarding, 597, 602
 router connections, 826 (fig.)
 scaling, 621–622
 serialization to location for coherence, 604–607
 speedups on, 205 (fig.)
 SPIDER switch, 825
 starvation, 608
 synchronization support, 611–612
 SysAD bus, 609, 612, 613
 true unloaded latency, 619
 virtual channels, 613
 write atomicity, 607
 write-back request handling, 603–604, 673
 write request handling, 601–603
 Xbow, 612, 613, 614
See also NUMA-Q
 SGI Origin2000 Hub, 706
 chip, 612
 chip layout, 616 (fig.)
 connections, 612
 controller components, 615
 crossbar, 617, 618
 implementation, 614–618
 memory/directory interface (MI), 617
 network interface (NI), 618
 processor interface (PI), 615–617
See also SGI Origin2000
 shared address, 29
Fortran 90/High Performance Fortran, 52
 programming, 26
 shared address space, 28–37, 116
 abstraction support, 475
 arrays representing grid in, 147 (fig.)
 artificial communication in, 142, 146
 cache-coherent, 879–891
 without caching of shared data, 877–879
 coherent replication in, 556
 communication abstraction, 473, 474 (fig.)
 convergence, 42–44
 data transfer, 37
 equation solver pseudocode, 104–105 (fig.)
 key primitives, 102 (fig.)
 memory consistency model, 681
 mutual exclusion, 538
 naming, 55, 184
 node structures, 728 (fig.)
 operations, 56
 orchestration under, 101–108
 ordering, 57
 parallel software operations, 527–528
 physical, 52, 506–513
 point-to-point events, 538
 prefetching in, 879 (fig.)
 programming model, 37
 read performance comparison, 527 (fig.)
 read/write to shared data, 183
 replication, 185
 spatial locality in, 146–148
 threads of control, 54
 virtual, 43
 without coherent replication, 723–724
 working set, 186
See also message passing; programming models
 shared address space latency tolerance, 851–922
 block data transfer, 853–863
 cache-coherent, 879–891
 communication structure, 852
 multithreading, 896–922
 precommunication, 877–896
 proceeding past communication in same threads, 863–867
 system application, 851–852
 technique properties, 923 (fig.)
See also latency tolerance
 shared bus, 453
 shared caches, 434–437
 associativity, 436
 bandwidth requirement satisfaction, 436
 benefits of, 434–435
 cache hardware utilization, 435
 disadvantages of, 436–437
 examples, 435
 first-level, 435, 436–437
 hit latency, 436
 multiprocessor architecture, 435 (fig.)
 size of, 436
 spatial locality and, 434
 working sets and, 434
See also caches
 shared memory multiprocessors, 23, 28, 269–376
 bus interconnection, 32
 communication assist and, 51
 communication hardware, 29

- shared memory multiprocessors
(continued)
 crossbar switch, 30, 31, 34
 extending systems into, 31 (fig.)
 interconnection schemes; 31
(fig.)
 latency and, 37
 memory capacity, increasing, 29
 memory model, 30 (fig.)
 multistage interconnect, 30–31,
 34
 processes, 49
 programming model, 29
 scalable, 34
 small-scale, 66, 453
 software implications, 359–366
 synchronization, 334–358
 shared pending (SP) state, 925
 shared physical address space, 52,
 506–513
 cachability, 508
 communication assist, 506, 507
 CRAY T3D case study, 508–511
 CRAY T3E case study, 512–513
 early designs, 506
 limitations, overcoming, 732
 machine organization, 507 (fig.)
 Memory Channel, 518
 MMU, 507
 summary, 513
See also shared address space
 shared pools, 807
 shared virtual memory (SVM),
 709–724
 coherence protocol, 733
 communication, 712 (fig.)
 high-performance, protocols, 709
 illustration, 710 (fig.)
 memory management problems,
 726
 page-based, 709
 page-grained, 739
 path of read operation, 719–720
 performance implications,
 720–721
 sequential consistency for, 711
(fig.)
 software, coherence, 721
 software, layer, 716
 TreadMarks system, 716
 write notices, 711
 sharing list
 elements, 625 (fig.)
 head node, 628
 long, 629
 primitive operations, 625
 purging, 630 (fig.)
 purging latency, 629
 writer in, 629
 sharing write back, 600
 Shasta, 745
 short links, 764
 SHRIMP, 718
 Sigma-1, 494
 Simple COMA, 681, 726–728
 design, 726
 drawbacks, 727
 path of read reference, 727–728
 performance trade-offs, 726–727
 presence check, 726
See also cache-only memory
 architecture (COMA)
 simulated time, 233
 simulations
 event-driven, 233, 234 (fig.)
 expense of, 232
 limitations of, 200, 232
 machine parameters for, 234–237
 multiprocessor, 200, 233–234
 speeding up, 234
 trace-driven, 233
 simultaneous multithreading,
 920–922
 illustrated, 921 (fig.)
 issues, 921–922
 for uniprocessors, 922
See also multithreading
 single-instruction-multiple-data
 (SIMD), 44, 103
 data parallel programming model,
 44
 evolution, 46–47
 organization, 45 (fig.)
 renaissance, 46
 single-instruction-single-data (SISD),
 44
 single-level caches, 281
 with atomic bus, 380–393
 atomic operation implementa-
 tion, 391–393
 base organization, 385
 cache controller design, 381–382
 deadlock, 390
 livelock, 390
 nonatomic state transitions,
 385–388
 serialization, 388–390
 snoop results, 382–384
 starvation, 390–391
 tag design, 381–382
 write backs, 384–385
See also caches
 single-program-multiple-data
 (SPMD), 46–47, 103
 single-system image, 513
 single writer protocol
 with consistency at acquire,
 734–737
 with consistency at release,
 733–734
 recent copy invalidation, 735
(fig.)
 slackness, 155
 slotted rings, 443
 snoop-based multiprocessors
 cache-coherent, 278 (fig.)
 design, 377–452
 snooping
 buses, 277–283, 589
 cache controllers, 277
 hierarchical, 559, 660–664
 latency, 383
 results, matching, 402–403
 results, reporting, 382–384
 snooping caches
 base machine design, 386 (fig.)
 organization of, 383
 two-level, organization, 398 (fig.)
See also caches
 snooping protocols, 280
 cache-coherent, 272
 components, 280
 design space for, 291–305
 supporting, 381
 software caches, 186
 software combining trees, 542–543
 flat arrive structure vs., 543 (fig.)
 for release, 543
 software-controlled prefetching, 880
 coverage, 889
 effectiveness, 889, 895
 hardware-controlled vs., 888–890
 scheduling, 884
 with single processor, 884–887
 unnecessary prefetch reduction,
 889
See also prefetching
 software instrumentation, 707–708
 protocol processing, 708
 run-time cost, 707
 of write operations, 718
 software locks, 338–340
 implementation, 339
 problem, 338–339
 state storage, 338
See also locks
 software overflow scheme, 657–658
 for limited pointer directories,
 658
 overhead, 657–658
 software queuing locks, 539
 algorithm for, 541 (fig.)
 compare&swap operation, 540

- space per lock, 541
 states, 540 (fig.)
See also locks; software locks
- Solaris UNIX, 416
 source-based routing, 790, 805
 space-sharing, 89–89
 sparse directory, 659
 spatial interleaving, 360–362
 spatial locality, 142, 319–320
 analyzing, 885
 Barnes-Hut application, 170–172, 322
 in capacity misses, 324
 centralized memory and, 360
 in cold misses, 324
 Data Mining application, 182
 in equation solver kernel, 149 (fig.)
 exploiting, 145–150
 increasing by copying data, 363–364
 interaction example, 225
 linked lists and, 324
 LU, 320, 322
 mismatches, 146
 Multiprog, 323–324
 Ocean application, 163–164, 320, 321
 poor, 145
 problem size impact on, 225 (fig.)
 of processes' access patterns, 148
 Radiosity application, 322
 Raytrace application, 176–177
 scaling models, 213
 in shared address space, 146–148
 shared caches and, 434
 using, 224–226
See also locality
- SPEC benchmark, 13, 199, 968
 SPEC95, 199
 SPEC92, 199
See also benchmarks
- speculative execution, 684, 870–871, 876
 instructions, 870–871
 lookahead buffer, 870
 Radix, 875 (fig.)
- speculative reads, 684, 872, 876
- speculative reply, 600
- speedup, 122, 203
 algorithmic, 220, 254, 256 (fig.)
 application, 430–431
 Barnes-Hut, 431
 blocked multithreading, 912 (fig.)
 CRAY T3D, 532
 IBM SP-2, 532
 interleaved multithreading, 912 (fig.)
 limit, 135, 136
 LU, 431
- MC scaling, 211, 213
 measuring, 163, 203, 229
 Ocean, 431
 on PRAM architectural model, 254
 on SGI Origin2000, 205 (fig.), 620–621
 PC scaling, 208, 213
 perfect, 266
 Radiosity, 431
 Radix, 431
 Raytrace, 431
 scaled, 210
 self-relative, 163
 superlinear, 204, 205
 TC scaling, 209, 213
 UltraSparc cluster (NOW), 532
 over uniprocessor, 204
- SPIDER switch, 825
 spin-waiting, 106–107
- SPLASH-2 suite, 307, 965–966
- Split-C language, 724
- split-transaction bus, 398–415
 advantages/disadvantages, 398
 alternative design choices, 409–410
 cache miss path, 404–406
 example design, 400
 flow control, 404
 incoming transactions, 408
 issues, 398–399
 lock-down, 391
 with multilevel caches, 410–413
 multiple outstanding miss support, 413–415
 negative acknowledgment (NACK), 400
 read transaction, 401 (fig.)
 request-response matching, 398, 400–402
 request transaction, 398
 response transaction, 398
 sequential consistency, 406–409
 serialization, 406–409
 snoop results, 402–403
 SparcCenter 2000, 410
- SRAM, 949, 950
- stable states, 387
- Stache, 726–728
 allocation management, 727
 memory management, 727
- stacked dimension switches, 810–811
 design, 810
 illustrated, 811 (fig.)
See also switches
- Stanford DASH, 640, 910
 deadlock detection, 594
 directory-based coherence, 596
- multiprocessor, 301
 project, 69
- Stanford FLASH, 640, 648, 658
 coherence controller, 731
 programmable protocol engine, 706
- starvation, 380, 390–391
 directory protocols, 595–596
 elimination of, 380, 390
 NUMA-Q, 633
 potential, 380
 SGI Origin2000, 608
 solution, 595–596
See also deadlock; starvation
- state bits, 560
- states
 busy, 603–604
 cache block, 279
 directory, 603–604
 dirty, 600
 exclusive, 603
 exclusive-clean (E), 302, 600
 exclusive pending (EP), 925
 expanding number of, 387
 invalid pending (IP), 925
 master, 704
 modified (M), 293, 302
 NUMA-Q, 624–626
 shared, 293
 shared-clean (Sc), 302
 shared-modified (Sm), 302
 share pending (SP), 925
 stable, 387
 transient, 388 (fig.)
- state transition diagrams
 cache block, 279–280
 Dragon protocol, 303 (fig.)
 MESI protocol, 301 (fig.), 388 (fig.)
- Powerpath-2 bus, 418 (fig.)
- state transitions, 308–309
 applications with smaller caches, 314
- bus actions corresponding to, 311
 cache block, 367
 Dragon update protocol, 303–304
 frequency data, 308–309
 MESI invalidation protocol, 299–300
- MSI invalidation protocol, 294–296
 nonatomic, 385–388
- static assignment, 88, 99
 dynamic assignment vs., 126–129
 load balance and, 126
 task granularity with, 130
 task management and, 126
 task size and, 130
See also assignment

- static ordering, 445
 steady-state loop, 850
 STOP symbol, 815
 storage technology revolution, 67
 store-and-forward, 40, 460
 store-and-forward routing, 756, 758
 cut-through routing vs., 757 (fig.)
 deadlock, 792
 delay, 942
 distance, 779
 store-conditional, 344, 652
 stream buffers, 882
 subblock write bits (SWBs), 925–926
 subgrids, 133, 148
 Sun Enterprise 1000, 445–446
 Sun Enterprise Server, 35 (fig.), 384,
 404
 Sun Enterprise 6000, 415–416,
 424–429
 block diagram, 417 (fig.)
 design, 415
 D-tags, 427, 429
 FiberChannel modules, 429
 Gigaplane system bus, 424–427
 I/O subsystem, 429
 memory, 416
 memory subsystem, 427–429
 memory system performance, 429
 operating system, 416
 processing and I/O board
 organization, 428 (fig.)
 processor subsystem, 427–429
 read microbenchmark results,
 430 (fig.)
 SysIO ASICs, 429
 See also SGI Challenge
 Sun Sparc
 MEMBAR instructions, 693
 PSO model, 689
 V8, 689
 V9 RMO, 689, 690, 693
 Sun SparcCenter 2000, 331
 multiple bus approach, 445
 split-transaction buses, 410
 Sun UltraSparc, 533, 868, 940
 supercomputers, 21–23
 CRAY vector, 8, 22
 performance, 24 (fig.)
 uniprocessor performance of, 22
 (fig.)
 vector processors, 21–22, 946
 superlinear speedup, 204, 205
 superscalar execution, 17
 superscalar processors, 895
 surface-area-to-volume ratio, 132
 swap instruction, 339
 switches, 457, 767–768
 bandwidth, 939
 buffering, 759
 context, 897, 901
 cost, 768, 900
 degree, 457, 752, 767, 768, 801
 design of, 801–811
 input buffered, 805 (fig.)
 input ports, 767, 802
 internal buffering, 768
 internal datapath, 802–804
 “intranode,” 457–458
 Myricom network, 827
 network, 457
 organization, 767 (fig.)
 output ports, 767, 802
 output scheduling, 808–810
 routing operations at, 789
 scale limitation, 457
 SPIDER, 825
 stacked dimension, 810–811
 VLSI, 768, 802, 804
 switching
 circuit, 756–757
 cost, 898
 delay, 756
 packet, 757
 strategy, 753
 time, 897
 symmetric multiprocessors (SMPs),
 32–34, 269–271
 building blocks, 515
 bus-based, 271, 457
 design challenge, 366–367
 inexpensive, 949
 layers of abstraction, 270 (fig.)
 nodes, 467, 503, 721
 symmetric successive overrelaxation
 (SSOR), 532
 Synapse multiprocessor, 298
 synchronization, 334–359
 algorithms for barriers, 542–547
 algorithms for locks, 538–541
 Barnes-Hut application, 172–173
 barrier, 252
 block data transfer and, 857
 Data Mining application, 182
 directory-based cache coherence,
 648–652
 directory-based multiprocessors,
 556
 event, 57, 103, 106, 283
 execution time component, 158
 explicit, 285 (fig.)
 fine-grained, 130
 frequency of, 95
 global, 95, 96, 106
 interprocess, 118
 library design, 336
 lock-free, 351
 MC scaling, 213
 message-passing program, 111,
 113
 microbenchmarks, 216
 multiple-producer, single-
 consumer group, 173
 mutual exclusion, 57, 103, 113,
 124, 188, 337–351
 nonblocking, 351
 Ocean application, 165
 operation order, 714 (fig.)
 PC scaling, 213
 point-to-point, 96, 117, 172
 QOLB, 626, 651
 Raytrace application, 177
 scalable multiprocessor, 538–547,
 655
 SGI Origin2000 support,
 611–612
 for shared data variable access,
 691 (fig.)
 shared memory multiprocessor,
 334–359
 software algorithms, 336
 summary, 358
 support, 57
 TC scaling, 213
 wait-free, 350
 wait time, 124, 866
 workload, 254
 synchronized programs, 695
 at programmer’s interface, 699
 yielding, 697
 synchronous links, 765
 synchronous message passing, 39
 matching rule, 478
 protocol, 478 (fig.)
 See also message passing
 system area networks (SANs), 467,
 750
 Myrinet, 516–518
 scalable high-performance, 503
 system design trends, 19–21
 system-level integration, 466–467
 system specification, 685–686, 686,
 690, 693
 Alpha, 693
 characteristics, 694 (fig.)
 PC, 686, 687, 688 (fig.)
 PowerPC, 693
 PSO, 686, 689
 RC, 687, 690, 691–692
 TSO, 686, 687, 688 (fig.), 689
 WO, 686–687, 690–691
 See also relaxed memory
 consistency models
 systolic architectures, 49–50
 computation of inner product, 50
 (fig.)
 PEs, 49
 solutions on generic machines, 50
 See also parallel architectures

T

*T machine, 495
 table-driven routing, 790
 tail node, 624
 Tandem Himalaya system, 12
 task granularity, 129–131
 determining, 129–131
 with dynamic task queuing, 129–130
 fineness, 130
 with static assignment, 130
 task pools, 127, 128
 task queues, 128, 129
 distributed, 192
 implementing, 130–131
 locking, 131
 remote, accessing, 131
 tasks, 82–83
 assignment of, 83, 116
 coarse-grained, 83
 examples, 82–83
 fine-grained, 83
 relationships of, 84 (fig.)
 task stealing, 128
 dynamic, 134
 implementing, 128
 termination detection and, 195
 TC (time-constrained) scaling, 207, 431, 433
 communication-to-computation ratio, 212, 433
 concurrency, 212
 memory requirements, 212
 naive, 433
 problem size, 208
 realistic, 433
 spatial locality, 213
 speedup, 209, 213
 synchronization, 213
 temporal locality, 213
 viability, 211
 See also scaling; scaling models
 technology trends, 12–14
 temporal locality, 142, 143–145
 analyzing, 885
 Barnes-Hut application, 172
 Data Mining application, 182
 in equation solver kernel, 143–144
 exploiting, 143–145, 359
 goal, 359
 implications, 145
 Ocean application, 164
 Raytrace application, 177
 scaling models, 213
 techniques, 145
 See also locality
 Tera architecture, 906–908
 active thread support, 907

general purpose multiprocessing, 908
 minimum issue delay, 909
 See also interleaved multi-threading
 termination detection, 128, 195
 tertiary caches, 700–701
 test&set instruction, 339, 341, 344, 391
 implementing, 391
 success determination, 339
 test&set locks
 with backoff, 342
 performance, 340, 341 (fig.)
 problem with, 341
 test-and-test&set locks, 342–343
 failure, 344
 latency, 343
 TFLOPS (one trillion floating-point operations per second), 502
 Thinking Machines. *See* CM-1; CM-2; CM-5
 thread-level parallelism, 17–19
 threads, 29, 53, 83
 active, 898, 907
 busy time, 897
 context, 897
 idle time, 898
 increasing number of, 899
 lightweight, 136
 multiple concurrent, 19
 ready, 898
 in RISC machines, 53
 shared address space
 programming model, 54
 switching arrangement, 850, 851
 switching time, 897
 unready, 909–910
 See also multithreading
 3D cubes, 769, 770 (fig.)
 three-message miss, 586
 three-state invalidation protocol, 293–299
 ticket locks, 346–347
 acquire method, 346–347
 performance, 350
 read traffic problem, 347
 See also locks
 tiles, 176
 token-passing rings, 442–443
 topologies. *See* network topologies
 topology-oriented program design, 153
 torus
 illustrated, 770 (fig.)
 links per node, 775
 routing chip, 810
 3D, 771
 See also meshes
 total store ordering (TSO), 686, 865, 866, 867
 comparison, 688 (fig.)
 write atomicity, 689
 write ordering preservation, 687
 See also system specification
 TPC. *See* Transaction Processing Council (TPC)
 trace-driven simulation, 233
 trade-offs, 122, 123
 architectural, 531
 block data transfer, 854–856, 859
 busy-waiting and blocking, 335
 cost-performance, 2, 4, 15
 emergence of, 148
 evaluating, 199, 231–243
 extra work, load balance,
 communication, 136
 hardware/software, 679–747
 identifying, 199
 latency, 784
 network design, 749–750
 partitioning, 135
 protocol design, 305–334
 realistic applications for, 123
 switch-to-switch layer, 827
 trailer, 754, 766
 Transaction Processing Council (TPC), 10
 data, 10, 11
 March 1996 report, 12
 result reporting, 964
 TPC-A, 964
 TPC-B, 642, 643, 964
 TPC-C, 10, 964–965
 TPC-D, 642, 643, 965
 See also benchmarks
 transient states, 388 (fig.)
 transistors, per processor chip, 16
 translation lookaside buffer (TLB), 67
 ASIDs, 440
 coherence, 439–441
 control registers, 915
 entries, 440, 441
 flush notices, 451
 handler, 429
 hardware-loaded, 440
 lazy, invalidation mechanism, 611
 misses, 223, 429
 PTEs, 439
 shootdown, 440, 451, 452 (fig.)
 software-loaded, 440
 translation mechanism, 686, 698
 TreadMarks SVM system, 716
 tree barriers
 arrival, 543 (fig.)
 combining, with sense removal, 543, 544 (fig.)
 with local spinning, 543–545

tree barriers (*continued*)
 release, 543
 static binary, 544
 traffic distribution, 543

trees, 772–774
 binary, 772, 773 (fig.)
 bisection, 774
 branch factor, 773
 fat, 774, 776
 global, 957
 links per node, 775
 locality, 668
 saturation, 760
 16-node, 774
 wire problem, 773

trends
 application, 6–12
 architectural, 14–21
 microprocessor design, 15–19
 system design, 19–21
 technology, 12–14
 VLSI technology, 938
TruCluster Memory Channel
 software, 520
true-sharing misses, 315.
 block size and, 318
 reducing, 316
turn-model routing, 797–799
 minimal, 798 (fig.)
 restrictions, 798 (fig.)
 virtual channels, 799
See also routing
twins, 716
2D grids, 769, 770 (fig.)
two-level sharing hierarchy, 587–589
 advantages, 587
 locality and, 588

U

uniform cluster cache (UCC) card, 663
uniform interconnection card (UIC), 663
uniprocessors
 bandwidth, 939
 cache controller, 381–382
 cache design in, 314, 381
 compilers, 289
 memory system, 275
 PC shipments, 936
 simultaneous multithreading for, 922
 speedup over, 204
 state diagram, 280
 supercomputer performance, 22 (fig.)
 write-back caches on, 292

unloaded latency, 755, 780–785
 with equal pin count, 785 (fig.)
 as function of degree, 781 (fig.)
 for k-ary d-cubes, 783 (fig.)
 for n-byte packet, 756
 scaling, 780 (fig.)
 for short messages, 780
See also latency

unnecessary prefetches, 881
 minimizing, 884
 reducing, 889
See also prefetching
up*-down* routing, 796–797, 799
update-based protocols, 278, 292
 bounding losses of, 331
Dragon (four-state), 301–305, 374
Firefly, 374
invalidation-based protocols
 combined with, 330–332
invalidation-based protocols vs., 329–330
miss rates, 332 (fig.)
upgrade/update rates for, 333 (fig.)
write atomicity, 673
write operation and, 292
See also protocols

upgrades, 318
user flags, 491
user-level access, 491–496
 case study, 493–494
 CM-5, 493–494
 node-to-network interface, 491–493
user-level handlers, 494–496
 communication assist, 495 (fig.)
 message processing, 493
user-level network port, 491
 architecture, 492 (fig.)
 communication assist for, 492 (fig.)

user-oriented properties, 207
utilization metric, 230, 262

V

vector processors, 21–22, 67
vector time stamp, 736
virtual channels, 613, 795–796
 in basic switch, 795 (fig.)
 breaking deadlock cycles with, 796 (fig.)
buffering, 807–808
routing algorithm and, 796
support, 807
turn-model routing with, 799
uses for, 795
See also channels

virtual circuits, 514
virtually indexed caches, 437–439
 organization, 439 (fig.)
 processor and, 438
 virtual index, 438
visualization case study. *See Raytrace application*
VLSI
 CMOS devices, 944
 feature size, 12
 generation, 15, 63
 scaling, 802
 switches, 768, 802, 804
 technology trends, 938
von Neumann model, 189

W

wait-free synchronization, 351
waiting algorithm, 335, 336
wall-clock time, 228–229
WAN flow control, 812–813
weak ordering (WO), 686–687, 690–691, 699, 866, 867

motivation, 690
at programmer's interface, 699
release consistency vs., 691 (fig.)

See also system specification
west-first algorithm, 797–798, 799 (fig.)

wide links, 764–765

working sets
 Barnes-Hut application, 172
 curves, 240, 260 (fig.)
 effect of, 227 (fig.)
 execution characteristics and, 222
 growth rates, 261
 LU benchmark curves, 537 (fig.)
 MC scaling, 227
 nonlocal data, 239
 Ocean application, 164
 PC scaling, 227
 problem sizes based on, 224 (fig.)
 Raytrace application, 177
 shared address space, 186
 shared caches and, 434
 sizes, 259–261

workload case studies, 244–253

LU, 244–248
 Multiprog, 244, 252
 Radiosity, 244, 249–252
 Radix, 244, 248–249, 267

workload-driven evaluation, 199–267
 of architectural idea, 231–243
 difficulty of, 200
 of fixed-size machine, 221–226
 performance metrics, choosing, 228–231

- protocol trade-offs, 332–334
 - for real machine, 215–231
 - of trade-off, 231–243
 - varying machine size and, 226–228
- workload parameters
 - relationships among, 214
 - scaling, 201, 213–214
- workloads
 - benchmark, 201
 - characteristics of, 253–261
 - choosing, 216–220, 238
 - communication-to-computation ratio, 257–259
 - concurrency, 220, 254–257
 - coverage of behavioral properties, 219–220
 - data access, 254
 - load balance, 254–257
 - with low/high communication-to-computation ratios, 219
 - multiprogrammed, 218
 - representation of application domains, 218–219
 - synchronization, 254
 - working set sizes, 259–261
- work metric, 209
- workstations
 - networks of (NOWs), 513–521
 - performance, 71 (fig.)
- wormhole routing, 759, 794, 808
- write atomicity, 288, 389
 - appearance of, 593
 - distributed interconnect and, 592
 - example, 289 (fig.)
 - importance, 288
 - in invalidation-based scheme, 592–593
 - preservation, 291
 - SGI Origin2000, 607
 - TSO, 689
 - with update protocols, 673
 - violation in scalable system, 593 (fig.)

See also sequential consistency
- write-back buffers, 385
- write-back caches, 274, 291
 - buffer deadlock problem in, 412
 - invalidation-based protocol, 293–299, 391
 - L₁, 396
 - on uniprocessors, 292
 - space of protocols for, 283
 - update-based protocol, 301–305

See also caches
- write backs, 384–385
 - NUMA-Q handling of, 630–632
- SGI Origin2000 handling of, 603–604, 673
- sharing, 600
- write buffers, 413
 - bypassable, 867
 - merge restriction into, 865
 - SC utilization of, 865, 874
 - See also* buffers
- write-fence operation, 741
- write memory barrier (WMB), 693
- write misses
 - hierarchy flow, 666
 - performance impact, 865–868
 - proceeding past, 864–868
- write-no-allocate caches, 281 (fig.)
- write notices, 711, 734
 - acquirer retention of, 735
 - at every release, 733
 - in lazy implementation, 717
 - propagation of, 734
 - in release-based protocol, 737
- write propagation, 277
- write request buffers (WRBs), 615
- write requests
 - NUMA-Q, 628–630
 - SGI Origin2000, 601–603
- write serialization, 277, 289, 663
 - extending, 288
 - providing, 389
- write sharing, 359
- write-through caches, 278, 279, 291
 - invalidation-based protocol for, 280, 283 (fig.)
 - L₁, 396
 - multiprocessor snoopy coherence, 281 (fig.)
- write-to-read program order, 687–689
- write-to-write program orders, 689

X-Z

Xbow, 612, 613, 614

PARALLEL COMPUTER ARCHITECTURE

A Hardware/Software Approach

**David E. Culler, University of California, Berkeley and Jaswinder Pal Singh, Princeton University,
with Anoop Gupta, Stanford University**

**From the foreword by John L. Hennessy, Frederick Emmons Terman
Dean of Engineering, Stanford University**

The insightful approach taken by the authors combined with a systematic and quantitative examination of different architectures distinguishes this book from all previous books on parallel architecture. This approach has three major innovations: it builds on the recent convergence of parallel architectures, it uses applications as a driver for evaluating and analyzing architectures, and it is grounded in a solid methodology for performance evaluation....This is an exciting and dynamic new exploration of the multiprocessor design space.

The Definitive Work on Parallel Computer Architecture

The most exciting development in parallel computer architecture is the convergence of traditionally disparate approaches on a common machine structure. This book explains the forces behind this convergence of shared memory, message-passing, data parallel, and data-driven computing architectures. It then examines the design issues that are critical to parallel architecture across the full range of modern designs, covering data access, communication performance, coordination of cooperative work, and correct implementation of useful semantics. It not only describes the hardware and software techniques for addressing each of these issues but also explores how these techniques interact in the same system.

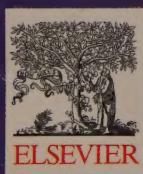
Features:

- Synthesizes a decade of research and development for practicing engineers, graduate students, and researchers in parallel computer architecture, system software, and applications development.
- Presents in-depth application case studies from computer graphics, computational science and engineering, and data mining to demonstrate quantitative evaluation of design trade-offs.
- Illustrates bus-based and network-based parallel systems with case studies of more than a dozen important commercial designs.

About the Authors

Each of the authors has led important research projects that have directly influenced the development and use of today's commercial parallel machines. Jaswinder Pal Singh led the development of the SPLASH and SPLASH-2 suites of parallel programs, which have defined the workloads and methodology used to drive decisions and evaluate trade-offs in shared memory parallel architecture. David Culler led the Berkeley Network of Workstations (NOW) projects, which sparked the current commercial revolution in high-performance clusters. Anoop Gupta co-led the Stanford DASH multiprocessor project, which developed the shared memory technology increasingly used in commercial machines.

This edition has been authorized by Elsevier for sale in the following countries:
India, Pakistan, Nepal, Sri Lanka and Bangladesh. Sale and purchase of this book
outside these countries is not authorized and is illegal.



MORGAN KAUFMANN PUBLISHERS

AN IMPRINT OF ELSEVIER

www.mkp.com

ISBN: 978-81-8147-189-5

