ADVANCED COMPUTER ARCHITECTURE

Kai Hwang

PARALLELISM

SCALABILITY

PROGRAMMABILITY

For sale in India, Nepal, Bangladesh, Sri Lanka and Bhutan Only

TATA McGRAW-HILL EDITION

Tata McGraw-Hill

ADVANCED COMPUTER ARCHITECTURE
Parallelism, Scalability, Programmability

Copyright © 2000 by The McGraw-Hill Companies, Inc.,
All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written permission of publisher, with the exception that the program listings may be entered, stored, and executed in a computer system, but they may not be reproduced for publication

Tata McGraw-Hill Edition 2001

Eighteenth reprint 2008
RACYCDRXRBZQD

Reprinted in India by arrangement with The McGraw-Hill Companies Inc.,
New York

Sales territories: India, Pakistan, Nepal, Bangladesh, Sri Lanka and Bhutan

Library of Congress Cataloging-in-Publication Data

Hwang, Kai, 1943 –
Advanced Computer Architecture: Parallelism, Scalability, Programmability/
Kai Hwang,
p. cm. – (McGraw-Hill computer science series. Computer organization and architecture. Network, parallel and distributed computing. McGraw-Hill computer engineering series)
Includes bibiiographical references (p.) and index.
ISBN 0-07-031622-8
1. Computer Architecture I. Title II Series
QA76.9A7H87 1993
004'35-dc20 92-44944

ISBN-13: 978-0-07-053070-6
ISBN-10: 0-07-053070-X

Published by Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008, and printed at
Gopaljee Enterprises, Delhi 110 053

The McGraw-Hill Companies

Copyrighted material

Contents

Foreword xvii

Preface xix

PART I THEORY OF PARALLELISM 1

Chapter 1 Parallel Computer Models 3

1.1 The State of Computing 3
1.1.1 Computer Development Milestones 3
1.1.2 Elements of Modern Computers 6
1.1.3 Evolution of Computer Architecture 9
1.1.4 System Attributes to Performance 14

1.2 Multiprocessors and Multicomputers 19
1.2.1 Shared-Memory Multiprocessors 19
1.2.2 Distributed-Memory Multicomputers 24
1.2.3 A Taxonomy of MIMD Computers 27

1.3 Multivector and SIMD Computers 27
1.3.1 Vector Supercomputers 27
1.3.2 SIMD Supercomputers 30

1.4 PRAM and VLSI Models 32
1.4.1 Parallel Random-Access Machines 33
1.4.2 VLSI Complexity Model 38

1.5 Architectural Development Tracks 41
1.5.1 Multiple-Processor Tracks 41
1.5.2 Multivector and SIMD Tracks 43
1.5.3 Multithreaded and Dataflow Tracks 44

1.6 Bibliographic Notes and Exercises 45

Copyrighted material

Chapter 2 Program and Network Properties 51
2.1 Conditions of Parallelism 51
2.1.1 Data and Resource Dependences 51
2.1.2 Hardware and Software Parallelism 57
2.1.3 The Role of Compilers 60
2.2 Program Partitioning and Scheduling 61
2.2.1 Grain Sizes and Latency 61
2.2.2 Grain Packing and Scheduling 64
2.2.3 Static Multiprocessor Scheduling 67
2.3 Program Flow Mechanisms 70
2.3.1 Control Flow Versus Data Flow 71
2.3.2 Demand-Driven Mechanisms 74
2.3.3 Comparison of Flow Mechanisms 75
2.4 System Interconnect Architectures 76
2.4.1 Network Properties and Routing 77
2.4.2 Static Connection Networks 80
2.4.3 Dynamic Connection Networks 89
2.5 Bibliographic Notes and Exercises 96

Chapter 3 Principles of Scalable Performance 105
3.1 Performance Metrics and Measures 105
3.1.1 Parallelism Profile in Programs 105
3.1.2 Harmonic Mean Performance 108
3.1.3 Efficiency, Utilization, and Quality 112
3.1.4 Standard Performance Measures 115
3.2 Parallel Processing Applications 118
3.2.1 Massive Parallelism for Grand Challenges 118
3.2.2 Application Models of Parallel Computers 122
3.2.3 Scalability of Parallel Algorithms 125
3.3 Speedup Performance Laws 129
3.3.1 Amdahl's Law for a Fixed Workload 129
3.3.2 Gustafson's Law for Scaled Problems 131
3.3.3 Memory-Bounded Speedup Model 134
3.4 Scalability Analysis and Approaches 138
3.4.1 Scalability Metrics and Goals 138
3.4.2 Evolution of Scalable Computers 143
3.4.3 Research Issues and Solutions 147
3.5 Bibliographic Notes and Exercises 149

PART II HARDWARE TECHNOLOGIES 155

Copyrighted material

Chapter 4 Processors and Memory Hierarchy **157**

4.1 Advanced Processor Technology 157
- 4.1.1 Design Space of Processors 157
- 4.1.2 Instruction-Set Architectures 162
- 4.1.3 CISC Scalar Processors 165
- 4.1.4 RISC Scalar Processors 169

4.2 Superscalar and Vector Processors 177
- 4.2.1 Superscalar Processors 178
- 4.2.2 The VLIW Architecture 182
- 4.2.3 Vector and Symbolic Processors 184

4.3 Memory Hierarchy Technology 188
- 4.3.1 Hierarchical Memory Technology 188
- 4.3.2 Inclusion, Coherence, and Locality 190
- 4.3.3 Memory Capacity Planning 194

4.4 Virtual Memory Technology 196
- 4.4.1 Virtual Memory Models 196
- 4.4.2 TLB, Paging, and Segmentation 198
- 4.4.3 Memory Replacement Policies 205

4.5 Bibliographic Notes and Exercises 208

Chapter 5 Bus, Cache, and Shared Memory **213**

5.1 Backplane Bus Systems 213
- 5.1.1 Backplane Bus Specification 213
- 5.1.2 Addressing and Timing Protocols 216
- 5.1.3 Arbitration, Transaction, and Interrupt 218
- 5.1.4 The IEEE Futurebus+ Standards 221

5.2 Cache Memory Organizations 224
- 5.2.1 Cache Addressing Models 225
- 5.2.2 Direct Mapping and Associative Caches 228
- 5.2.3 Set-Associative and Sector Caches 232
- 5.2.4 Cache Performance Issues 236

5.3 Shared-Memory Organizations 238
- 5.3.1 Interleaved Memory Organization 239
- 5.3.2 Bandwidth and Fault Tolerance 242
- 5.3.3 Memory Allocation Schemes 244

5.4 Sequential and Weak Consistency Models 248
- 5.4.1 Atomicity and Event Ordering 248
- 5.4.2 Sequential Consistency Model 252
- 5.4.3 Weak Consistency Models 253

5.5 Bibliographic Notes and Exercises 256

Chapter 6 Pipelining and Superscalar Techniques **265**

Copyrighted material

6.1 Linear Pipeline Processors 265
6.1.1 Asynchronous and Synchronous Models 265
6.1.2 Clocking and Timing Control 267
6.1.3 Speedup, Efficiency, and Throughput 268
6.2 Nonlinear Pipeline Processors 270
6.2.1 Reservation and Latency Analysis 270
6.2.2 Collision-Free Scheduling 274
6.2.3 Pipeline Schedule Optimization 276
6.3 Instruction Pipeline Design 280
6.3.1 Instruction Execution Phases 280
6.3.2 Mechanisms for Instruction Pipelining 283
6.3.3 Dynamic Instruction Scheduling 288
6.3.4 Branch Handling Techniques 291
6.4 Arithmetic Pipeline Design 297
6.4.1 Computer Arithmetic Principles 297
6.4.2 Static Arithmetic Pipelines 299
6.4.3 Multifunctional Arithmetic Pipelines 307
6.5 Superscalar and Superpipeline Design 308
6.5.1 Superscalar Pipeline Design 310
6.5.2 Superpipelined Design 316
6.5.3 Supersymmetry and Design Tradeoffs 320
6.6 Bibliographic Notes and Exercises 322

PART III PARALLEL AND SCALABLE ARCHITECTURES 329

Chapter 7 Multiprocessors and Multicomputers 331
7.1 Multiprocessor System Interconnects 331
7.1.1 Hierarchical Bus Systems 333
7.1.2 Crossbar Switch and Multiport Memory 336
7.1.3 Multistage and Combining Networks 341
7.2 Cache Coherence and Synchronization Mechanisms 348
7.2.1 The Cache Coherence Problem 348
7.2.2 Snoopy Bus Protocols 351
7.2.3 Directory-Based Protocols 358
7.2.4 Hardware Synchronization Mechanisms 364
7.3 Three Generations of Multicomputers 368
7.3.1 Design Choices in the Past 368
7.3.2 Present and Future Development 370
7.3.3 The Intel Paragon System 372
7.4 Message-Passing Mechanisms 375
7.4.1 Message-Routing Schemes 375

Copyrighted material

7.4.2 Deadlock and Virtual Channels ... 379
7.4.3 Flow Control Strategies ... 383
7.4.4 Multicast Routing Algorithms ... 387
7.5 Bibliographic Notes and Exercises ... 393

Chapter 8 Multivector and SIMD Computers ... 403
8.1 Vector Processing Principles ... 403
8.1.1 Vector Instruction Types ... 403
8.1.2 Vector-Access Memory Schemes ... 408
8.1.3 Past and Present Supercomputers ... 410
8.2 Multivector Multiprocessors ... 415
8.2.1 Performance-Directed Design Rules ... 415
8.2.2 Cray Y-MP, C-90, and MPP ... 419
8.2.3 Fujitsu VP2000 and VPP500 ... 425
8.2.4 Mainframes and Minisupercomputers ... 429
8.3 Compound Vector Processing ... 435
8.3.1 Compound Vector Operations ... 436
8.3.2 Vector Loops and Chaining ... 437
8.3.3 Multipipeline Networking ... 442
8.4 SIMD Computer Organizations ... 447
8.4.1 Implementation Models ... 447
8.4.2 The CM-2 Architecture ... 449
8.4.3 The MasPar MP-1 Architecture ... 453
8.5 The Connection Machine CM-5 ... 457
8.5.1 A Synchronized MIMD Machine ... 457
8.5.2 The CM-5 Network Architecture ... 460
8.5.3 Control Processors and Processing Nodes ... 462
8.5.4 Interprocessor Communications ... 465
8.6 Bibliographic Notes and Exercises ... 468

Chapter 9 Scalable, Multithreaded, and Dataflow Architectures ... 475
9.1 Latency-Hiding Techniques ... 475
9.1.1 Shared Virtual Memory ... 476
9.1.2 Prefetching Techniques ... 480
9.1.3 Distributed Coherent Caches ... 482
9.1.4 Scalable Coherence Interface ... 483
9.1.5 Relaxed Memory Consistency ... 486
9.2 Principles of Multithreading ... 490
9.2.1 Multithreading Issues and Solutions ... 490
9.2.2 Multiple-Context Processors ... 495
9.2.3 Multidimensional Architectures ... 499
9.3 Fine-Grain Multicomputers ... 504

Copyrighted material

9.3.1 Fine-Grain Parallelism 505
9.3.2 The MIT J-Machine 506
9.3.3 The Caltech Mosaic C 514
9.4 Scalable and Multithreaded Architectures 516
9.4.1 The Stanford Dash Multiprocessor 516
9.4.2 The Kendall Square Research KSR-1 521
9.4.3 The Tera Multiprocessor System 524
9.5 Dataflow and Hybrid Architectures 531
9.5.1 The Evolution of Dataflow Computers 531
9.5.2 The ETL/EM-4 in Japan 534
9.5.3 The MIT/Motorola *T Prototype 536
9.6 Bibliographic Notes and Exercises 539

PART IV SOFTWARE FOR PARALLEL PROGRAMMING 545

Chapter 10 Parallel Models, Languages, and Compilers 547
10.1 Parallel Programming Models 547
10.1.1 Shared-Variable Model 547
10.1.2 Message-Passing Model 551
10.1.3 Data-Parallel Model 554
10.1.4 Object-Oriented Model 556
10.1.5 Functional and Logic Models 559
10.2 Parallel Languages and Compilers 560
10.2.1 Language Features for Parallelism 560
10.2.2 Parallel Language Constructs 562
10.2.3 Optimizing Compilers for Parallelism 564
10.3 Dependence Analysis of Data Arrays 567
10.3.1 Iteration Space and Dependence Analysis 567
10.3.2 Subscript Separability and Partitioning 570
10.3.3 Categorized Dependence Tests 573
10.4 Code Optimization and Scheduling 578
10.4.1 Scalar Optimization with Basic Blocks 578
10.4.2 Local and Global Optimizations 581
10.4.3 Vectorization and Parallelization Methods 585
10.4.4 Code Generation and Scheduling 592
10.4.5 Trace Scheduling Compilation 596
10.5 Loop Parallelization and Pipelining 599
10.5.1 Loop Transformation Theory 599
10.5.2 Parallelization and Wavefronting 602
10.5.3 Tiling and Localization 605
10.5.4 Software Pipelining 610

Copyrighted material

10.6 Bibliographic Notes and Exercises 612

Chapter 11 Parallel Program Development and Environments 617

11.1 Parallel Programming Environments 617
11.1.1 Software Tools and Environments 617
11.1.2 Y-MP, Paragon, and CM-5 Environments 621
11.1.3 Visualization and Performance Tuning 623

11.2 Synchronization and Multiprocessing Modes 625
11.2.1 Principles of Synchronization 625
11.2.2 Multiprocessor Execution Modes 628
11.2.3 Multitasking on Cray Multiprocessors 629

11.3 Shared-Variable Program Structures 634
11.3.1 Locks for Protected Access 634
11.3.2 Semaphores and Applications 637
11.3.3 Monitors and Applications 640

11.4 Message-Passing Program Development 644
11.4.1 Distributing the Computation 644
11.4.2 Synchronous Message Passing 645
11.4.3 Asynchronous Message Passing 647

11.5 Mapping Programs onto Multicomputers 648
11.5.1 Domain Decomposition Techniques 648
11.5.2 Control Decomposition Techniques 652
11.5.3 Heterogeneous Processing 656

11.6 Bibliographic Notes and Exercises 661

Chapter 12 UNIX, Mach, and OSF/1 for Parallel Computers 667

12.1 Multiprocessor UNIX Design Goals 667
12.1.1 Conventional UNIX Limitations 668
12.1.2 Compatibility and Portability 670
12.1.3 Address Space and Load Balancing 671
12.1.4 Parallel I/O and Network Services 671

12.2 Master-Slave and Multithreaded UNIX 672
12.2.1 Master-Slave Kernels 672
12.2.2 Floating-Executive Kernels 674
12.2.3 Multithreaded UNIX Kernel 678

12.3 Multicomputer UNIX Extensions 683
12.3.1 Message-Passing OS Models 683
12.3.2 Cosmic Environment and Reactive Kernel 683
12.3.3 Intel NX/2 Kernel and Extensions 685

12.4 Mach/OS Kernel Architecture 686
12.4.1 Mach/OS Kernel Functions 687
12.4.2 Multithreaded Multitasking 688

Copyrighted material

12.4.3 Message-Based Communications 694
12.4.4 Virtual Memory Management 697
12.5 OSF/1 Architecture and Applications 701
12.5.1 The OSF/1 Architecture 702
12.5.2 The OSF/1 Programming Environment 707
12.5.3 Improving Performance with Threads 709
12.6 Bibliographic Notes and Exercises 712

Bibliography **717**

Index **739**

Answers to Selected Problems **765**

Copyrighted material

Foreword

by Gordon Bell

Kai Hwang has introduced the issues in designing and using high performance parallel computers at a time when a plethora of scalable computers utilizing commodity microprocessors offer higher peak performance than traditional vector supercomputers. These new machines, their operating environments including the operating system and languages, and the programs to effectively utilize them are introducing more rapid changes for researchers, builders, and users than at any time in the history of computer structures.

For the first time since the introduction of Cray 1 vector processor in 1975, it may again be necessary to change and evolve the programming paradigm — provided that massively parallel computers can be shown to be useful outside of research on massive parallelism. Vector processors required modest data parallelism and these operations have been reflected either explicitly in Fortran programs or implicitly with the need to evolve Fortran (e.g., Fortran 90) to build in vector operations.

So far, the main line of supercomputing as measured by the usage (hours, jobs, number of programs, program portability) has been the shared memory, vector multiprocessor as pioneered by Cray Research. Fujitsu, IBM, Hitachi, and NEC all produce computers of this type. In 1993, the Cray C90 supercomputer delivers a peak of 16 billion floating-point operations per second (a Gigaflops) with 16 processors and costs about $30 million, providing roughly 500 floating-point operations per second per dollar.

In contrast, massively parallel computers introduced in the early 1990s are nearly all based on utilizing the same powerful, RISC-based, CMOS microprocessors that are used in workstations. These scalar processors provide a peak of $\approx$ 100 million floating-point operations per second and cost $20 thousand, providing an order of magnitude more peak per dollar (5000 flops per dollar). Unfortunately, to obtain peak power requires large-scale problems that can require $O(n^3)$ operations over supers, and this significantly increases the running time when peak power is the goal.

Copyrighted material

The multicomputer approach interconnects computers built from microprocessors through high-bandwidth switches that introduce latency. Programs are written in either an evolved parallel data model utilizing Fortran or as independent programs that communicate by passing messages. The book describes a variety of multicomputers including Thinking Machines' CM5, the first computer announced that could reach a teraflops using 8K independent computer nodes, each of which can deliver 128 Mflops utilizing four 32-Mflops floating-point units.

The architecture research trend is toward scalable, shared-memory multiprocessors in order to handle general workloads ranging from technical to commercial tasks and workloads, negate the need to explicitly pass messages for communication, and provide memory addressed accessing. KSR's scalable multiprocessor and Stanford's Dash prototype have proven that such machines are possible.

The author starts by positing a framework based on evolution that outlines the main approaches to designing computer structures. He covers both the scaling of computers and workloads, various multiprocessors, vector processing, multicomputers, and emerging scalable or multithreaded multiprocessors. The final three chapters describe parallel programming techniques and discuss the host operating environment necessary to utilize these new computers.

The book provides case studies of both industrial and research computers, including the Illinois Cedar, Intel Paragon, TMC CM-2, MasPar M1, TMC CM-5, Cray Y-MP, C-90, and Cray MPP, Fujitsu VP2000 and VPP500, NEC SX, Stanford Dash, KSR-1, MIT J-Machine, MIT *T, ETL EM-4, Caltech Mosaic C, and Tera Computer.

The book presents a balanced treatment of the theory, technology, architecture, and software of advanced computer systems. The emphasis on parallelism, scalability, and programmability makes this book rather unique and educational.

I highly recommend Dr. Hwang's timely book. I believe it will benefit many readers and be a fine reference.

C. Gordon Bell

Copyrighted material

Preface

The Aims

This book provides a comprehensive study of scalable and parallel computer architectures for achieving a proportional increase in performance with increasing system resources. System resources are scaled by the number of processors used, the memory capacity enlarged, the access latency tolerated, the I/O bandwidth required, the performance level desired, etc.

Scalable architectures delivering a sustained performance are desired in both sequential and parallel computers. Parallel architecture has a higher potential to deliver scalable performance. The scalability varies with different architecture-algorithm combinations. Both hardware and software issues need to be studied in building scalable computer systems.

It is my intent to put the reader in a position to design scalable computer systems. Scalability is defined in a broader sense to reflect the interplay among architectures, algorithms, software, and environments. The integration between hardware and software is emphasized for building cost-effective computers.

We should explore cutting-edge technologies in scalable parallel computing. Systems architecture is thus studied with generality, scalability, programmability, and performability in mind.

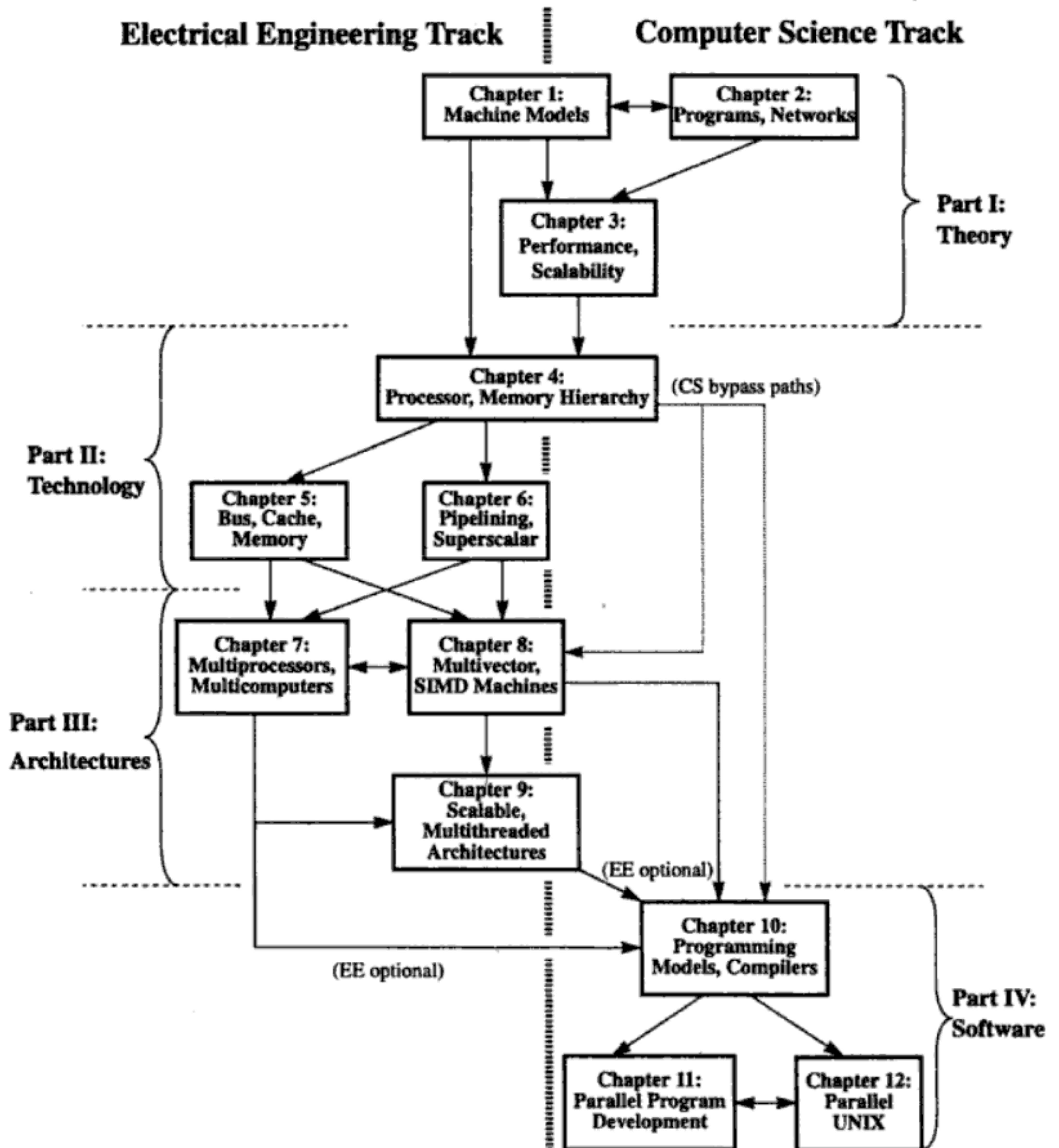
Since high technology changes so rapidly, I have presented the material in a generic manner, unbiased toward particular machine implementations. Representative processors and systems are presented only if they contain important features which may last into the future.

Every author faces the same dilemma in writing a technology-dependent book which may become obsolete quickly. To cope with the problem, frequent updates with newer editions become a necessity, and I plan to make revisions every few years in the future.

Copyrighted material

The Contents

This book consists of twelve chapters divided into four parts covering *theory*, *technology*, *architecture*, and *software* aspects of parallel and vector computers as shown in the flowchart:



Readers' Guide

Copyrighted material

Part I presents principles of parallel processing in three chapters. These include parallel computer models, scalability analysis, theory of parallelism, data dependences, program flow mechanisms, network topologies, benchmark measures, performance laws, and program behaviors. These chapters lay the necessary foundations for readers to study hardware and software in subsequent chapters.

In Part II, three chapters are devoted to studying advanced processors, cache and memory technology, and pipelining techniques. Technological bases touched include RISC, CISC, superscalar, superpipelining, and VLIW architectures. Shared memory, consistency models, cache architecture, and coherence protocols are studied.

Pipelining is extensively applied in memory-access, instruction execution, scalar, superscalar, and vector arithmetic operations. Instruction prefetch, data forwarding, software interlocking, scoreboarding, branch handling, and out-of-order issue and completion are studied for designing advanced processors.

In Part III, three chapters are provided to cover shared-memory multiprocessors, vector and SIMD supercomputers, message-passing multicomputers, and scalable or multithreaded architectures. Since we emphasize scalable architectures, special treatment is given to the IEEE Futurebus+ standards, multistage networks, cache coherence, latency tolerance, fast synchronization, and hierarchical and multidimensional structures for building shared-memory systems.

Massive parallelism is addressed in message-passing systems as well as in synchronous SIMD computers. Shared virtual memory and multithreaded architectures are the important topics, in addition to compound vector processing on pipelined supercomputers and coordinated data parallelism on the CM-5.

Part IV consists of three chapters dealing with parallel programming models, multiprocessor UNIX, software environments, and compiler development for parallel/vector computers. Both shared variables and message-passing schemes are studied for interprocessor communications. Languages, compilers, and software tools for program and benchmark development and performance monitoring are studied.

Among various UNIX extensions, we discuss master-slave, floating-executive, and multithreaded kernels for resource management in a network of heterogeneous computer systems. The Mach/OS and OSF/1 are studied as example systems.

This book has been completely newly written based on recent material. The contents are rather different from my earlier book coauthored with Dr. Faye Briggs in 1983. The two books, separated by 10 years, have very little in common.

The Audience

The material included in this text is an outgrowth of two graduate-level courses: Computer Systems Architecture (EE 557) and Parallel Processing (EE 657) that I have taught at the University of Southern California, the University of Minnesota (Spring 1989), and National Taiwan University (Fall 1991) during the last eight years.

The book can be adopted as a textbook in senior- or graduate-level courses offered by Computer Science (CS), Computer Engineering (CE), Electrical Engineering (EE), or Computational Science programs. The flowchart guides the students and instructors in reading this book.

Copyrighted material

The first four chapters should be taught to all disciplines. The three technology chapters are necessary for EE and CE students. The three architecture chapters can be selectively taught to CE and CS students, depending on the instructor's interest and the computing facilities available to teach the course. The three software chapters are written for CS students and are optional to EE students.

Five course outlines are suggested below for different audiences. The first three outlines are for 45-hour, one-semester courses. The last two outlines are for two-quarter courses in a sequence.

(1) For a Computer Science course on *Parallel Computers and Programming*, the minimum coverage should include Chapters 1–4, 7, and 9–12.
(2) For an exclusive Electrical Engineering course on *Advanced Computer Architecture*, the minimum coverage should include Chapters 1–9.
(3) For a joint CS and EE course on *Parallel Processing Computer Systems*, the minimum coverage should include Chapters 1–4 and 7–12.
(4) Chapters 1 through 6 can be taught to a senior or first-year graduate course under the title *Computer Architecture* in one quarter (10 weeks / 30 hours).
(5) Chapters 7 through 12 can be taught to a graduate course on *Parallel Computer Architecture and Programming* in a one-quarter course with course (4) as the prerequisite.

Instructors may wish to include some advanced research topics treated in Sections 1.4, 2.3, 3.4, 5.4, 6.2, 6.5, 7.2, 7.3, 8.3, 10.4, 11.1, 12.5, and selected sections from Chapter 9 in each of the above course options. The architecture chapters present four different families of commercially available computers. Instructors may choose to teach a subset of these machine families based on the accessibility of corresponding machines on campus or via a public network. Students are encouraged to learn through hands-on programming experience on parallel computers.

A *Solutions Manual* is available to instructors only from the Computer Science Editor, College Division, McGraw-Hill, Inc., 1221 Avenue of the Americas, New York, NY 10020. Answers to a few selected exercise problems are given at the end of the book.

The Prerequisites

This is an advanced text on computer architecture and parallel programming. The reader should have been exposed to some basic computer organization and programming courses at the undergraduate level. Some of the required background material can be found in *Computer Architecture: A Quantitative Approach* by John Hennessy and David Patterson (Morgan Kaufman, 1990) or in *Machine and Assembly Language Programming* by Arthur Gill (Prentice-Hall, 1978).

Students should have some knowledge and experience in logic design, computer hardware, system operations, assembly languages, and Fortran or C programming. Because of the emphasis on scalable architectures and the exploitation of parallelism in practical applications, readers will find it useful to have some background in probability, discrete mathematics, matrix algebra, and optimization theory.

Copyrighted material

Acknowledgments

I have tried to identify all sources of information in the bibliographic notes. As the subject area evolves rapidly, omissions are almost unavoidable. I apologize to those whose valuable work has not been included in this edition. I am responsible for all omissions and for any errors found in the book. Readers are encouraged to contact me directly regarding error correction or suggestions for future editions.

The writing of this book was inspired, taught, or assisted by numerous scholars or specialists working in the area. I would like to thank each of them for intellectual exchanges, valuable suggestions, critical reviews, and technical assistance.

First of all, I want to thank a number of my former and current Ph.D. students. Hwang-Cheng Wang has assisted me in producing the entire manuscript in LaTeX. Besides, he has coauthored the Solutions Manual with Jung-Gen Wu, who visited USC during 1992. Weihua Mao has drawn almost all the figure illustrations using FrameMaker, based on my original sketches. I want to thank D.K. Panda, Joydeep Ghosh, Ahmed Louri, Dongseung Kim, Zhi-Wei Xu, Sugih Jamin, Chien-Ming Cheng, Santosh Rao, Shisheng Shang, Jih-Cheng Liu, Scott Toborg, Stanley Wang, and Myungho Lee for their assistance in collecting material, proofreading, and contributing some of the homework problems. The errata from Teerapon Jungwiwattanaporn were also useful. The Index was compiled by H.C. Wang and J.G. Wu jointly.

I want to thank Gordon Bell for sharing his insights on supercomputing with me and for writing the Foreword to motivate my readers. John Hennessy and Anoop Gupta provided the Dash multiprocessor-related results from Stanford University. Charles Seitz has taught me through his work on Cosmic Cube, Mosaic, and multicomputers. From MIT, I received valuable inputs from the works of Charles Leiserson, William Dally, Anant Agarwal, and Rishiyur Nikhil. From University of Illinois, I received the Cedar and Perfect benchmark information from Pen Yew.

Jack Dongarra of the University of Tennessee provided me the Linpack benchmark results. James Smith of Cray Research provided up-to-date information on the C-90 clusters and on the Cray/MPP. Ken Miura provided the information on Fujitsu VPP500. Lionel Ni of Michigan State University helped me in the areas of performance laws and adaptive wormhole routing. Justin Ratter provided information on the Intel Delta and Paragon systems. Burton Smith provided information on the Tera computer development.

Harold Stone and John Hayes suggested corrections and ways to improve the presentation. H.C. Torng of Cornell University, Andrew Chien of University of Illinois, and Daniel Tobak of George-Mason University made useful suggestions. Among my colleagues at the University of Southern California, Jean-Luc Gaudiot, Michel Dubois, Rafael Saavedra, Monte Ung, and Viktor Prasanna have made concrete suggestions to improve the manuscript. I appreciate the careful proofreading of an earlier version of the manuscript by D.K. Panda of the Ohio State University. The inputs from Vipin Kumar of the University of Minnesota, Xian-He Sun of NASA Langley Research Center, and Alok Choudhary of Syracuse University are also appreciated.

In addition to the above individuals, my understanding on computer architecture and parallel processing has been influenced by the works of David Kuck, Ken Kennedy,

Copyrighted material

Jack Dennis, Michael Flynn, Arvind, T.C. Chen, Wolfgang Giloi, Harry Jordan, H.T. Kung, John Rice, H.J. Siegel, Allan Gottlieb, Philips Treleaven, Faye Briggs, Peter Kogge, Steve Chen, Ben Wah, Edward Davidson, Alvin Despain, James Goodman, Robert Keller, Duncan Lawrie, C.V. Ramamoorthy, Sartaj Sahni, Jean-Loup Baer, Milos Ercegovac, Doug DeGroot, Janak Patel, Dharma Agrawal, Lenart Johnsson, John Gustafson, Tse-Yun Feng, Herbert Schewetman, and Ken Batcher. I want to thank all of them for sharing their vast knowledge with me.

I want to acknowledge the research support I have received from the National Science Foundation, the Office of Naval Research, the Air Force Office of Scientific Research, International Business Machines, Intel Corporation, Alliant Computer Systems, and American Telephone and Telegraph Laboratories.

Technical exchanges with the Electrotechnical Laboratory (ETL) in Japan, the German National Center for Computer Research (GMD) in Germany, and the Industrial Technology Research Institute (ITRI) in Taiwan are always rewarding experiences to the author.

I appreciate the staff and facility support provided by Purdue University, the University of Southern California, the University of Minnesota, the University of Tokyo, National Taiwan University, and Academia Sinica during the past twenty years. In particular, I appreciate the encouragement and professional advices received from Henry Yang, Lofti Zadeh, Richard Karp, George Bekey, Authur Gill, Ben Coates, Melvin Breuer, Jerry Mendel, Len Silverman, Solomon Golomb, and Irving Reed over the years.

Excellent work by the McGraw-Hill editorial and production staff has greatly improved the readability of this book. In particular, I want to thank Eric Munson for his continuous sponsorship of my book projects. I appreciate Joe Murphy and his coworkers for excellent copy editing and production jobs. Suggestions from reviewers listed below have greatly helped improve the contents and presentation.

The book was completed at the expense of cutting back many aspects of my life, spending many long hours, evenings, and weekends in seclusion during the last several years. I appreciate the patience and understanding of my friends, my students, and my family members during those tense periods. Finally, the book has been completed and I hope you enjoy reading it.

Kai Hwang

Reviewers:

Andrew A. Chien, *University of Illinois*;
David Culler, *University of California, Berkeley*;
Ratan K. Guha, *University of Central Florida*;
John P. Hayes, *University of Michigan*;
John Hennessy, *Stanford University*;
Dhamir Mannai, *Northeastern University*;
Michael Quinn, *Oregon State University*;
H. J. Siegel, *Purdue University*;
Daniel Tabak, *George-Mason University*.

Copyrighted material

ADVANCED COMPUTER ARCHITECTURE:
Parallelism, Scalability, Programmability

Copyrighted material

Copyrighted material

Part I

Theory of Parallelism

Chapter 1
Parallel Computer Models

Chapter 2
Program and Network Properties

Chapter 3
Principles of Scalable Performance

Summary

This theoretical part presents computer models, program behavior, architectural choices, scalability, programmability, and performance issues related to parallel processing. These topics form the foundations for designing high-performance computers and for the development of supporting software and applications.

Physical computers modeled include shared-memory multiprocessors, message-passing multicomputers, vector supercomputers, synchronous processor arrays, and massively parallel processors. The theoretical parallel random-access machine (PRAM) model is also presented. Differences between the PRAM model and physical architectural models are discussed. The VLSI complexity model is presented for implementing parallel algorithms directly in integrated circuits.

Network design principles and parallel program characteristics are introduced. These include dependence theory, computing granularity, communication latency, program flow mechanisms, network properties, performance laws, and scalability studies. This evolving theory of parallelism consolidates our understanding of parallel computers, from abstract models to hardware machines, software systems, and performance evaluation.

Copyrighted material

Chapter 1

Parallel Computer Models

Parallel processing has emerged as a key enabling technology in modern computers, driven by the ever-increasing demand for higher performance, lower costs, and sustained productivity in real-life applications. Concurrent events are taking place in today's high-performance computers due to the common practice of multiprogramming, multiprocessing, or multicomputing.

Parallelism appears in various forms, such as lookahead, pipelining, vectorization, concurrency, simultaneity, data parallelism, partitioning, interleaving, overlapping, multiplicity, replication, time sharing, space sharing, multitasking, multiprogramming, multithreading, and distributed computing at different processing levels.

In this chapter, we model physical architectures of parallel computers, vector supercomputers, multiprocessors, multicomputers, and massively parallel processors. Theoretical machine models are also presented, including the *parallel random-access machines* (PRAMs) and the complexity model of VLSI (*very large-scale integration*) circuits. Architectural development tracks are identified with case studies in the book. Hardware and software subsystems are introduced to pave the way for detailed studies in subsequent chapters.

1.1 The State of Computing

Modern computers are equipped with powerful hardware facilities driven by extensive software packages. To assess state-of-the-art computing, we first review historical milestones in the development of computers. Then we take a grand tour of the crucial hardware and software elements built into modern computer systems. We then examine the evolutional relations in milestone architectural development. Basic hardware and software factors are identified in analyzing the performance of computers.

1.1.1 Computer Development Milestones

Computers have gone through two major stages of development: mechanical and electronic. Prior to 1945, computers were made with mechanical or electromechanical

Copyrighted material

parts. The earliest mechanical computer can be traced back to 500 BC in the form of the abacus used in China. The abacus is manually operated to perform decimal arithmetic with carry propagation digit by digit.

Blaise Pascal built a mechanical adder/subtractor in France in 1642. Charles Babbage designed a difference engine in England for polynomial evaluation in 1827. Konrad Zuse built the first binary mechanical computer in Germany in 1941. Howard Aiken proposed the very first electromechanical decimal computer, which was built as the Harvard Mark I by IBM in 1944. Both Zuse's and Aiken's machines were designed for general-purpose computations.

Obviously, the fact that computing and communication were carried out with moving mechanical parts greatly limited the computing speed and reliability of mechanical computers. Modern computers were marked by the introduction of electronic components. The moving parts in mechanical computers were replaced by high-mobility electrons in electronic computers. Information transmission by mechanical gears or levers was replaced by electric signals traveling almost at the speed of light.

Computer Generations Over the past five decades, electronic computers have gone through five generations of development. Table 1.1 provides a summary of the five generations of electronic computer development. Each of the first three generations lasted about 10 years. The fourth generation covered a time span of 15 years. We have just entered the fifth generation with the use of processors and memory devices with more than 1 million transistors on a single silicon chip.

The division of generations is marked primarily by sharp changes in hardware and software technologies. The entries in Table 1.1 indicate the new hardware and software features introduced with each generation. Most features introduced in earlier generations have been passed to later generations. In other words, the latest generation computers have inherited all the nice features and eliminated all the bad ones found in previous generations.

Progress in Hardware As far as hardware technology is concerned, the first generation (1945–1954) used vacuum tubes and relay memories interconnected by insulated wires. The second generation (1955–1964) was marked by the use of discrete transistors, diodes, and magnetic ferrite cores, interconnected by printed circuits.

The third generation (1965–1974) began to use *integrated circuits* (ICs) for both logic and memory in *small-scale* or *medium-scale integration* (SSI or MSI) and multilayered printed circuits. The fourth generation (1974–1991) used *large-scale* or *very-large-scale integration* (LSI or VLSI). Semiconductor memory replaced core memory as computers moved from the third to the fourth generation.

The fifth generation (1991–present) is highlighted by the use of high-density and high-speed processor and memory chips based on even more improved VLSI technology. For example, 64-bit 150-MHz microprocessors are now available on a single chip with over one million transistors. Four-megabit dynamic *random-access memory* (RAM) and 256K-bit static RAM are now in widespread use in today's high-performance computers.

It has been projected that four microprocessors will be built on a single CMOS chip with more than 50 million transistors, and 64M-bit dynamic RAM will become

Copyrighted material

Table 1.1 Five Generations of Electronic Computers

Generation	Technology and Architecture	Software and Applications	Representative Systems
First (1945–54)	Vacuum tubes and relay memories, CPU driven by PC and accumulator, fixed-point arithmetic.	Machine/assembly languages, single user, no subroutine linkage, programmed I/O using CPU.	ENIAC, Princeton IAS, IBM 701.
Second (1955–64)	Discrete transistors and core memories, floating-point arithmetic, I/O processors, multiplexed memory access.	HLL used with compilers, subroutine libraries, batch processing monitor.	IBM 7090, CDC 1604, Univac LARC.
Third (1965–74)	Integrated circuits (SSI/-MSI), microprogramming, pipelining, cache, and lookahead processors.	Multiprogramming and time-sharing OS, multiuser applications.	IBM 360/370, CDC 6600, TI-ASC, PDP-8.
Fourth (1975–90)	LSI/VLSI and semiconductor memory, multiprocessors, vector supercomputers, multicomputers.	Multiprocessor OS, languages, compilers, and environments for parallel processing.	VAX 9000, Cray X-MP, IBM 3090, BBN TC2000.
Fifth (1991–present)	ULSI/VHSIC processors, memory, and switches, high-density packaging, scalable architectures.	Massively parallel processing, grand challenge applications, heterogeneous processing.	Fujitsu VPP500, Cray/MPP, TMC/CM-5, Intel Paragon.

available in large quantities within the next decade.

The First Generation From the architectural and software points of view, first-generation computers were built with a single *central processing unit* (CPU) which performed serial fixed-point arithmetic using a program counter, branch instructions, and an accumulator. The CPU must be involved in all memory access and *input/output* (I/O) operations. Machine or assembly languages were used.

Representative systems include the ENIAC (Electronic Numerical Integrator and Calculator) built at the Moore School of the University of Pennsylvania in 1950; the IAS (Institute for Advanced Studies) computer based on a design proposed by John von Neumann, Arthur Burks, and Herman Goldstine at Princeton in 1946; and the IBM 701, the first electronic stored-program commercial computer built by IBM in 1953. Subroutine linkage was not implemented in early computers.

The Second Generation Index registers, floating-point arithmetic, multiplexed memory, and I/O processors were introduced with second-generation computers. *High-level languages* (HLLs), such as Fortran, Algol, and Cobol, were introduced along with compilers, subroutine libraries, and batch processing monitors. Register transfer language was developed by Irving Reed (1957) for systematic design of digital computers.

Representative systems include the IBM 7030 (the Stretch computer) featuring

Copyrighted material

instruction lookahead and error-correcting memories built in 1962, the Univac LARC (Livermore Atomic Research Computer) built in 1959, and the CDC 1604 built in the 1960s.

The Third Generation The third generation was represented by the IBM/360–370 Series, the CDC 6600/7600 Series, Texas Instruments ASC (Advanced Scientific Computer), and Digital Equipment's PDP-8 Series from the mid-1960s to the mid-1970s.

Microprogrammed control became popular with this generation. Pipelining and cache memory were introduced to close up the speed gap between the CPU and main memory. The idea of multiprogramming was implemented to interleave CPU and I/O activities across multiple user programs. This led to the development of time-sharing *operating systems* (OS) using virtual memory with greater sharing or multiplexing of resources.

The Fourth Generation Parallel computers in various architectures appeared in the fourth generation of computers using shared or distributed memory or optional vector hardware. Multiprocessing OS, special languages, and compilers were developed for parallelism. Software tools and environments were created for parallel processing or distributed computing.

Representative systems include the VAX 9000, Cray X-MP, IBM/3090 VF, BBN TC-2000, etc. During these 15 years (1975–1990), the technology of parallel processing gradually became mature and entered the production mainstream.

The Fifth Generation Fifth-generation computers have just begun to appear. These machines emphasize *massively parallel processing* (MPP). Scalable and latency-tolerant architectures are being adopted in MPP systems using VLSI silicon, GaAs technologies, high-density packaging, and optical technologies.

Fifth-generation computers are targeted to achieve Teraflops (10^{12} floating-point operations per second) performance by the mid-1990s. *Heterogeneous processing* is emerging to solve large-scale problems using a network of heterogeneous computers with shared virtual memories. The fifth-generation MPP systems are represented by several recently announced projects at Fujitsu (VPP500), Cray Research (MPP), Thinking Machines Corporation (the CM-5), and Intel Supercomputer Systems (the Paragon).

1.1.2 Elements of Modern Computers

Hardware, software, and programming elements of a modern computer system are briefly introduced below in the context of parallel processing.

Computing Problems It has been long recognized that the concept of computer architecture is no longer restricted to the structure of the bare machine hardware. A modern computer is an integrated system consisting of machine hardware, an instruction set, system software, application programs, and user interfaces. These system elements are depicted in Fig. 1.1. The use of a computer is driven by real-life problems demanding

Copyrighted material

fast and accurate solutions. Depending on the nature of the problems, the solutions may require different computing resources.

For numerical problems in science and technology, the solutions demand complex mathematical formulations and tedious integer or floating-point computations. For alphanumerical problems in business and government, the solutions demand accurate transactions, large database management, and information retrieval operations.

For artificial intelligence (AI) problems, the solutions demand logic inferences and symbolic manipulations. These computing problems have been labeled *numerical computing*, *transaction processing*, and *logical reasoning*. Some complex problems may demand a combination of these processing modes.

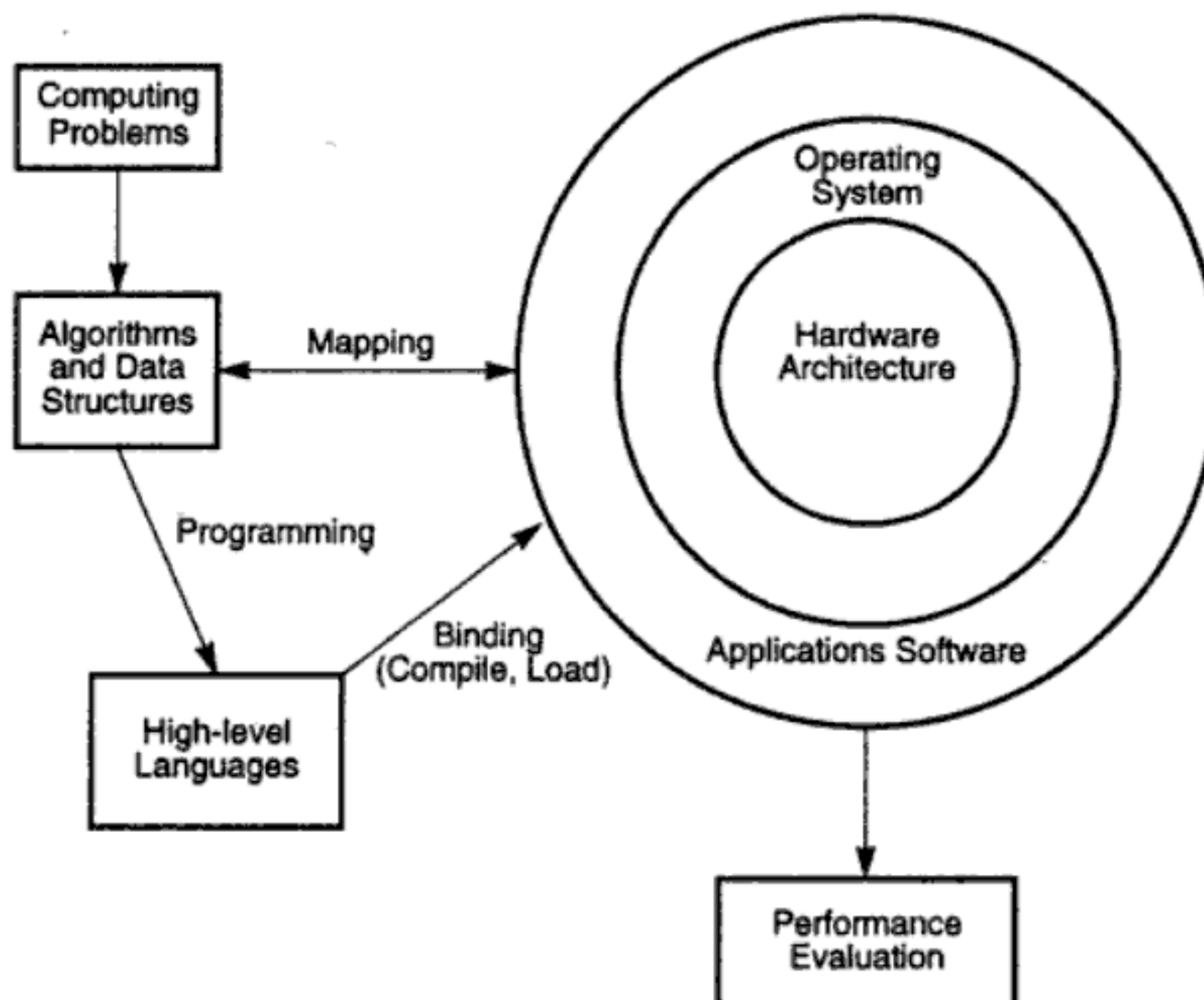


Figure 1.1 Elements of a modern computer system.

Algorithms and Data Structures Special algorithms and data structures are needed to specify the computations and communications involved in computing problems. Most numerical algorithms are deterministic, using regularly structured data. Symbolic processing may use heuristics or nondeterministic searches over large knowledge bases.

Problem formulation and the development of parallel algorithms often require interdisciplinary interactions among theoreticians, experimentalists, and computer programmers. There are many books dealing with the design and mapping of algorithms or heuristics onto parallel computers. In this book, we are more concerned about the

Copyrighted material

resources mapping problem than about the design and analysis of parallel algorithms.

Hardware Resources The system architecture of a computer is represented by three nested circles on the right in Fig. 1.1. A modern computer system demonstrates its power through coordinated efforts by hardware resources, an operating system, and application software. Processors, memory, and peripheral devices form the hardware core of a computer system. We will study instruction-set processors, memory organization, multiprocessors, supercomputers, multicomputers, and massively parallel computers.

Special hardware interfaces are often built into I/O devices, such as terminals, workstations, optical page scanners, magnetic ink character recognizers, modems, file servers, voice data entry, printers, and plotters. These peripherals are connected to mainframe computers directly or through local or wide-area networks.

In addition, software interface programs are needed. These software interfaces include file transfer systems, editors, word processors, device drivers, interrupt handlers, network communication programs, etc. These programs greatly facilitate the portability of user programs on different machine architectures.

Operating System An effective operating system manages the allocation and deallocation of resources during the execution of user programs. We will study UNIX extensions for multiprocessors and multicomputers in Chapter 12. Mach/OS kernel and OSF/1 will be specially studied for multithreaded kernel functions, virtual memory management, file subsystems, and network communication services. Beyond the OS, application software must be developed to benefit the users. Standard benchmark programs are needed for performance evaluation.

Mapping is a bidirectional process matching algorithmic structure with hardware architecture, and vice versa. Efficient mapping will benefit the programmer and produce better source codes. The mapping of algorithmic and data structures onto the machine architecture includes processor scheduling, memory maps, interprocessor communications, etc. These activities are usually architecture-dependent.

Optimal mappings are sought for various computer architectures. The implementation of these mappings relies on efficient compiler and operating system support. Parallelism can be exploited at algorithm design time, at program time, at compile time, and at run time. Techniques for exploiting parallelism at these levels form the core of parallel processing technology.

System Software Support Software support is needed for the development of efficient programs in high-level languages. The source code written in a HLL must be first translated into object code by an optimizing compiler. The *compiler* assigns variables to registers or to memory words and reserves functional units for operators. An *assembler* is used to translate the compiled object code into machine code which can be recognized by the machine hardware. A *loader* is used to initiate the program execution through the OS kernel.

Resource binding demands the use of the compiler, assembler, loader, and OS kernel to commit physical machine resources to program execution. The effectiveness of this process determines the efficiency of hardware utilization and the programmability of the

Copyrighted material

computer. Today, programming parallelism is still very difficult for most programmers due to the fact that existing languages were originally developed for sequential computers. Programmers are often forced to program hardware-dependent features instead of programming parallelism in a generic and portable way. Ideally, we need to develop a parallel programming environment with architecture-independent languages, compilers, and software tools.

To develop a parallel language, we aim for efficiency in its implementation, portability across different machines, compatibility with existing sequential languages, expressiveness of parallelism, and ease of programming. One can attempt a new language approach or try to extend existing sequential languages gradually. A new language approach has the advantage of using explicit high-level constructs for specifying parallelism. However, new languages are often incompatible with existing languages and require new compilers or new passes to existing compilers. Most systems choose the language extension approach.

Compiler Support There are three compiler upgrade approaches: *preprocessor*, *precompiler*, and *parallelizing compiler*. A preprocessor uses a sequential compiler and a low-level library of the target computer to implement high-level parallel constructs. The precompiler approach requires some program flow analysis, dependence checking, and limited optimizations toward parallelism detection. The third approach demands a fully developed parallelizing or vectorizing compiler which can automatically detect parallelism in source code and transform sequential codes into parallel constructs. These approaches will be studied in Chapter 10.

The efficiency of the binding process depends on the effectiveness of the preprocessor, the precompiler, the parallelizing compiler, the loader, and the OS support. Due to unpredictable program behavior, none of the existing compilers can be considered fully automatic or fully intelligent in detecting all types of parallelism. Very often *compiler directives* are inserted into the source code to help the compiler do a better job. Users may interact with the compiler to restructure the programs. This has been proven useful in enhancing the performance of parallel computers.

1.1.3 Evolution of Computer Architecture

The study of computer architecture involves both hardware organization and programming/software requirements. As seen by an assembly language programmer, computer architecture is abstracted by its instruction set, which includes opcode (operation codes), addressing modes, registers, virtual memory, etc.

From the hardware implementation point of view, the abstract machine is organized with CPUs, caches, buses, microcode, pipelines, physical memory, etc. Therefore, the study of architecture covers both instruction-set architectures and machine implementation organizations.

Over the past four decades, computer architecture has gone through evolutional rather than revolutional changes. Sustaining features are those that were proven performance deliverers. As depicted in Fig. 1.2, we started with the von Neumann architecture built as a sequential machine executing scalar data. The sequential computer was

Copyrighted material

improved from bit-serial to word-parallel operations, and from fixed-point to floating-point operations. The von Neumann architecture is slow due to sequential execution of instructions in programs.

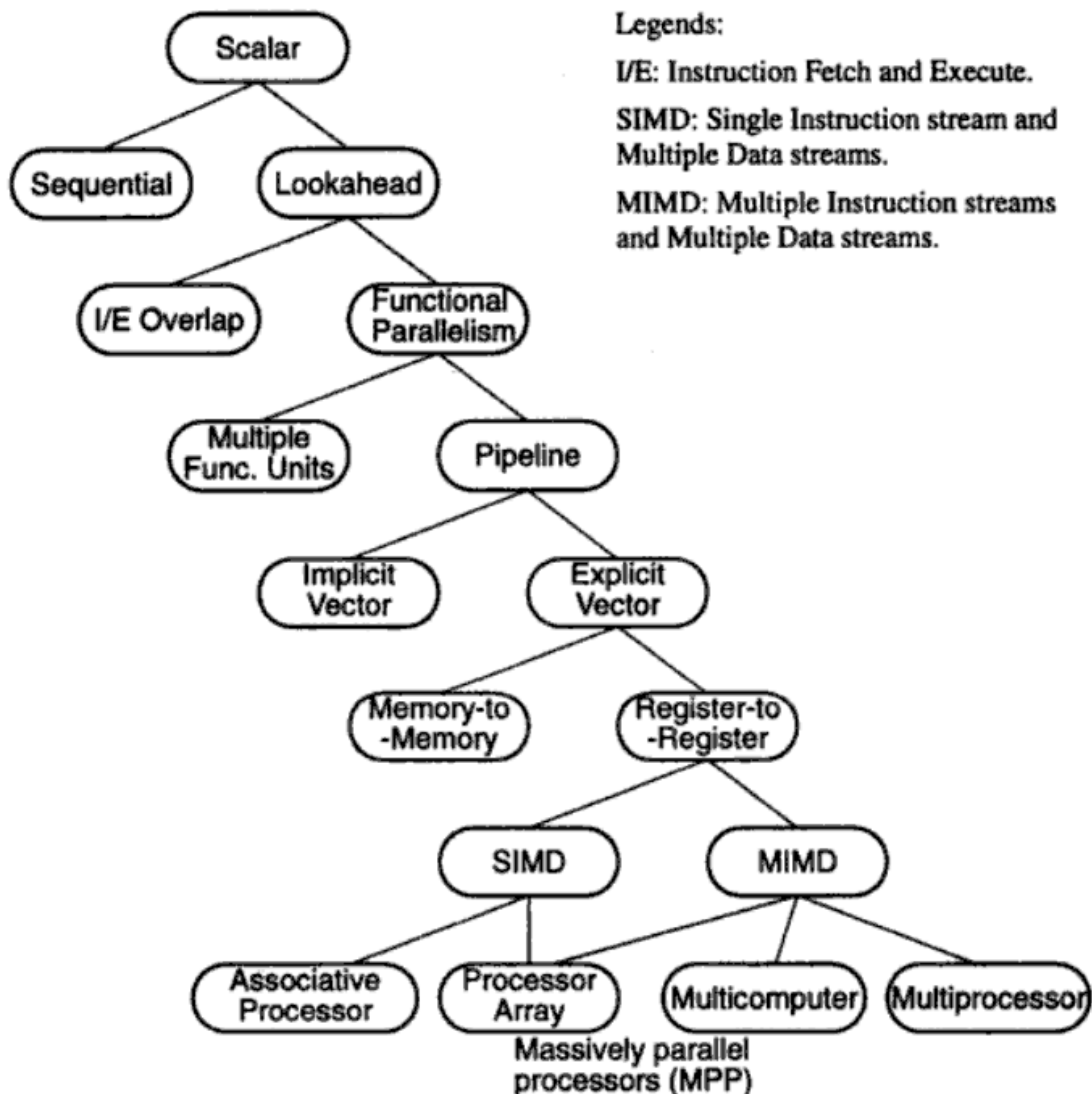


Figure 1.2 Tree showing architectural evolution from sequential scalar computers to vector processors and parallel computers.

Lookahead, Parallelism, and Pipelining Lookahead techniques were introduced to prefetch instructions in order to overlap I/E (instruction fetch/decode and execution) operations and to enable functional parallelism. Functional parallelism was supported by two approaches: One is to use multiple functional units simultaneously, and the other is to practice pipelining at various processing levels.

The latter includes pipelined instruction execution, pipelined arithmetic computations, and memory-access operations. Pipelining has proven especially attractive in

Copyrighted material

performing identical operations repeatedly over vector data strings. Vector operations were originally carried out implicitly by software-controlled looping using scalar pipeline processors.

Flynn's Classification Michael Flynn (1972) introduced a classification of various computer architectures based on notions of instruction and data streams. As illustrated in Fig. 1.3a, conventional sequential machines are called SISD (*single instruction stream over a single data stream*) computers. Vector computers are equipped with scalar and vector hardware or appear as SIMD (*single instruction stream over multiple data streams*) machines (Fig. 1.3b). Parallel computers are reserved for MIMD (*multiple instruction streams over multiple data streams*) machines.

An MISD (*multiple instruction streams and a single data stream*) machines are modeled in Fig. 1.3d. The same data stream flows through a linear array of processors executing different instruction streams. This architecture is also known as *systolic arrays* (Kung and Leiserson, 1978) for pipelined execution of specific algorithms.

Of the four machine models, most parallel computers built in the past assumed the MIMD model for general-purpose computations. The SIMD and MISD models are more suitable for special-purpose computations. For this reason, MIMD is the most popular model, SIMD next, and MISD the least popular model being applied in commercial machines.

Parallel/Vector Computers Intrinsic parallel computers are those that execute programs in MIMD mode. There are two major classes of parallel computers, namely, *shared-memory multiprocessors* and *message-passing multicomputers*. The major distinction between multiprocessors and multicomputers lies in memory sharing and the mechanisms used for interprocessor communication.

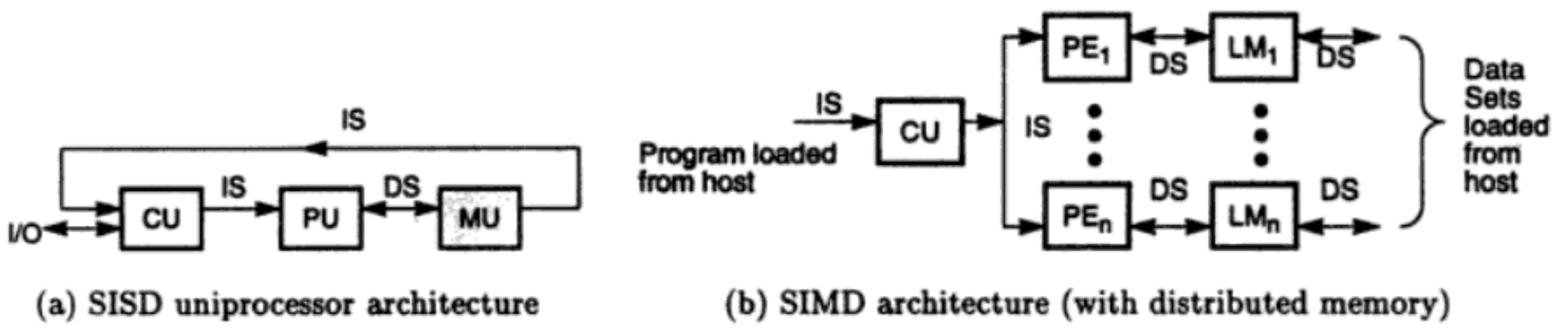
The processors in a multiprocessor system communicate with each other through *shared variables* in a common memory. Each computer node in a multicomputer system has a local memory, unshared with other nodes. Interprocessor communication is done through *message passing* among the nodes.

Explicit vector instructions were introduced with the appearance of *vector processors*. A vector processor is equipped with multiple vector pipelines that can be concurrently used under hardware or firmware control. There are two families of pipelined vector processors:

Memory-to-memory architecture supports the pipelined flow of vector operands directly from the memory to pipelines and then back to the memory. *Register-to-register* architecture uses vector registers to interface between the memory and functional pipelines. Vector processor architectures will be studied in Chapter 8.

Another important branch of the architecture tree consists of the SIMD computers for synchronized vector processing. An SIMD computer exploits *spatial parallelism* rather than *temporal parallelism* as in a pipelined computer. SIMD computing is achieved through the use of an array of *processing elements* (PEs) synchronized by the same controller. Associative memory can be used to build SIMD associative processors. SIMD machines will be treated in Chapter 8 along with pipelined vector computers.

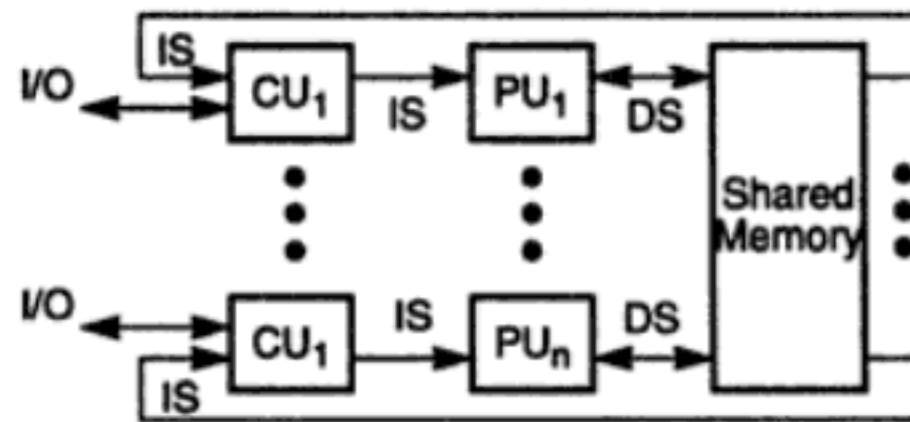
Copyrighted material



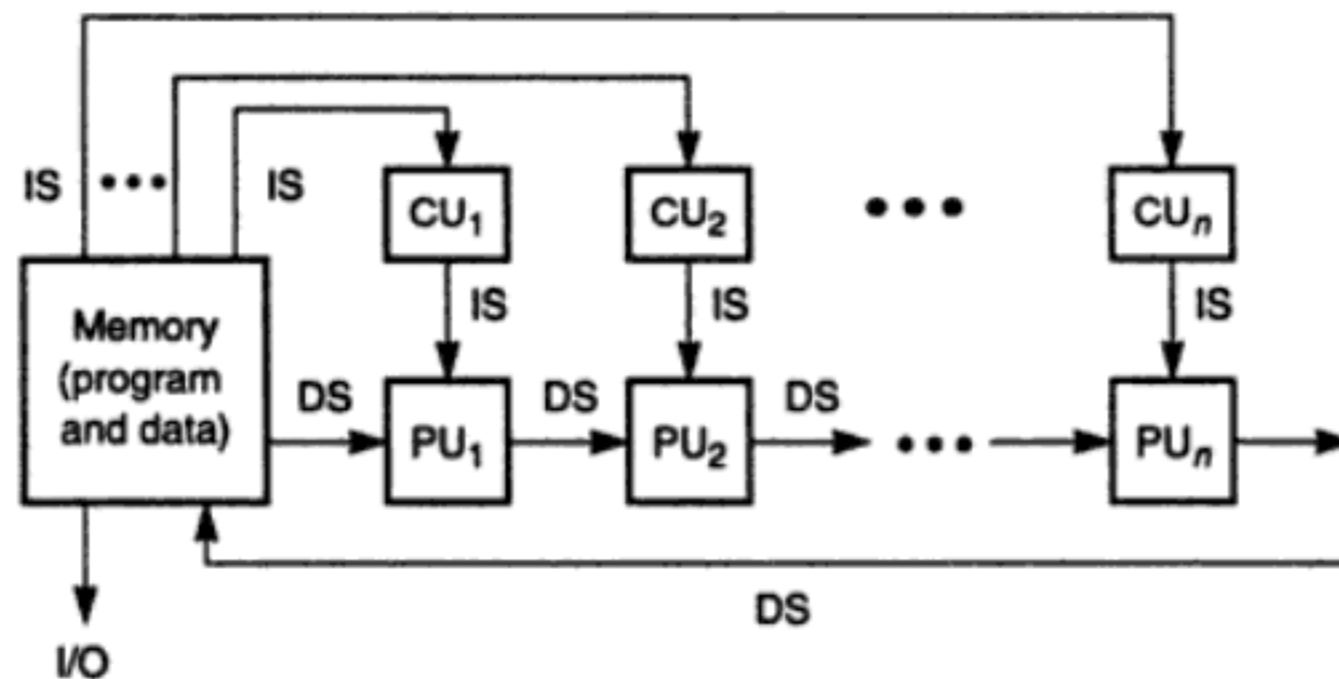
(a) SISD uniprocessor architecture

(b) SIMD architecture (with distributed memory)

Captions:
CU = Control Unit
PU = Processing Unit
MU = Memory Unit
IS = Instruction Stream
DS = Data Stream
PE = Processing Element
LM = Local Memory



(c) MIMD architecture (with shared memory)



(d) MISD architecture (the systolic array)

Figure 1.3 Flynn's classification of computer architectures. (Derived from Michael Flynn, 1972)

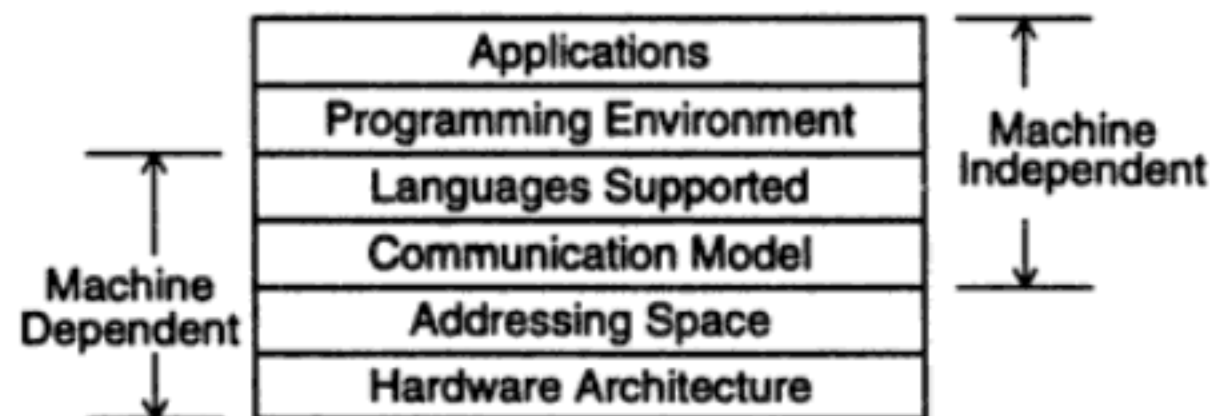


Figure 1.4 Six layers for computer system development. (Courtesy of Lionel Ni, 1990)

Copyrighted material

Development Layers A layered development of parallel computers is illustrated in Fig. 1.4, based on a recent classification by Lionel Ni (1990). Hardware configurations differ from machine to machine, even those of the same model. The address space of a processor in a computer system varies among different architectures. It depends on the memory organization, which is machine-dependent. These features are up to the designer and should match the target application domains.

On the other hand, we want to develop application programs and programming environments which are machine-independent. Independent of machine architecture, the user programs can be ported to many computers with minimum conversion costs. High-level languages and communication models depend on the architectural choices made in a computer system. From a programmer's viewpoint, these two layers should be architecture-transparent.

At present, Fortran, C, Pascal, Ada, and Lisp are supported by most computers. However, the communication models, shared variables versus message passing, are mostly machine-dependent. The Linda approach using *tuple spaces* offers an architecture-transparent communication model for parallel computers. These language features will be studied in Chapter 10.

Application programmers prefer more architectural transparency. However, kernel programmers have to explore the opportunities supported by hardware. As a good computer architect, one has to approach the problem from both ends. The compilers and OS support should be designed to remove as many architectural constraints as possible from the programmer.

New Challenges The technology of parallel processing is the outgrowth of four decades of research and industrial advances in microelectronics, printed circuits, high-density packaging, advanced processors, memory systems, peripheral devices, communication channels, language evolution, compiler sophistication, operating systems, programming environments, and application challenges.

The rapid progress made in hardware technology has significantly increased the economical feasibility of building a new generation of computers adopting parallel processing. However, the major barrier preventing parallel processing from entering the production mainstream is on the software and application side.

To date, it is still very difficult and painful to program parallel and vector computers. We need to strive for major progress in the software area in order to create a user-friendly environment for high-power computers. A whole new generation of programmers need to be trained to program parallelism effectively. High-performance computers provide fast and accurate solutions to scientific, engineering, business, social, and defense problems.

Representative real-life problems include weather forecast modeling, computer-aided design of VLSI circuits, large-scale database management, artificial intelligence, crime control, and strategic defense initiatives, just to name a few. The application domains of parallel processing computers are expanding steadily. With a good understanding of scalable computer architectures and mastery of parallel programming techniques. the reader will be better prepared to face future computing challenges.

Copyrighted material

1.1.4 System Attributes to Performance

The ideal performance of a computer system demands a perfect match between machine capability and program behavior. Machine capability can be enhanced with better hardware technology, innovative architectural features, and efficient resources management. However, program behavior is difficult to predict due to its heavy dependence on application and run-time conditions.

There are also many other factors affecting program behavior, including algorithm design, data structures, language efficiency, programmer skill, and compiler technology. It is impossible to achieve a perfect match between hardware and software by merely improving only a few factors without touching other factors.

Besides, machine performance may vary from program to program. This makes *peak performance* an impossible target to achieve in real-life applications. On the other hand, a machine cannot be said to have an average performance either. All performance indices or benchmarking results must be tied to a program mix. For this reason, the performance should be described as a range or as a harmonic distribution.

We introduce below fundamental factors for projecting the performance of a computer. These performance indicators are by no means conclusive in all applications. However, they can be used to guide system architects in designing better machines or to educate programmers or compiler writers in optimizing the codes for more efficient execution by the hardware.

Consider the execution of a given program on a given computer. The simplest measure of program performance is the *turnaround time*, which includes disk and memory accesses, input and output activities, compilation time, OS overhead, and CPU time. In order to shorten the turnaround time, one must reduce all these time factors.

In a multiprogrammed computer, the I/O and system overheads of a given program may overlap with the CPU times required in other programs. Therefore, it is fair to compare just the total CPU time needed for program execution. The CPU is used to execute both system programs and user programs. It is the user CPU time that concerns the user most.

Clock Rate and CPI The CPU (or simply the *processor*) of today's digital computer is driven by a clock with a constant *cycle time* (τ in nanoseconds). The inverse of the cycle time is the *clock rate* ($f = 1/\tau$ in megahertz). The size of a program is determined by its *instruction count* (I_c), in terms of the number of machine instructions to be executed in the program. Different machine instructions may require different numbers of clock cycles to execute. Therefore, the *cycles per instruction* (CPI) becomes an important parameter for measuring the time needed to execute each instruction.

For a given instruction set, we can calculate an *average* CPI over all instruction types, provided we know their frequencies of appearance in the program. An accurate estimate of the average CPI requires a large amount of program code to be traced over a long period of time. Unless specifically focusing on a single instruction type, we simply use the term CPI to mean the average value with respect to a given instruction set and a given program mix.

Copyrighted material

Performance Factors Let I_c be the number of instructions in a given program, or the instruction count. The CPU time (T in seconds/program) needed to execute the program is estimated by finding the product of three contributing factors:

$$T = I_c \times \text{CPI} \times \tau \tag{1.1}$$

The execution of an instruction requires going through a cycle of events involving the instruction fetch, decode, operand(s) fetch, execution, and store results. In this cycle, only the instruction decode and execution phases are carried out in the CPU. The remaining three operations may be required to access the memory. We define a *memory cycle* as the time needed to complete one memory reference. Usually, a memory cycle is k times the processor cycle τ. The value of k depends on the speed of the memory technology and processor-memory interconnection scheme used.

The CPI of an instruction type can be divided into two component terms corresponding to the total processor cycles and memory cycles needed to complete the execution of the instruction. Depending on the instruction type, the complete instruction cycle may involve one to four memory references (one for instruction fetch, two for operand fetch, and one for store results). Therefore we can rewrite Eq. 1.1 as follows:

$$T = I_c \times (p + m \times k) \times \tau \tag{1.2}$$

where p is the number of processor cycles needed for the instruction decode and execution, m is the number of memory references needed, k is the ratio between memory cycle and processor cycle, I_c is the instruction count, and τ is the processor cycle time. Equation 1.2 can be further refined once the CPI components (p, m, k) are weighted over the entire instruction set.

System Attributes The above five performance factors (I_c, p, m, k, τ) are influenced by four system attributes: instruction-set architecture, compiler technology, CPU implementation and control, and cache and memory hierarchy, as specified in Table 1.2.

The instruction-set architecture affects the program length (I_c) and processor cycle needed (p). The compiler technology affects the values of I_c, p, and the memory reference count (m). The CPU implementation and control determine the total processor time ($p \cdot \tau$) needed. Finally, the memory technology and hierarchy design affect the memory access latency ($k \cdot \tau$). The above CPU time can be used as a basis in estimating the execution rate of a processor.

MIPS Rate Let C be the total number of clock cycles needed to execute a given program. Then the CPU time in Eq. 1.2 can be estimated as $T = C \times \tau = C/f$. Furthermore, $\text{CPI} = C/I_c$ and $T = I_c \times \text{CPI} \times \tau = I_c \times \text{CPI}/f$. The processor speed is often measured in terms of *million instructions per second* (MIPS). We simply call it the MIPS rate of a given processor. It should be emphasized that the MIPS rate varies with respect to a number of factors, including the clock rate (f), the instruction count (I_c), and the CPI of a given machine, as defined below:

$$\text{MIPS rate} = \frac{I_c}{T \times 10^6} = \frac{f}{\text{CPI} \times 10^6} = \frac{f \times I_c}{C \times 10^6} \tag{1.3}$$

Copyrighted material

Table 1.2 Performance Factors Versus System Attributes

System Attributes	Performance Factors				
	Instr. Count, I_c	Average Cycles per Instruction, CPI			Processor Cycle Time, τ
		Processor Cycles per Instruction, p	Memory References per Instruction, m	Memory-Access Latency, k	
Instruction-set Architecture	X	X			
Compiler Technology	X	X	X		
Processor Implementation and Control		X			X
Cache and Memory Hierarchy				X	X

Based on Eq. 1.3, the CPU time in Eq. 1.2 can also be written as $T = I_c \times 10^{-6}/\text{MIPS}$. Based on the system attributes identified in Table 1.2 and the above derived expressions, we conclude by indicating the fact that the MIPS rate of a given computer is directly proportional to the clock rate and inversely proportional to the CPI. All four system attributes, instruction set, compiler, processor, and memory technologies, affect the MIPS rate, which varies also from program to program.

Throughput Rate Another important concept is related to how many programs a system can execute per unit time, called the *system throughput* W_s (in programs/second). In a multiprogrammed system, the system throughput is often lower than the *CPU throughput* W_p defined by:

$$W_p = \frac{f}{I_c \times \text{CPI}} \tag{1.4}$$

Note that $W_p = (\text{MIPS}) \times 10^6 / I_c$ from Eq. 1.3. The unit for W_p is programs/second. The CPU throughput is a measure of how many programs can be executed per second, based on the MIPS rate and average program length (I_c). The reason why $W_s < W_p$ is due to the additional system overheads caused by the I/O, compiler, and OS when multiple programs are interleaved for CPU execution by multiprogramming or timesharing operations. If the CPU is kept busy in a perfect program-interleaving fashion, then $W_s = W_p$. This will probably never happen, since the system overhead often causes an extra delay and the CPU may be left idle for some cycles.

Example 1.1 MIPS ratings and performance measurement

Consider the use of a VAX/780 and an IBM RS/6000-based workstation to execute a hypothetical benchmark program. Machine characteristics and claimed performance are given below:

Copyrighted material

Machine	Clock	Performance	CPU Time
VAX 11/780	5 MHz	1 MIPS	$12x$ seconds
IBM RS/6000	25 MHz	18 MIPS	x seconds

These data indicate that the measured CPU time on the VAX 11/780 is 12 times longer than that measured on the RS/6000. The object codes running on the two machines have different lengths due to the differences in the machines and compilers used. All other overhead times are ignored.

Based on Eq. 1.3, the instruction count of the object code running on the RS/6000 is 1.5 times longer than that of the code running on the VAX machine. Furthermore, the average CPI on the VAX/780 is assumed to be 5, while that on the RS/6000 is 1.39 executing the same benchmark program.

The VAX 11/780 has a typical CISC (*complex instruction set computing*) architecture, while the IBM machine has a typical RISC (*reduced instruction set computing*) architecture to be characterized in Chapter 4. This example offers a simple comparison between the two types of computers based on a single program run. When a different program is run, the conclusion may not be the same.

We cannot calculate the CPU throughput W_p unless we know the program length and the average CPI of each code. The system throughput W_s should be measured across a large number of programs over a long observation period. The message being conveyed is that one should not draw a sweeping conclusion about the performance of a machine based on one or a few program runs.

■

Programming Environments The programmability of a computer depends on the programming environment provided to the users. Most computer environments are not user-friendly. In fact, the marketability of any new computer system depends on the creation of a user-friendly environment in which programming becomes a joyful undertaking rather than a nuisance. We briefly introduce below the environmental features desired in modern computers.

Conventional uniprocessor computers are programmed in a *sequential environment* in which instructions are executed one after another in a sequential manner. In fact, the original UNIX/OS kernel was designed to respond to one system call from the user process at a time. Successive system calls must be serialized through the kernel.

Most existing compilers are designed to generate sequential object codes to run on a sequential computer. In other words, conventional computers are being used in a sequential programming environment using languages, compilers, and operating systems all developed for a uniprocessor computer.

When using a parallel computer, one desires a *parallel environment* where parallelism is automatically exploited. Language extensions or new constructs must be developed to specify parallelism or to facilitate easy detection of parallelism at various granularity levels by more intelligent compilers.

Besides parallel languages and compilers, the operating systems must be also extended to support parallel activities. The OS must be able to manage the resources

Copyrighted material

behind parallelism. Important issues include parallel scheduling of concurrent events, shared memory allocation, and shared peripheral and communication links.

Implicit Parallelism An implicit approach uses a conventional language, such as C, Fortran, Lisp, or Pascal, to write the source program. The sequentially coded source program is translated into parallel object code by a parallelizing compiler. As illustrated in Fig. 1.5a, this compiler must be able to detect parallelism and assign target machine resources. This compiler approach has been applied in programming shared-memory multiprocessors.

With parallelism being implicit, success relies heavily on the "intelligence" of a parallelizing compiler. This approach requires less effort on the part of the programmer. David Kuck of the University of Illinois and Ken Kennedy of Rice University and their associates have adopted this implicit-parallelism approach.

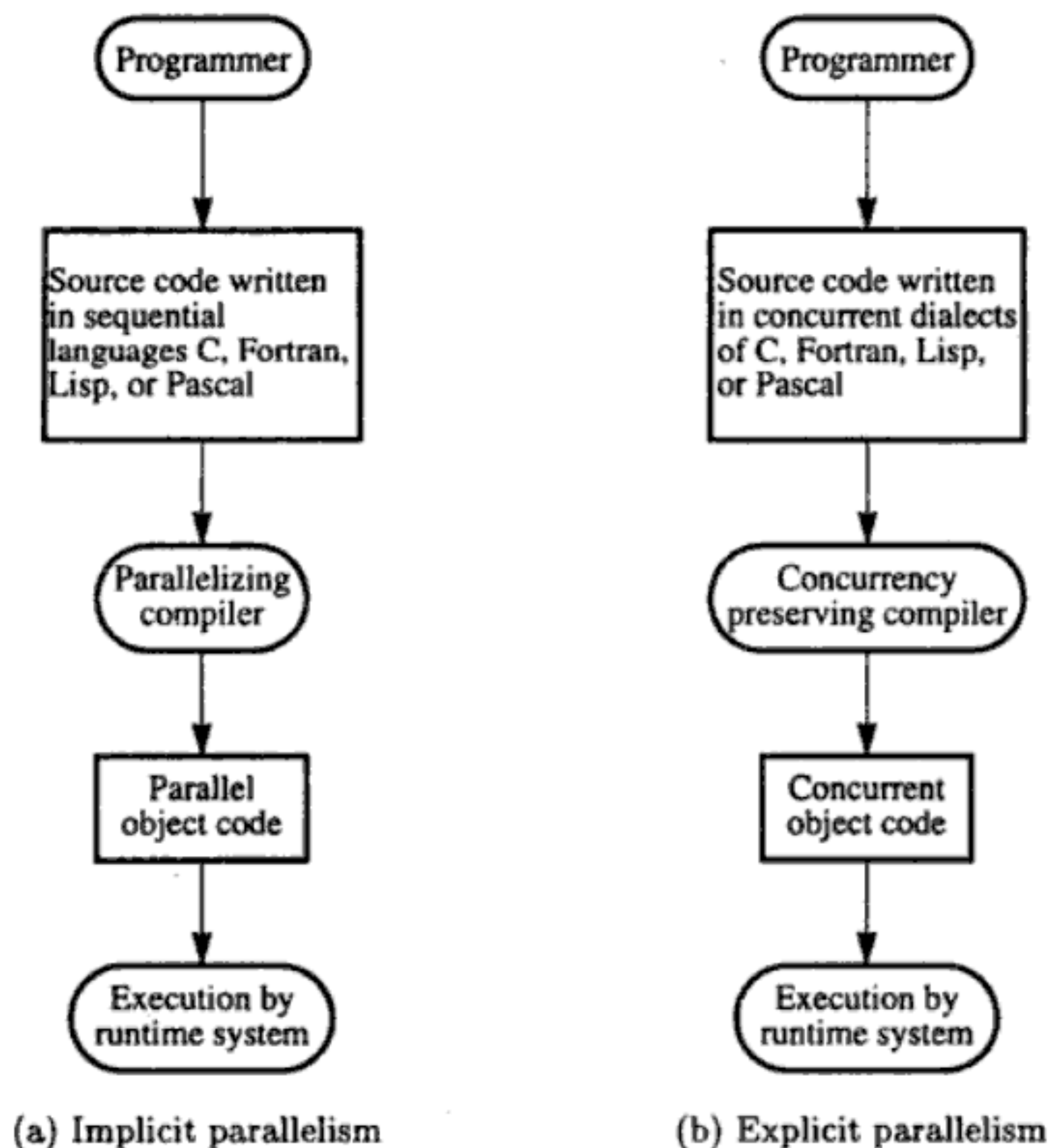


Figure 1.5 Two approaches to parallel programming. (Courtesy of Charles Seitz; reprinted with permission from "Concurrent Architectures", p. 51 and p. 53, *VLSI and Parallel Computation*, edited by Suaya and Birtwistle, Morgan Kaufmann Publishers, 1990)

Copyrighted material

Explicit Parallelism The second approach (Fig. 1.5b) requires more effort by the programmer to develop a source program using parallel dialects of C, Fortran, Lisp, or Pascal. Parallelism is explicitly specified in the user programs. This will significantly reduce the burden on the compiler to detect parallelism. Instead, the compiler needs to preserve parallelism and, where possible, assigns target machine resources. Charles Seitz of California Institute of Technology and William Dally of Massachusetts Institute of Technology adopted this explicit approach in multicomputer development.

Special software tools are needed to make an environment more friendly to user groups. Some of the tools are parallel extensions of conventional high-level languages. Others are integrated environments which include tools providing different levels of program abstraction, validation, testing, debugging, and tuning; performance prediction and monitoring; and visualization support to aid program development, performance measurement, and graphics display and animation of computational results.

1.2 Multiprocessors and Multicomputers

Two categories of parallel computers are architecturally modeled below. These physical models are distinguished by having a shared common memory or unshared distributed memories. Only architectural organization models are described in Sections 1.2 and 1.3. Theoretical and complexity models for parallel computers are presented in Section 1.4.

1.2.1 Shared-Memory Multiprocessors

We describe below three shared-memory multiprocessor models: the *uniform-memory-access* (UMA) model, the *nonuniform-memory-access* (NUMA) model, and the *cache-only memory architecture* (COMA) model. These models differ in how the memory and peripheral resources are shared or distributed.

The UMA Model In a UMA multiprocessor model (Fig. 1.6), the physical memory is uniformly shared by all the processors. All processors have equal access time to all memory words, which is why it is called uniform memory access. Each processor may use a private cache. Peripherals are also shared in some fashion.

Multiprocessors are called *tightly coupled systems* due to the high degree of resource sharing. The system interconnect takes the form of a common bus, a crossbar switch, or a multistage network to be studied in Chapter 7.

Most computer manufacturers have *multiprocessor* (MP) extensions of their *uniprocessor* (UP) product line. The UMA model is suitable for general-purpose and time-sharing applications by multiple users. It can be used to speed up the execution of a single large program in time-critical applications. To coordinate parallel events, synchronization and communication among processors are done through using shared variables in the common memory.

When all processors have equal access to all peripheral devices, the system is called a *symmetric* multiprocessor. In this case, all the processors are equally capable of running the executive programs, such as the OS kernel and I/O service routines.

Copyrighted material

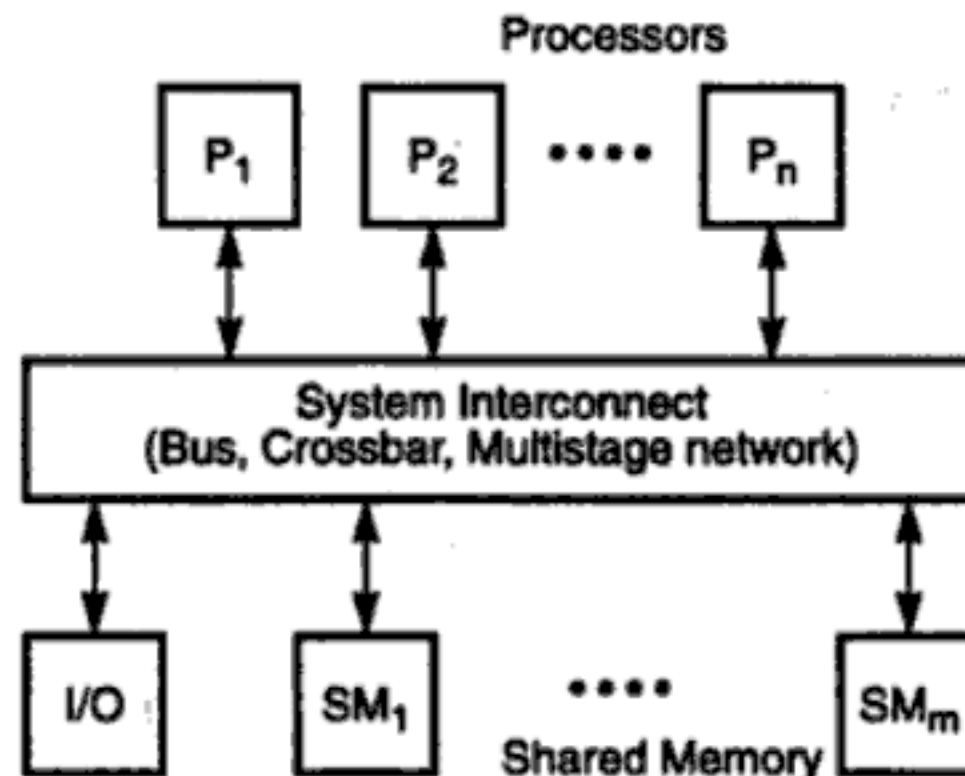


Figure 1.6 The UMA multiprocessor model (e.g., the Sequent Symmetry S-81).

In an *asymmetric* multiprocessor, only one or a subset of processors are executive-capable. An executive or a master processor can execute the operating system and handle I/O. The remaining processors have no I/O capability and thus are called *attached processors* (APs). Attached processors execute user codes under the supervision of the master processor. In both MP and AP configurations, memory sharing among master and attached processors is still in place.

Example 1.2 Approximated performance of a multiprocessor

This example exposes the reader to parallel program execution on a shared-memory multiprocessor system. Consider the following Fortran program written for sequential execution on a uniprocessor system. All the arrays, A(I), B(I), and C(I), are assumed to have N elements.

```
L1:        Do 10 I = 1, N
L2:        A(I) = B(I) + C(I)
L3:    10  Continue
L4:        SUM = 0
L5:        Do 20 J = 1, N
L6:        SUM = SUM + A(J)
L7:    20  Continue
```

Suppose each line of code L2, L4, and L6 takes 1 machine cycle to execute. The time required to execute the program control statements L1, L3, L5, and L7 is ignored to simplify the analysis. Assume that k cycles are needed for each interprocessor communication operation via the shared memory.

Initially, all arrays are assumed already loaded in the main memory and the short program fragment already loaded in the instruction cache. In other words, instruction fetch and data loading overhead is ignored. Also, we ignore bus con-

Copyrighted material

tention or memory access conflicts problems. In this way, we can concentrate on the analysis of CPU demand.

The above program can be executed on a sequential machine in $2N$ cycles under the above assumptions. N cycles are needed to execute the N independent iterations in the I loop. Similarly, N cycles are needed for the J loop, which contains N recursive iterations.

To execute the program on an M-processor system, we partition the looping operations into M sections with $L = N/M$ elements per section. In the following parallel code, **Doall** declares that all M sections be executed by M processors in parallel:

```
        Doall K=1,M
              Do 10 I = L(K-1) + 1, KL
                    A(I) = B(I) + C(I)
10            Continue
              SUM(K) = 0
              Do 20 J = 1, L
                    SUM(K) = SUM(K) + A(L(K-1) + J)
20            Continue
        Endall
```

For M-way parallel execution, the sectioned I loop can be done in L cycles. The sectioned J loop produces M partial sums in L cycles. Thus $2L$ cycles are consumed to produce all M partial sums. Still, we need to merge these M partial sums to produce the final sum of N elements.

The addition of each pair of partial sums requires k cycles through the shared memory. An l-level binary adder tree can be constructed to merge all the partial sums, where $l = \log_2 M$. The adder tree takes $l(k+1)$ cycles to merge the M partial sums sequentially from the leaves to the root of the tree. Therefore, the multiprocessor requires $2L + l(k+1) = 2N/M + (k+1)\log_2 M$ cycles to produce the final sum.

Suppose $N = 2^{20}$ elements in the array. Sequential execution of the original program takes $2N = 2^{21}$ machine cycles. Assume that each IPC synchronization overhead has an average value of $k = 200$ cycles. Parallel execution on $M = 256$ processors requires $2^{13} + 1608 = 9800$ machine cycles.

Comparing the above timing results, the multiprocessor shows a speedup factor of 214 out of the maximum value of 256. Therefore, an efficiency of $214/256 = 83.6\%$ has been achieved. We will study the speedup and efficiency issues in Chapter 3.

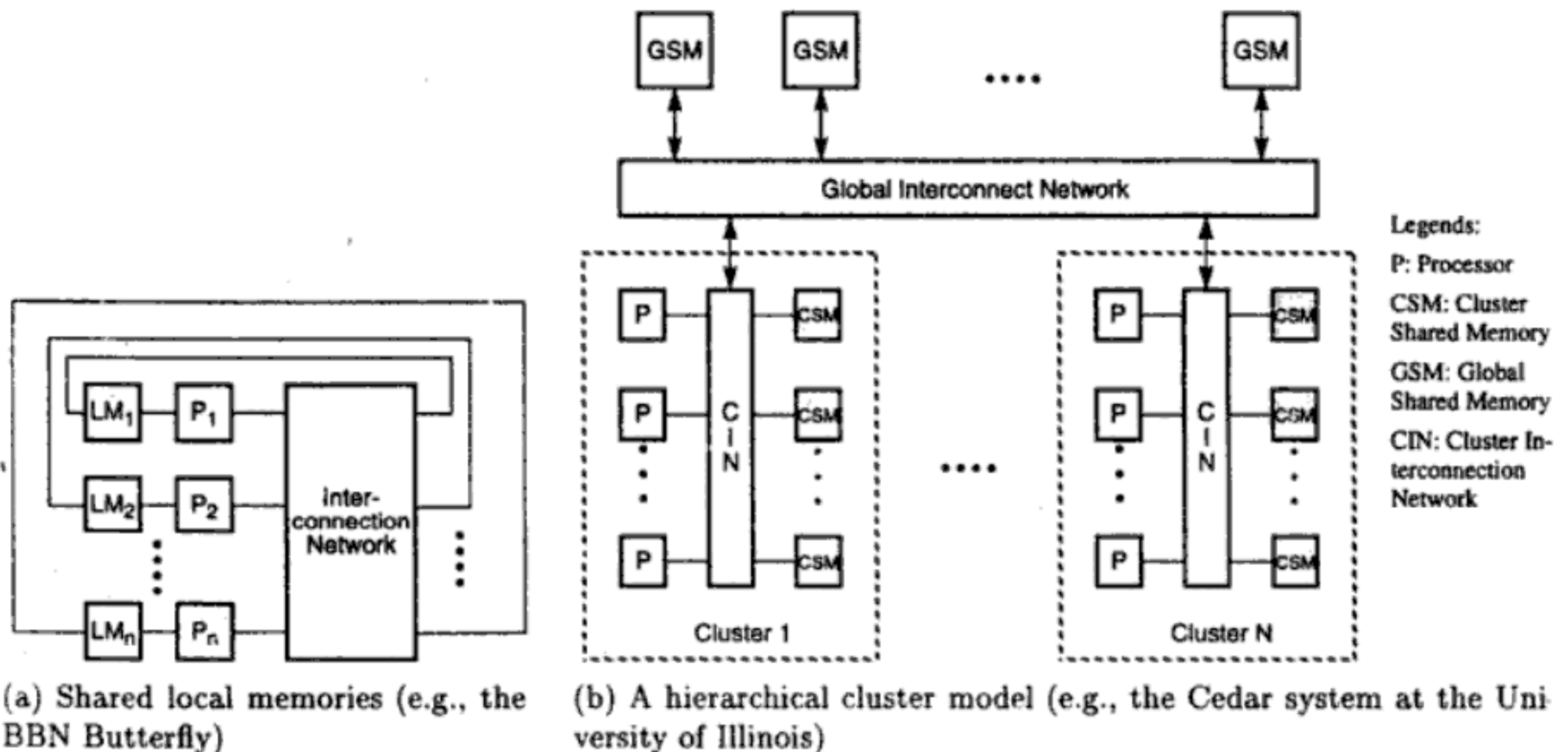
The above result was obtained under favorable assumptions about overhead. In reality, the resulting speedup might be lower after considering all software overhead and potential resource conflicts. Nevertheless, the example shows the promising side of parallel processing if the interprocessor communication overhead can be reduced to a sufficiently low level.

■

Copyrighted material

The NUMA Model A NUMA multiprocessor is a shared-memory system in which the access time varies with the location of the memory word. Two NUMA machine models are depicted in Fig. 1.7. The shared memory is physically distributed to all processors, called *local memories*. The collection of all local memories forms a global address space accessible by all processors.

It is faster to access a local memory with a local processor. The access of remote memory attached to other processors takes longer due to the added delay through the interconnection network. The BBN TC-2000 Butterfly multiprocessor assumes the configuration shown in Fig. 1.7a.



(a) Shared local memories (e.g., the BBN Butterfly)

(b) A hierarchical cluster model (e.g., the Cedar system at the University of Illinois)

Figure 1.7 Two NUMA models for multiprocessor systems.

Besides distributed memories, globally shared memory can be added to a multiprocessor system. In this case, there are three memory-access patterns: The fastest is local memory access. The next is global memory access. The slowest is access of remote memory as illustrated in Fig. 1.7b. As a matter of fact, the models shown in Figs. 1.6 and 1.7 can be easily modified to allow a mixture of shared memory and private memory with prespecified access rights.

A hierarchically structured multiprocessor is modeled in Fig. 1.7b. The processors are divided into several *clusters*. Each cluster is itself an UMA or a NUMA multiprocessor. The clusters are connected to *global shared-memory* modules. The entire system is considered a NUMA multiprocessor. All processors belonging to the same cluster are allowed to uniformly access the *cluster shared-memory* modules.

All clusters have equal access to the global memory. However, the access time to the cluster memory is shorter than that to the global memory. One can specify the access right among intercluster memories in various ways. The Cedar multiprocessor,

Copyrighted material

built at the University of Illinois, assumes such a structure in which each cluster is an Alliant FX/80 multiprocessor.

Table 1.3 Representative Multiprocessor Systems

Company and Model	Hardware and Architecture	Software and Applications	Remarks
Sequent Symmetry S-81	Bus-connected with 30 i386 processors, IPC via SLIC bus; Weitek floating-point accelerator.	DYNIX/OS, KAP/Sequent preprocessor, transaction multiprocessing.	i486-based multiprocessor available 1991. i586-based systems to appear.
IBM ES/9000 Model 900/VF	Model 900/VF has 6 ES/9000 processors with vector facilities, crossbar connected to I/O channels and shared memory.	OS support: MVS, VM KMS, AIX/370, parallel Fortran, VSF V2.5 compiler.	Fiber optic channels, integrated cryptographic architecture.
BBN TC-2000	512 M88100 processors with local memory connected by a Butterfly switch, a NUMA machine.	Ported Mach/OS with multiclustering, use parallel Fortran, time-critical applications.	Shooting for higher performance using faster processors in future models.

The COMA Model A multiprocessor using cache-only memory assumes the COMA model. Examples of COMA machines include the Swedish Institute of Computer Science's Data Diffusion Machine (DDM, Hagersten et al., 1990) and Kendall Square Research's KSR-1 machine (Burkhardt et al., 1992). The COMA model is depicted in Fig. 1.8. Details of KSR-1 are given in Chapter 9.

The COMA model is a special case of a NUMA machine, in which the distributed main memories are converted to caches. There is no memory hierarchy at each processor node. All the caches form a global address space. Remote cache access is assisted by the distributed cache directories (D in Fig. 1.8). Depending on the interconnection network used, sometimes hierarchical directories may be used to help locate copies of cache blocks. Initial data placement is not critical because data will eventually migrate to where it will be used.

Besides the UMA, NUMA, and COMA models specified above, other variations exist for multiprocessors. For example, a *cache-coherent non-uniform memory access* (CC-NUMA) model can be specified with distributed shared memory and cache directories. Examples of the CC-NUMA model include the Stanford Dash (Lenoski et al., 1990) and the MIT Alewife (Agarwal et al., 1990) to be studied in Chapter 9. One can also insist on a cache-coherent COMA machine in which all cache copies must be kept consistent.

Copyrighted material

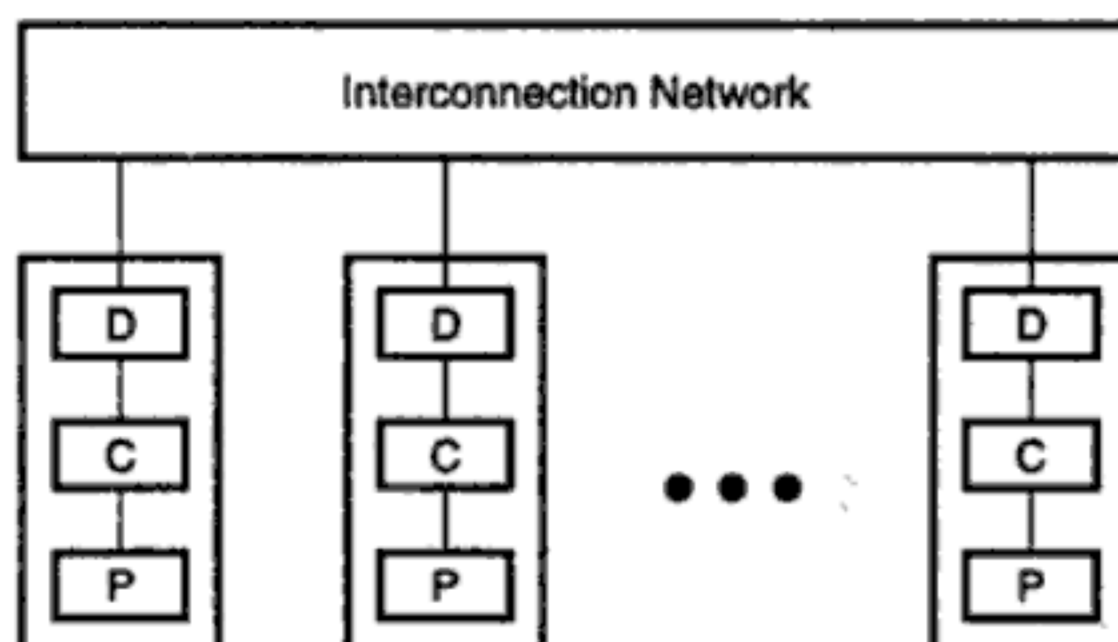


Figure 1.8 The COMA model of a multiprocessor. (P: Processor, C: Cache, D: Directory; e.g., the KSR-1)

Representative Multiprocessors Several commercially available multiprocessors are summarized in Table 1.3. They represent four classes of multiprocessors. The Sequent Symmetry S81 belongs to a class called minisupercomputers. The IBM System/390 models are high-end mainframes, sometimes called near-supercomputers. The BBN TC-2000 represents the MPP class.

The S-81 is a transaction processing multiprocessor consisting of 30 i386/i486 microprocessors tied to a common backplane bus. The IBM ES/9000 models are the latest IBM mainframes having up to 6 processors with attached vector facilities. The TC-2000 can be configured to have 512 M88100 processors interconnected by a multistage Butterfly network. This is designed as a NUMA machine for real-time or time-critical applications.

Multiprocessor systems are suitable for general-purpose multiuser applications where programmability is the major concern. A major shortcoming of multiprocessors is the lack of scalability. It is rather difficult to build MPP machines using centralized shared-memory model. Latency tolerance for remote memory access is also a major limitation.

Packaging and cooling impose additional constraints on scalability. We will study scalability and programmability in subsequent chapters. In building MPP systems, distributed-memory multicomputers are more scalable but less programmable due to added communication protocols.

1.2.2 Distributed-Memory Multicomputers

A distributed-memory multicomputer system is modeled in Fig. 1.9. The system consists of multiple computers, often called *nodes*, interconnected by a message-passing network. Each node is an autonomous computer consisting of a processor, local memory, and sometimes attached disks or I/O peripherals.

The message-passing network provides point-to-point static connections among the nodes. All local memories are private and are accessible only by local processors. For this reason, traditional multicomputers have been called *no-remote-memory-access* (NORMA) machines. However, this restriction will gradually be removed in future mul-

Copyrighted material

ticomputers with distributed shared memories. Internode communication is carried out by passing messages through the static connection network.

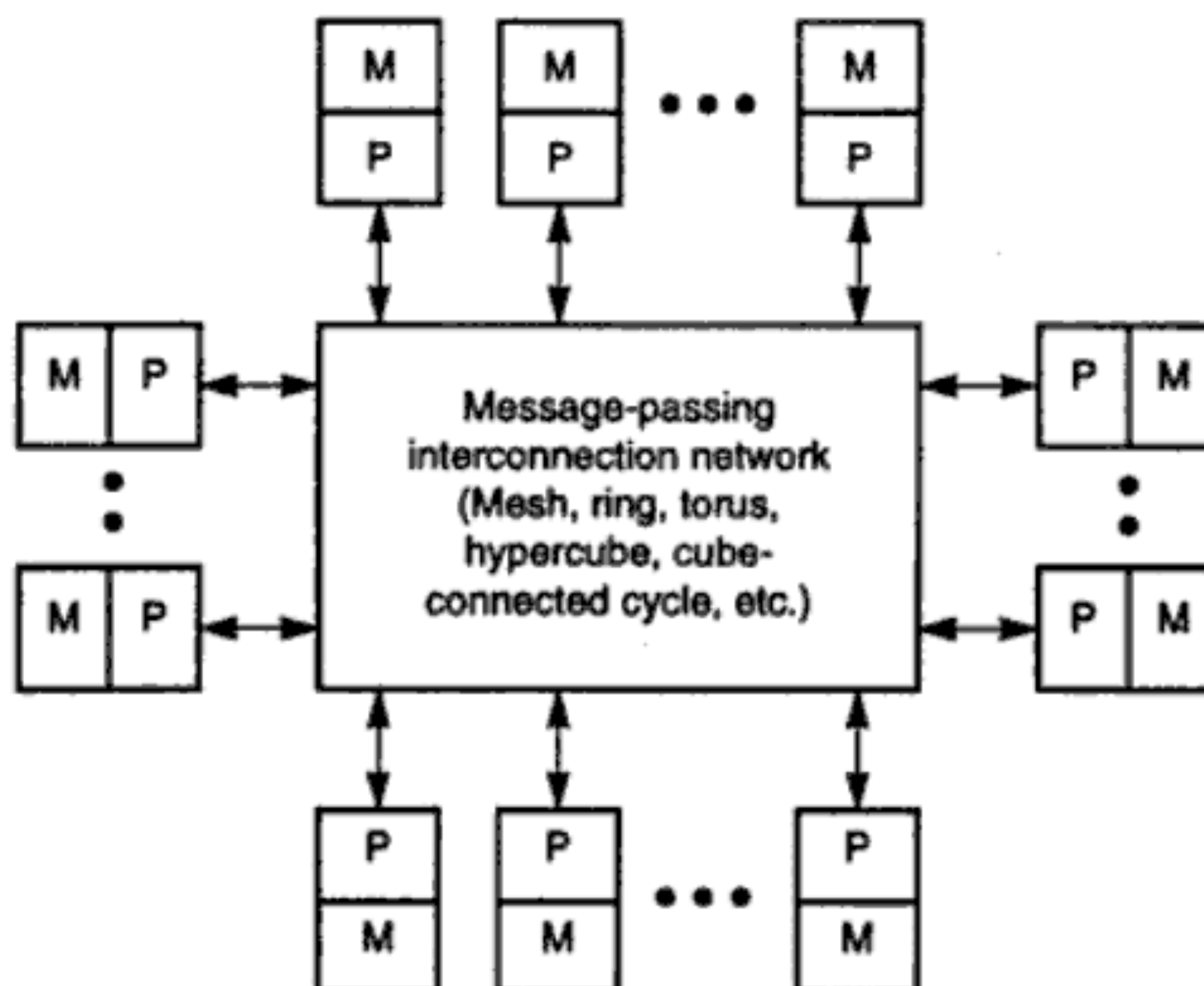


Figure 1.9 Generic model of a message-passing multicomputer.

Multicomputer Generations Modern multicomputers use hardware routers to pass messages. A computer node is attached to each router. The boundary router may be connected to I/O and peripheral devices. Message passing between any two nodes involves a sequence of routers and channels. Mixed types of nodes are allowed in a heterogeneous multicomputer. The internode communications in a heterogeneous multicomputer are achieved through compatible data representations and message-passing protocols.

Message-passing multicomputers have gone through two generations of development, and a new generation is emerging.

The *first generation* (1983–1987) was based on processor board technology using hypercube architecture and software-controlled message switching. The Caltech Cosmic and Intel iPSC/1 represented the first-generation development.

The *second generation* (1988–1992) was implemented with mesh-connected architecture, hardware message routing, and a software environment for medium-grain distributed computing, as represented by the Intel Paragon and the Parsys SuperNode 1000.

The emerging *third generation* (1993–1997) is expected to be fine-grain multicomputers, like the MIT J-Machine and Caltech Mosaic, implemented with both processor and communication gears on the same VLSI chip.

In Section 2.4, we will study various static network topologies used to construct

Copyrighted material

multicomputers. Famous topologies include the *ring*, *tree*, *mesh*, *torus*, *hypercube*, *cube-connected cycle*, etc. Various communication patterns are demanded among the nodes, such as one-to-one, broadcasting, permutations, and multicast patterns.

Important issues for multicomputers include message-routing schemes, network flow control strategies, deadlock avoidance, virtual channels, message-passing primitives, and program decomposition techniques. In Part IV, we will study the programming issues of three generations of multicomputers.

Representative Multicomputers Three message-passing multicomputers are summarized in Table 1.4. With distributed processor/memory nodes, these machines are better in achieving a scalable performance. However, message passing imposes a hardship on programmers to distribute the computations and data sets over the nodes or to establish efficient communication among nodes. Until intelligent compilers and efficient distributed OSs become available, multicomputers will continue to lack programmability.

Table 1.4 Representative Multicomputer Systems

System Features	Intel Paragon XP/S	nCUBE/2 6480	Parsys Ltd. SuperNode1000
Node Types and Memory	50 MHz i860 XP computing nodes with 16–128 Mbytes per node, special I/O service nodes.	Each node contains a CISC 64-bit CPU, with FPU, 14 DMA ports, with 1–64 Mbytes/node.	EC-funded Esprit supernode built with multiple T-800 Transputers per node.
Network and I/O	2-D mesh with SCSI, HIPPI, VME, Ethernet, and custom I/O.	13-dimensional hypercube of 8192 nodes, 512-Gbyte memory, 64 I/O boards.	Reconfigurable interconnect, expandable to have 1024 processors.
OS and Software task parallelism Support	OSF conformance with 4.3 BSD, visualization and programming support.	Vertex/OS or UNIX supporting message passing using wormhole routing.	IDRIS/OS is UNIX-compatible, supported.
Application Drivers	General sparse matrix methods, parallel data manipulation, strategic computing.	Scientific number crunching with scalar nodes, database processing.	Scientific and academic applications.
Performance Remarks	5–300 Gflops peak 64-bit results, 2.8–160 GIPS peak integer performance.	27 Gflops peak, 36 Gbytes/s I/O, challenge to build even larger machine.	200 MIPS to 13 GIPS peak, largest supernode in use contains 256 Transputers.

The Paragon system assumes a mesh architecture, and the nCUBE/2 has a hy-

Copyrighted material

percube architecture. The Intel i860s and some custom-designed VLSI processors are used as building blocks in these machines. All three OSs are UNIX-compatible with extended functions to support message passing.

Most multicomputers are being upgraded to yield a higher degree of parallelism with enhanced processors. We will study various massively parallel systems in Part III where the tradeoffs between scalability and programmability are analyzed.

1.2.3 A Taxonomy of MIMD Computers

Parallel computers appear as either SIMD or MIMD configurations. The SIMDs appeal more to special-purpose applications. It is clear that SIMDs are not size-scalable, but unclear whether large SIMDs are generation-scalable. The fact that CM-5 has an MIMD architecture, away from the SIMD architecture in CM-2, may shed some light on the architectural trend. Furthermore, the boundary between multiprocessors and multicomputers has become blurred in recent years, Eventually, the distinctions may vanish.

The architectural trend for future general-purpose computers is in favor of MIMD configurations with distributed memories having a globally shared virtual address space. For this reason, Gordon Bell (1992) has provided a taxonomy of MIMD machines, reprinted in Fig. 1.10. He considers shared-memory multiprocessors as having a single address space. Scalable multiprocessors or multicomputers must use distributed shared memory. Unscalable multiprocessors use centrally shared memory.

Multicomputers use distributed memories with multiple address spaces. They are scalable with distributed memory. Centralized multicomputers are yet to appear. Many of the identified example systems will be treated in subsequent chapters. The evolution of fast LAN (*local area network*)-connected workstations will create "commodity supercomputing". Bell advocates high-speed workstation clusters interconnected by high-speed switches in lieu of special-purpose multicomputers. The CM-5 development has already moved in this direction.

The scalability of MIMD computers will be further studied in Section 3.4 and Chapter 9. In Part III, we will study distributed-memory multiprocessors (KSR-1, SCI, etc.); central-memory multiprocessors (Cray, IBM, DEC, Fujitsu, Encore, etc.); multicomputers by Intel, TMC, and nCUBE; fast LAN-based workstation clusters; and other exploratory research systems.

1.3 Multivector and SIMD Computers

In this section, we introduce supercomputers and parallel processors for vector processing and data parallelism. We classify supercomputers either as pipelined vector machines using a few powerful processors equipped with vector hardware, or as SIMD computers emphasizing massive data parallelism.

1.3.1 Vector Supercomputers

A vector computer is often built on top of a scalar processor. As shown in Fig. 1.11, the vector processor is attached to the scalar processor as an optional feature. Program

Copyrighted material

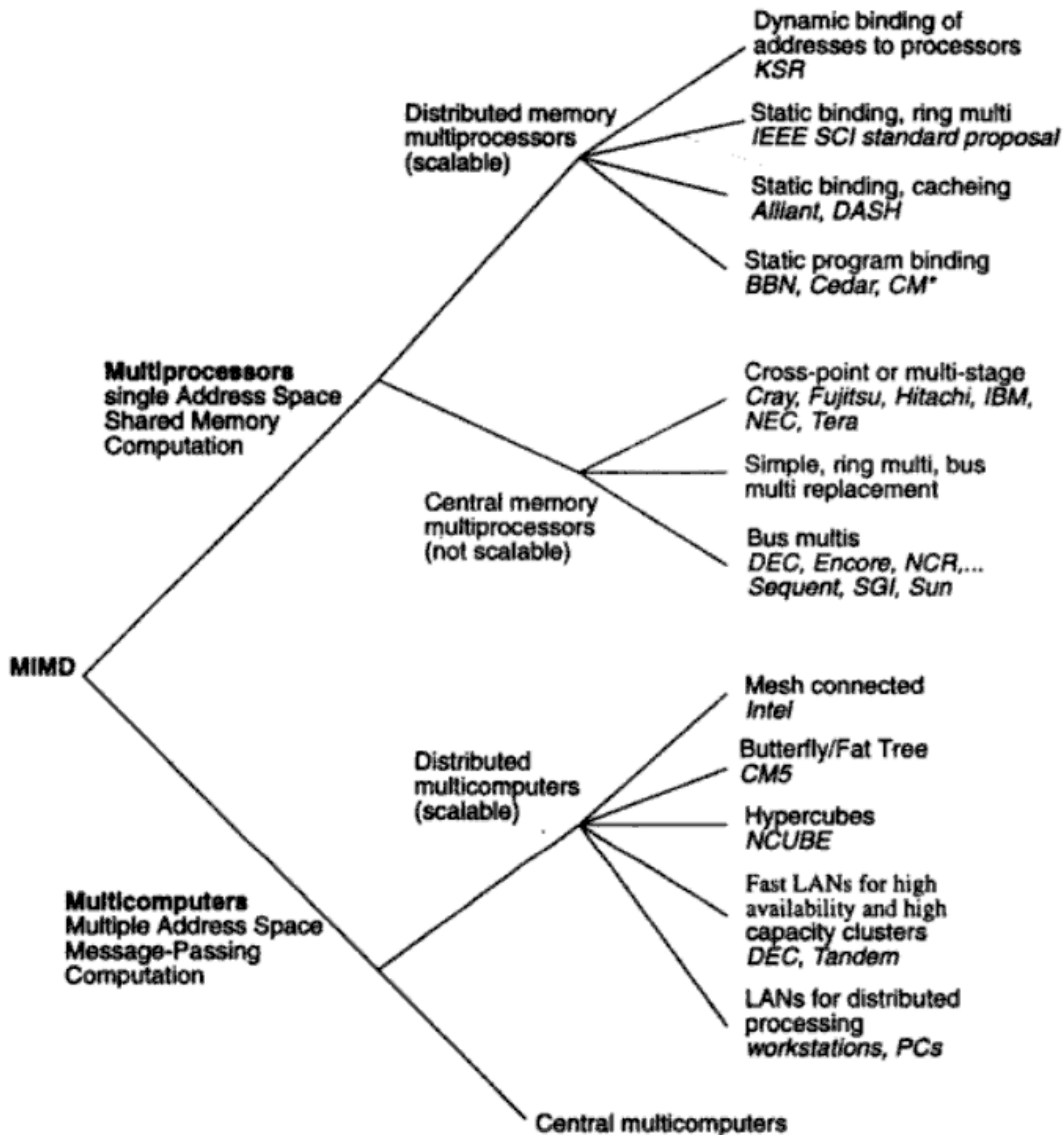


Figure 1.10 Bell's taxonomy of MIMD computers. (Courtesy of Gordon Bell; reprinted with permission from the *Communications of ACM*, August 1992)

and data are first loaded into the main memory through a host computer. All instructions are first decoded by the scalar control unit. If the decoded instruction is a scalar operation or a program control operation, it will be directly executed by the scalar processor using the scalar functional pipelines.

If the instruction is decoded as a vector operation, it will be sent to the vector control unit. This control unit will supervise the flow of vector data between the main memory and vector functional pipelines. The vector data flow is coordinated by the control unit. A number of vector functional pipelines may be built into a vector processor. Two pipeline vector supercomputer models are described below.

Copyrighted material

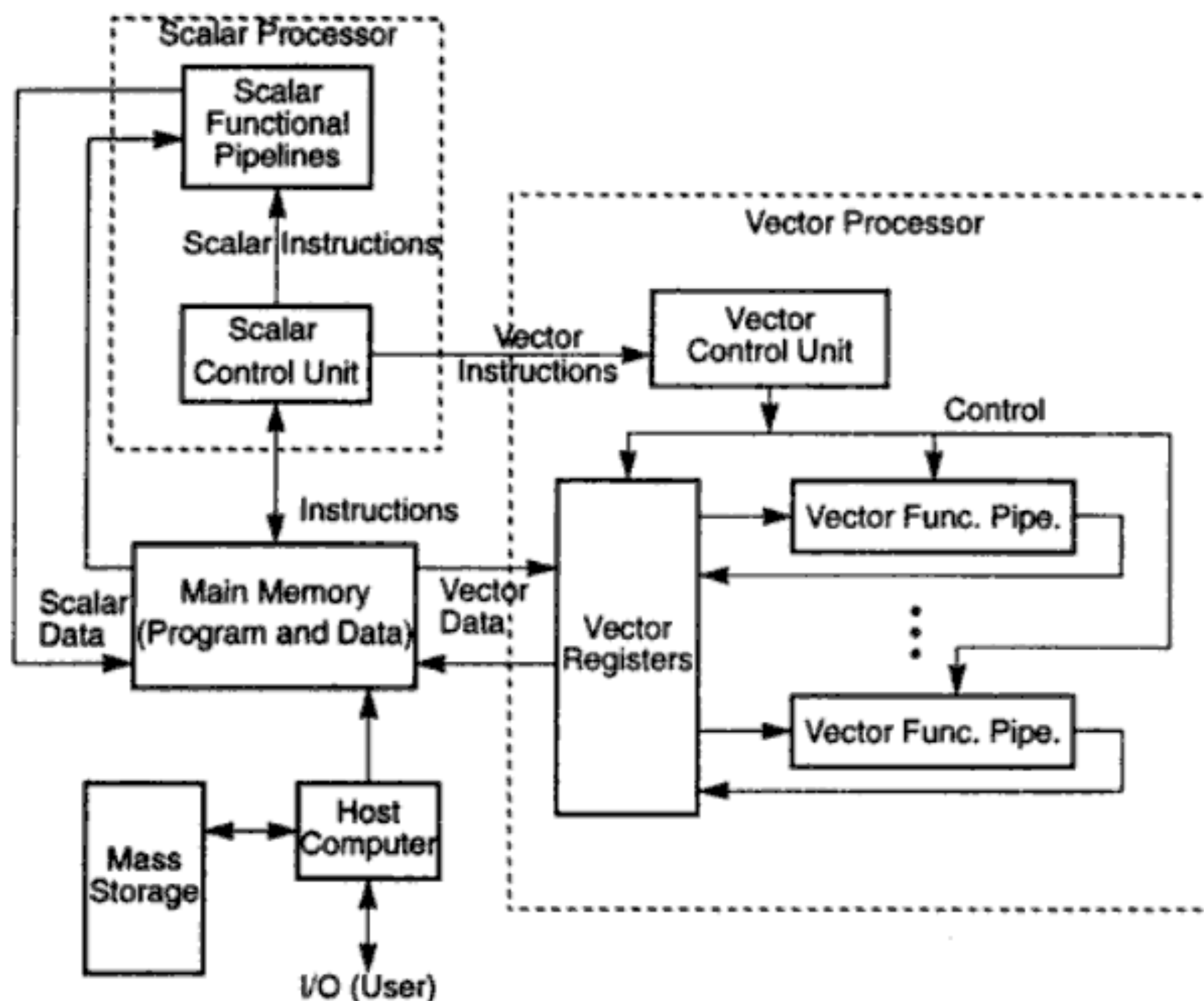


Figure 1.11 **The architecture of a vector supercomputer.**

Vector Processor Models Figure 1.11 shows a *register-to-register* architecture. Vector registers are used to hold the vector operands, intermediate and final vector results. The vector functional pipelines retrieve operands from and put results into the vector registers. All vector registers are programmable in user instructions. Each vector register is equipped with a component counter which keeps track of the component registers used in successive pipeline cycles.

The length of each vector register is usually fixed, say, sixty-four 64-bit component registers in a vector register in a Cray Series supercomputer. Other machines, like the Fujitsu VP2000 Series, use reconfigurable vector registers to dynamically match the register length with that of the vector operands.

In general, there are fixed numbers of vector registers and functional pipelines in a vector processor. Therefore, both resources must be reserved in advance to avoid resource conflicts between different vector operations. Several vector-register based supercomputers are summarized in Table 1.5.

A *memory-to-memory* architecture differs from a register-to-register architecture in the use of a vector stream unit to replace the vector registers. Vector operands and results are directly retrieved from the main memory in superwords, say, 512 bits as in the Cyber 205.

Pipelined vector supercomputers started with uniprocessor models such as the Cray 1 in 1976. Recent supercomputer systems offer both uniprocessor and multiprocessor models such as the Cray Y-MP Series. Most high-end mainframes offer multiprocessor

Copyrighted material

models with add-on vector hardware, such as the VAX 9000 and IBM 390/VF models.

Representative Supercomputers Over a dozen pipelined vector computers have been manufactured, ranging from workstations to mini- and supercomputers. Notable ones include the Stardent 3000 multiprocessor equipped with vector pipelines, the Convex C3 Series, the DEC VAX 9000, the IBM 390/VF, the Cray Research Y-MP family, the NEC SX Series, the Fujitsu VP2000, and the Hitachi S-810/20.

Table 1.5 Representative Vector Supercomputers

System Model	Vector Hardware Architecture and Capabilities	Compiler and Software Support
Convex C3800 family	GaAs-based multiprocessor with 8 processors and 500-Mbyte/s access port. 4 Gbytes main memory. 2 Gflops peak performance with concurrent scalar/vector operations.	Advanced C, Fortran, and Ada vectorizing and parallelizing compilers. Also support inter-procedural optimization, POSIX 1003.1/OS plus I/O interfaces and visualization system
Digital VAX 9000 System	Integrated vector processing in the VAX environment, 125–500 Mflops peak performance. 63 vector instructions. 16 x 64 x 64 vector registers. Pipeline chaining possible.	MS or ULTRIX/OS, VAX Fortran and VAX Vector Instruction Emulator (VVIEF) for vectorized program debugging.
Cray Research Y-MP and C-90	Y-MP runs with 2, 4, or 8 processors, 2.67 Gflop peak with Y-MP8256. C-90 has 2 vector pipes/CPU built with 10K gate ECL with 16 Gflops peak performance.	CF77 compiler for automatic vectorization, scalar optimization, and parallel processing. UNICOS improved from UNIX/V and Berkeley BSD/OS.

The Convex C1 and C2 Series were made with ECL/CMOS technologies. The latest C3 Series is based on GaAs technology.

The DEC VAX 9000 is Digital's largest mainframe system providing concurrent scalar/vector and multiprocessing capabilities. The VAX 9000 processors use a hybrid architecture. The vector unit is an optional feature attached to the VAX 9000 CPU. The Cray Y-MP family offers both vector and multiprocessing capabilities.

1.3.2 SIMD Supercomputers

In Fig. 1.3b, we have shown an abstract model of SIMD computers having a single instruction stream over multiple data streams. An operational model of SIMD

Copyrighted material

computers is presented below (Fig. 1.12) based on the work of H. J. Siegel (1979). Implementation models and case studies of SIMD machines are given in Chapter 8.

SIMD Machine Model An operational model of an SIMD computer is specified by a 5-tuple:

$$M = \langle N, C, I, M, R \rangle \tag{1.5}$$

where

(1) N is the number of *processing elements* (PEs) in the machine. For example, the Illiac IV has 64 PEs and the Connection Machine CM-2 uses 65,536 PEs.

(2) C is the set of instructions directly executed by the *control unit* (CU), including scalar and program flow control instructions.

(3) I is the set of instructions broadcast by the CU to all PEs for parallel execution. These include arithmetic, logic, data routing, masking, and other local operations executed by each active PE over data within that PE.

(4) M is the set of masking schemes, where each mask partitions the set of PEs into enabled and disabled subsets.

(5) R is the set of data-routing functions, specifying various patterns to be set up in the interconnection network for inter-PE communications.

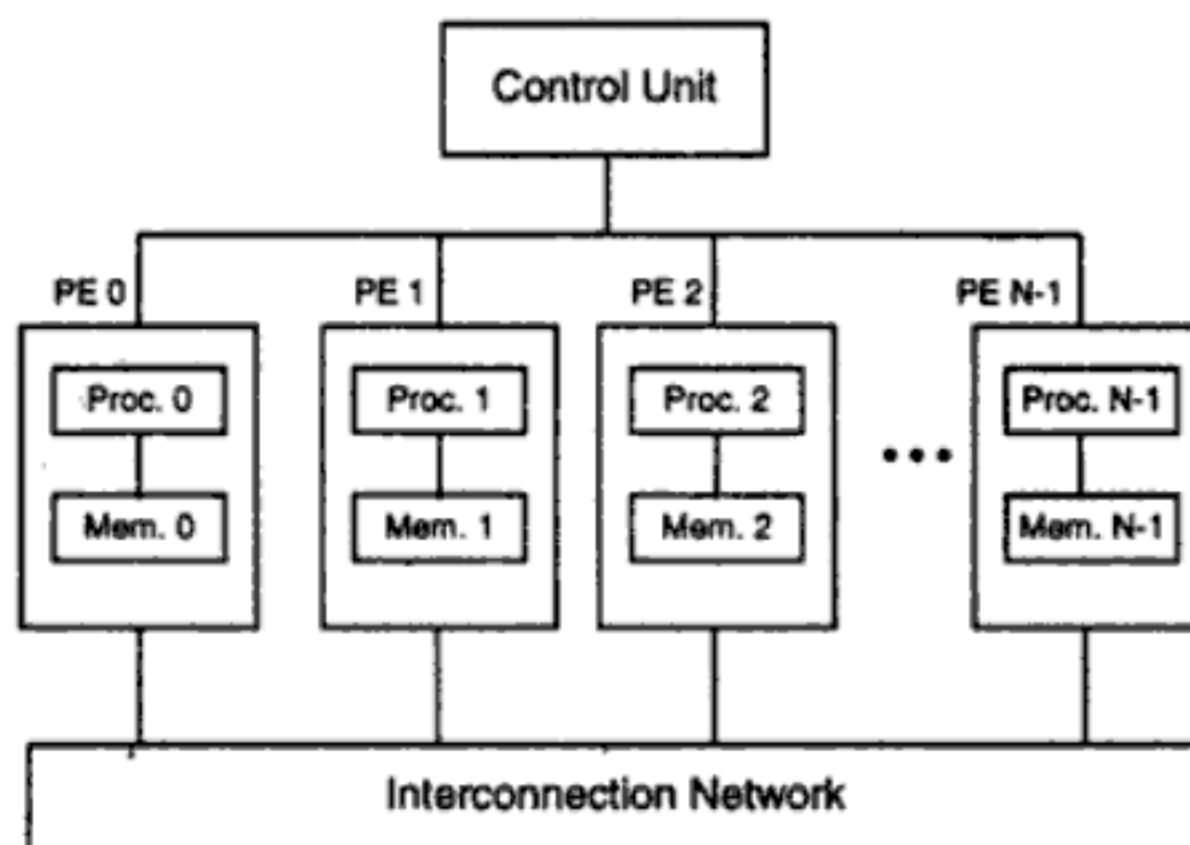


Figure 1.12 Operational model of SIMD computers.

One can describe a particular SIMD machine architecture by specifying the 5-tuple. An example SIMD machine is partially specified below:

Example 1.3 Operational specification of the MasPar MP-1 computer

We will study the detailed architecture of the MasPar MP-1 in Chapter 7. Listed below is a partial specification of the 5-tuple for this machine:

Copyrighted material

(1) The MP-1 is an SIMD machine with $N = 1024$ to 16,384 PEs, depending on which configuration is considered.

(2) The CU executes scalar instructions, broadcasts decoded vector instructions to the PE array, and controls the inter-PE communications.

(3) Each PE is a register-based load/store RISC processor capable of executing integer operations over various data sizes and standard floating-point operations. The PEs receive instructions from the CU.

(4) The masking scheme is built within each PE and continuously monitored by the CU which can set and reset the status of each PE dynamically at run time.

(5) The MP-1 has an X-Net mesh network plus a global multistage crossbar router for inter-CU-PE, X-Net nearest 8-neighbor, and global router communications.

■

Representative SIMD Computers Three SIMD supercomputers are summarized in Table 1.6. The number of PEs in these systems ranges from 4096 in the DAP610 to 16,384 in the MasPar MP-1 and 65,536 in the CM-2. Both the CM-2 and DAP610 are fine-grain, bit-slice SIMD computers with attached floating-point accelerators for blocks of PEs.

Each PE of the MP-1 is equipped with a 1-bit logic unit, 4-bit integer ALU, 64-bit mantissa unit, and 16-bit exponent unit. Therefore, the MP-1 is a medium-grain SIMD machine. Multiple PEs can be built on a single chip due to the simplicity of each PE. The MP-1 implements 32 PEs per chip with forty 32-bit registers per PE. The 32 PEs are interconnected by an *X-Net mesh*, which is a 4-neighbor mesh augmented with diagonal dual-stage links.

The CM-2 implements 16 PEs as a mesh on a single chip. Each 16-PE mesh chip is placed at one vertex of a 12-dimensional hypercube. Thus $16 \times 2^{12} = 2^{16} = 65,536$ PEs form the entire SIMD array.

The DAP610 implements 64 PEs as a mesh on a chip. Globally, a large mesh (64×64) is formed by interconnecting these small meshes on chips. Fortran 90 and modified versions of C, Lisp, and other synchronous programming languages have been developed to program SIMD machines.

1.4 PRAM and VLSI Models

Theoretical models of parallel computers are abstracted from the physical models studied in previous sections. These models are often used by algorithm designers and VLSI device/chip developers. The ideal models provide a convenient framework for developing parallel algorithms without worry about the implementation details or physical constraints.

The models can be applied to obtain theoretical performance bounds on parallel computers or to estimate VLSI complexity on chip area and execution time before the

Copyrighted material

Table 1.6 Representative SIMD Supercomputers

System Model	SIMD Machine Architecture and Capabilities	Languages, Compilers and Software Support
MasPar Computer Corporation MP-1 Family	Available in configurations from 1024 to 16,384 processors with 26,000 MIPS or 1.3 Gflops. Each PE is a RISC processor, with 16 Kbytes local memory. An X-Net mesh plus a multistage crossbar interconnect.	Fortran 77, MasPar Fortran (MPF), and MasPar Parallel Application Language; UNIX/OS with X-window, symbolic debugger, visualizers and animators.
Thinking Machines Corporation, CM-2	A bit-slice array of up to 65,536 PEs arranged as a 10-dimensional hypercube with 4 × 4 mesh on each vertex, up to 1M bits of memory per PE, with optional FPU shared between blocks of 32 PEs. 28 Gflops peak and 5.6 Gflops sustained.	Driven by a host of VAX, Sun, or Symbolics 3600, Lisp compiler, Fortran 90, C*, and *Lisp supported by PARIS
Active Memory Technology DAP600 Family	A fine-grain, bit-slice SIMD array of up to 4096 PEs interconnected by a square mesh with 1K bits per PE, orthogonal and 4-neighbor links, 20 GIPS and 560 Mflops peak performance.	Provided by host VAX/VMS or UNIX Fortran-plus or APAL on DAP, Fortran 77 or C on host. Fortran-plus affected Fortran 90 standards.

chip is fabricated. The abstract models are also useful in scalability and programmability analysis, when real machines are compared with an idealized parallel machine without worrying about communication overhead among processing nodes.

1.4.1 Parallel Random-Access Machines

Theoretical models of parallel computers are presented below. We define first the time and space complexities. Computational tractability is reviewed for solving difficult problems on computers. Then we introduce the *random-access machine* (RAM), *parallel random-access machine* (PRAM), and variants of PRAMs. These complexity models facilitate the study of asymptotic behavior of algorithms implementable on parallel computers.

Time and Space Complexities The complexity of an algorithm for solving a problem of size s on a computer is determined by the execution time and the storage space required. The *time complexity* is a function of the problem size. The time complexity function in order notation is the *asymptotic time complexity* of the algorithm. Usually, the worst-case time complexity is considered. For example, a time complexity $g(s)$ is said to be $O(f(s))$, read "order $f(s)$", if there exist positive constants c and s_0 such that $g(s) \leq cf(s)$ for all nonnegative values of $s > s_0$.

Copyrighted material

The *space complexity* can be similarly defined as a function of the problem size s. The *asymptotic space complexity* refers to the data storage of large problems. Note that the program (code) storage requirement and the storage for input data are not considered in this.

The time complexity of a serial algorithm is simply called *serial complexity*. The time complexity of a parallel algorithm is called *parallel complexity*. Intuitively, the parallel complexity should be lower than the serial complexity, at least asymptotically. We consider only *deterministic algorithms*, in which every operational step is uniquely defined in agreement with the way programs are executed on real computers.

A *nondeterministic algorithm* contains operations resulting in one outcome in a set of possible outcomes. There exist no real computers that can execute nondeterministic algorithms. Therefore, all algorithms (or machines) considered in this book are deterministic, unless otherwise noted.

NP-Completeness An algorithm has a *polynomial complexity* if there exists a polynomial $p(s)$ such that the time complexity is $O(p(s))$ for any problem size s. The set of problems having polynomial-complexity algorithms is called *P-class* (for polynomial class). The set of problems solvable by nondeterministic algorithms in polynomial time is called *NP-class* (for nondeterministic polynomial class).

Since deterministic algorithms are special cases of the nondeterministic ones, we know that $P \subset NP$. The P-class problems are computationally *tractable*, while the NP − P-class problems are *intractable*. But we do not know whether $P = NP$ or $P \neq NP$. This is still an open problem in computer science.

To simulate a nondeterministic algorithm with a deterministic algorithm may require exponential time. Therefore, intractable NP-class problems are also said to have exponential-time complexity.

Example 1.4 Polynomial- and exponential-complexity algorithms

Polynomial-complexity algorithms are known for sorting n numbers in $O(n \log n)$ time and for multiplication of two $n \times n$ matrices in $O(n^3)$ time. Therefore, both problems belong to the P-class.

Nonpolynomial algorithms have been developed for the traveling salesperson problem with complexity $O(n^2 2^n)$ and for the knapsack problem with complexity $O(2^{n/2})$. These complexities are *exponential*, greater than the polynomial complexities. So far, deterministic polynomial algorithms have not been found for these problems. Therefore, exponential-complexity problems belong to the NP-class.

■

Most computer scientists believe that $P \neq NP$. This leads to the conjecture that there exists a subclass, called *NP-complete* (NPC) problems, such that $NPC \subset NP$ but $NPC \cap P = \emptyset$ (Fig. 1.13). In fact, it has been proved that if any NP-complete problem is polynomial-time solvable, then one can conclude $P = NP$. Thus NP-complete problems are considered the hardest ones to solve. Only approximation algorithms were derived for solving some of the NP-complete problems.

Copyrighted material

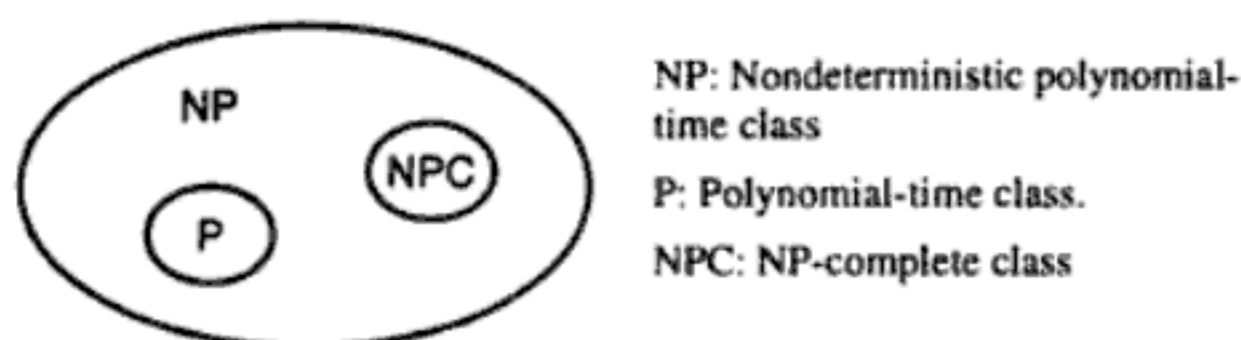


Figure 1.13 The relationships conjectured among the NP, P, and NPC classes of computational problems.

PRAM Models Conventional uniprocessor computers have been modeled as *random-access machines* (RAM) by Sheperdson and Sturgis (1963). A *parallel random-access machine* (PRAM) model has been developed by Fortune and Wyllie (1978) for modeling idealized parallel computers with zero synchronization or memory access overhead. This PRAM model will be used for parallel algorithm development and for scalability and complexity analysis.

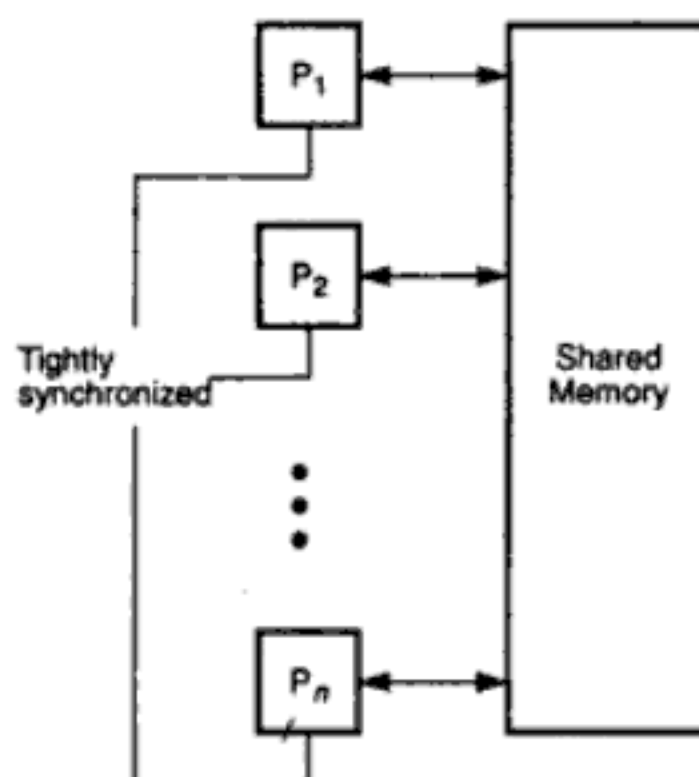


Figure 1.14 PRAM model of a multiprocessor system with shared memory, on which all n processors operate in lockstep in memory access and program execution operations. Each processor can access any memory location in unit time.

An n-processor PRAM (Fig. 1.14) has a globally addressable memory. The shared memory can be distributed among the processors or centralized in one place. The n processors [also called *processing elements* (PEs) by other authors] operate on a synchronized read-memory, compute, and write-memory cycle. With shared memory, the model must specify how concurrent read and concurrent write of memory are handled. Four memory-update options are possible:

- *Exclusive read* (ER) — This allows at most one processor to read from any memory location in each cycle, a rather restrictive policy.

Copyrighted material

- *Exclusive write* (EW) — This allows at most one processor to write into a memory location at a time.
- *Concurrent read* (CR) — This allows multiple processors to read the same information from the same memory cell in the same cycle.
- *Concurrent write* (CW) — This allows simultaneous writes to the same memory location. In order to avoid confusion, some policy must be set up to resolve the write conflicts.

Various combinations of the above options lead to several variants of the PRAM model as specified below. Since CR does not create a conflict problem, variants differ mainly in how they handle the CW conflicts.

PRAM Variants Described below are four variants of the PRAM model, depending on how the memory reads and writes are handled.

(1) The *EREW-PRAM model* — This model forbids more than one processor from reading or writing the same memory cell simultaneously (Snir, 1982; Karp and Ramachandran, 1988). This is the most restrictive PRAM model proposed.

(2) The *CREW-PRAM model* — The write conflicts are avoided by mutual exclusion. Concurrent reads to the same memory location are allowed.

(3) The *ERCW-PRAM model* — This allows exclusive read or concurrent writes to the same memory location.

(4) The *CRCW-PRAM model* — This model allows either concurrent reads or concurrent writes at the same time. The conflicting writes are resolved by one of the following four policies (Fortune and Wyllie, 1978):

  - *Common* — All simultaneous writes store the same value to the hot-spot memory location.
  - *Arbitrary* — Any one of the values written may remain; the others are ignored.
  - *Minimum* — The value written by the processor with the minimum index will remain.
  - *Priority* — The values being written are combined using some associative functions, such as summation or maximum.

Example 1.5 Multiplication of two $n \times n$ matrices in $O(\log n)$ time on a PRAM with $n^3/\log n$ processors (Viktor Prasanna, 1992)

Let A and B be the input matrices. Assume n^3 PEs are available initially. We later reduce the number of PEs to $n^3/\log n$. To visualize the algorithm, assume the memory is organized as a three-dimensional array with inputs A and B stored in two planes. Also, for sake of explanation, assume a three-dimensional indexing of the PEs. PE(i, j, k), $0 \leq k \leq n-1$ are used for computing the (i, j)th entry of the output matrix, $0 \leq i, j \leq n-1$, and n is a power of 2.

In step 1, n product terms corresponding to each output are computed using n PEs in $O(1)$ time. In step 2, these are added to produce an output in $O(\log n)$ time.

Copyrighted material

The total number of PEs used is n^3. The result is available in $C(\mathrm{i},\mathrm{j},0), 0 \le \mathrm{i},\mathrm{j} \le n-1$. Listed below are programs for each PE(i, j, k) to execute. All n^3 PEs operate in parallel for n^3 multiplications. But at most $n^3/2$ PEs are busy for $(n^3 - n^2)$ additions. Also, the PRAM is assumed to be CREW.

Step 1:
1. Read A(i, k)
2. Read B(k, j)
3. Compute A(i, k) × B(k, j)
4. Store in C(i, j, k)

Step 2:

```
1. ℓ ← n
2. Repeat
        ℓ ← ℓ/2
        if (k < ℓ) then
            begin
                Read C(i, j, k)
                Read C(i, j, k + ℓ)
                Compute C(i, j, k) + C(i, j, k + ℓ)
                Store in C(i, j, k)
            end
   until (ℓ = 1)
```

To reduce the number of PEs to $n^3/\log n$, use a PE array of size $n \times n \times n/\log n$. Each PE is responsible for computing $\log n$ product terms and summing them up. Step 1 can be easily modified to produce $n/\log n$ partial sums, each consisting of $\log n$ multiplications and $(\log n - 1)$ additions. Now we have an array C(i, j, k), $0 \le \mathrm{i},\mathrm{j} \le n-1, 0 \le \mathrm{k} \le n/\log n - 1$, which can be summed up in $\log(n/\log n)$ time. Combining the time spent in step 1 and step 2, we have a total execution time $2\log n - 1 + \log(n/\log n) \approx O(\log n)$ for large n.

■

Discrepancy with Physical Models PRAM models idealized parallel computers, in which all memory references and program executions by multiple processors are synchronized without extra cost. In reality, such parallel machines do not exist. An SIMD machine with shared memory is the closest architecture modeled by PRAM. However, PRAM allows different instructions to be executed on different processors simultaneously. Therefore, PRAM really operates in synchronized MIMD mode with a shared memory.

Among the four PRAM variants, the EREW and CRCW are the most popular models used. In fact, every CRCW algorithm can be simulated by an EREW algorithm. The CRCW algorithm runs faster than an equivalent EREW algorithm. It has been proved that the best n-processor EREW algorithm can be no more than $O(\log n)$ times slower than any n-processor CRCW algorithm.

Copyrighted material

The CREW model has received more attention in the literature than the ERCW model. The CREW models MISD machines, which have attracted little attention. For our purposes, we will use the CRCW-PRAM model unless otherwise stated. This particular model will be used in defining scalability in Chapter 3.

For complexity analysis or performance comparison, various PRAM variants offer an ideal model of parallel computers. Therefore, computer scientists use the PRAM model more often than computer engineers. In this book, we design parallel/vector computers using physical architectural models rather than PRAM models.

The PRAM model will be used for scalability and performance studies in Chapter 3 as a theoretical reference machine. For regularly structured parallelism, the PRAM can model much better than practical machine models. Therefore, sometimes PRAMs can indicate an upper bound on the performance of real parallel computers.

1.4.2 VLSI Complexity Model

Parallel computers rely on the use of VLSI chips to fabricate the major components such as processor arrays, memory arrays, and large-scale switching networks. An AT^2 model for two-dimensional VLSI chips is presented below, based on the work of Clark Thompson (1980). Three lower bounds on VLSI circuits are interpreted by Jeffrey Ullman (1984). The bounds are obtained by setting limits on memory, I/O, and communication for implementing parallel algorithms with VLSI chips.

The AT^2 Model Let A be the chip area and T be the latency for completing a given computation using a VLSI circuit chip. Let s by the problem size involved in the computation. Thompson stated in his doctoral thesis that for certain computations, there exists a lower bound $f(s)$ such that

$$A \times T^2 \geq O(f(s)) \tag{1.6}$$

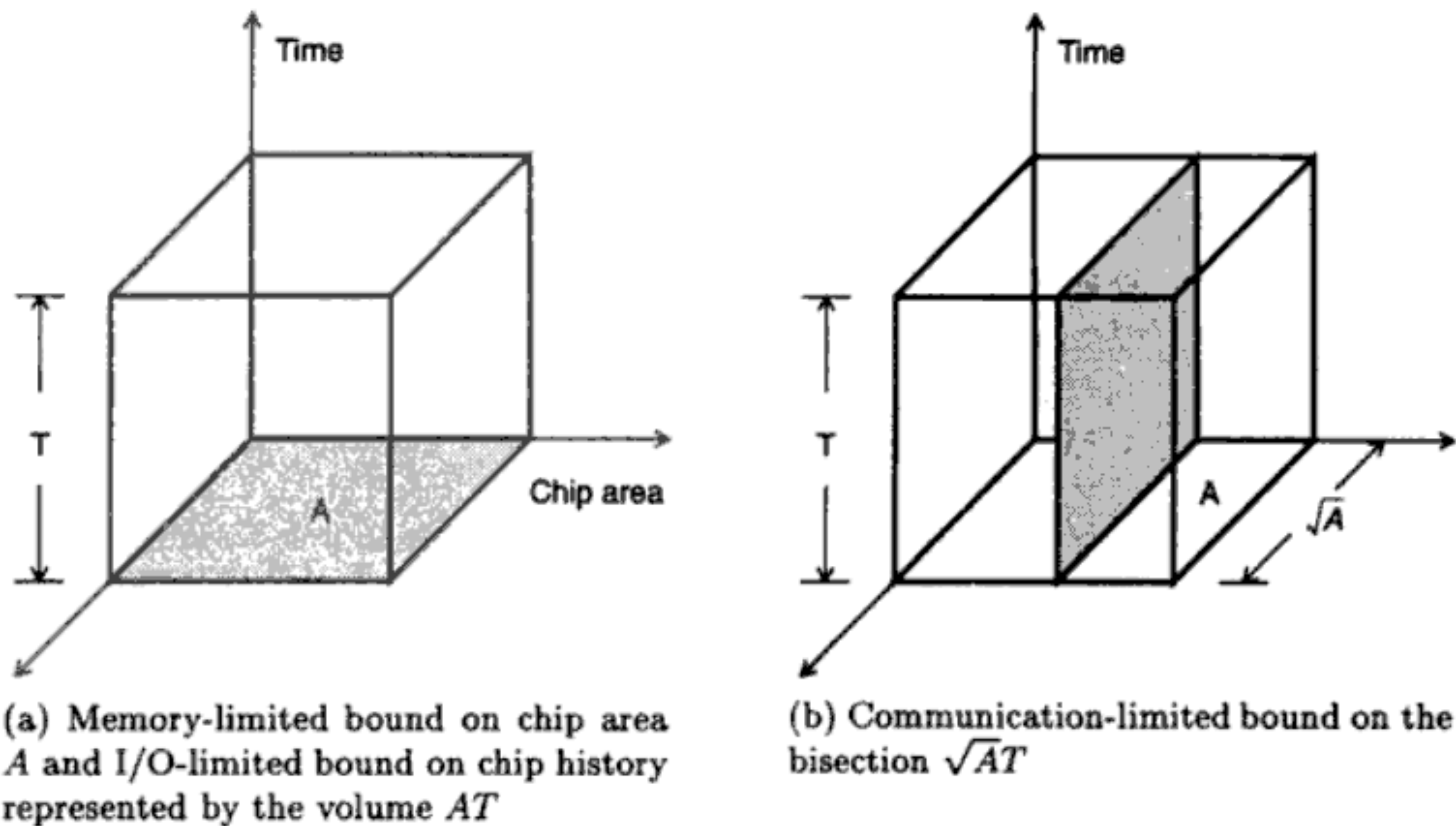
The chip area A is a measure of the chip's complexity. The latency T is the time required from when inputs are applied until all outputs are produced for a single problem instance. Figure 1.15 shows how to interpret the AT^2 complexity results in VLSI chip development. The chip is represented by the base area in the two horizontal dimensions. The vertical dimension corresponds to time. Therefore, the three-dimensional solid represents the history of the computation performed by the chip.

Memory Bound on Chip Area A There are many computations which are memory-bound, due to the need to process large data sets. To implement this type of computation in silicon, one is limited by how densely information (bit cells) can be placed on the chip. As depicted in Fig. 1.15a, the memory requirement of a computation sets a lower bound on the chip area A.

The amount of information processed by the chip can be visualized as information flow upward across the chip area. Each bit can flow through a unit area of the horizontal chip slice. Thus, the chip area bounds the amount of memory bits stored on the chip.

I/O Bound on Volume AT The volume of the rectangular cube is represented by

Copyrighted material



(a) Memory-limited bound on chip area A and I/O-limited bound on chip history represented by the volume AT

(b) Communication-limited bound on the bisection $\sqrt{A}T$

Figure 1.15 The $\mathbf{AT^2}$ complexity model of two-dimensional VLSI chips.

the product AT. As information flows through the chip for a period of time T, the number of input bits cannot exceed the volume. This provides an I/O-limited lower bound on the product AT, as demonstrated in Fig. 1.15a.

The area A corresponds to data into and out of the entire surface of the silicon chip. This areal measure sets the maximum I/O limit rather than using the peripheral I/O pads as seen in conventional chips. The height T of the volume can be visualized as a number of snapshots on the chip, as computing time elapses. The volume represents the amount of information flowing through the chip during the entire course of the computation.

Bisection Communication Bound, $\sqrt{\mathbf{A}}\mathbf{T}$ Figure 1.15b depicts a communication limited lower bound on the bisection area $\sqrt{A}T$. The bisection is represented by the vertical slice cutting across the shorter dimension of the chip area. The distance of this dimension is at most $\sqrt{A}$ for a square chip. The height of the cross section is T.

The bisection area represents the maximum amount of information exchange between the two halves of the chip circuit during the time period T. The cross-section area $\sqrt{A}T$ limits the communication bandwidth of a computation. VLSI complexity theoreticians have used the square of this measure, AT^2, as the lower bound.

Charles Seitz (1990) has given another interpretation of the AT^2 result. He considers the area-time product AT the cost of a computation, which can be expected to vary as $1/T$. This implies that the cost of computation for a two-dimensional chip decreases with the execution time allowed.

When three-dimensional (multilayer) silicon chips are used, Seitz asserted that the

Copyrighted material

cost of computation, as limited by volume-time product, would vary as $1/\sqrt{T}$. This is due to the fact that the bisection will vary as $(AT)^{2/3}$ for 3-D chips instead of as $\sqrt{AT}$ for 2-D chips.

Example 1.6 VLSI chip implementation of a matrix multiplication algorithm (Viktor Prasanna, 1992)

This example shows how to estimate the chip area A and compute time T for $n \times n$ matrix multiplication C = A × B on a mesh of processing elements (PEs) with a broadcast bus on each row and each column. The 2-D mesh architecture is shown in Fig. 1.16. Inter-PE communicaiton is done through the broadcast buses. We want to prove the bound $AT^2 = O(n^4)$ by developing a parallel matrix multiplication algorithm with time $T = O(n)$ in using the mesh with broadcast buses. Therefore, we need to prove that the chip area is bounded by $A = O(n^2)$.

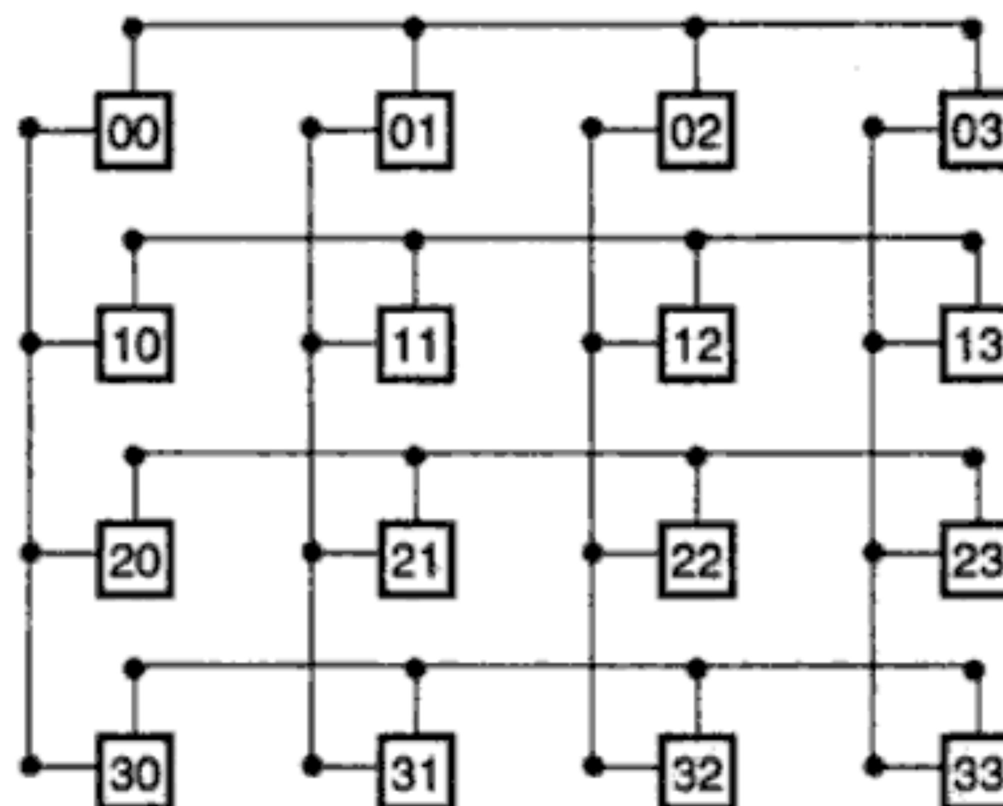


Figure 1.16 A 4 × 4 mesh of processing elements (PEs) with broadcast buses on each row and on each column. (Courtesy of Prasanna Kumar and Raghavendra; reprinted from *Journal of Parallel and Distributed Computing,* April 1987)

Each PE occupies a unit area, and the broadcast buses require $O(n^2)$ wire area. Thus the total chip area needed is $O(n^2)$ for an $n \times n$ mesh with broadcast buses. We show next that the $n \times n$ matrix multiplication can be performed on this mesh chip in $T = O(n)$ time. Denote the PEs as PE(i,j), $0 \le \mathrm{i,j} \le n-1$.

Initially the input matrix elements A(i, j) and B(i, j) are stored in PE(i, j) with no duplicated data. The memory is distributed among all the PEs. Each PE can access only its own local memory. The following parallel algorithm shows how to perform the dot-product operations in generating all the output elements $\mathrm{C(i,j)} = \sum_{\mathrm{k}=0}^{n-1} \mathrm{A(i,k)} \times \mathrm{B(k,j)}$ for $0 \le \mathrm{i,j} \le n-1$.

```
     Doall 10 for 0 ≤ i,j ≤ n − 1
10      PE(i,j) sets C(i,j) to 0 /Initialization/
```

Copyrighted material

```
        Do 50 for 0 ≤ k ≤ n − 1
            Doall 20 for 0 ≤ i ≤ n − 1
20              PE(i,k) broadcasts A(i,k) along its row bus
            Doall 30 for 0 ≤ j ≤ n − 1
30              PE(k,j) broadcasts B(k,j) along its column bus
                    /PE(i,j) now has A(i,k) and B(k,j), 0 ≤ i,j ≤ n − 1/
            Doall 40 for 0 ≤ i,j ≤ n − 1
40              PE(i,j) computes C(i,j) ← C(i,j) + A(i,k) × B(k,j)
50      Continue
```

The above algorithm has a sequential loop along the dimension indexed by k. It takes n time units (iterations) in this k-loop. Thus, we have $T = O(n)$. Therefore, $AT^2 = O(n^2) \cdot (O(n))^2 = O(n^4)$.

■

1.5 Architectural Development Tracks

The architectures of most existing computers follow certain development tracks. Understanding features of various tracks provides insights for new architectural development. We look into six tracks to be studied in later chapters. These tracks are distinguished by similarity in computational models and technological bases. We review below a few representative systems in each track.

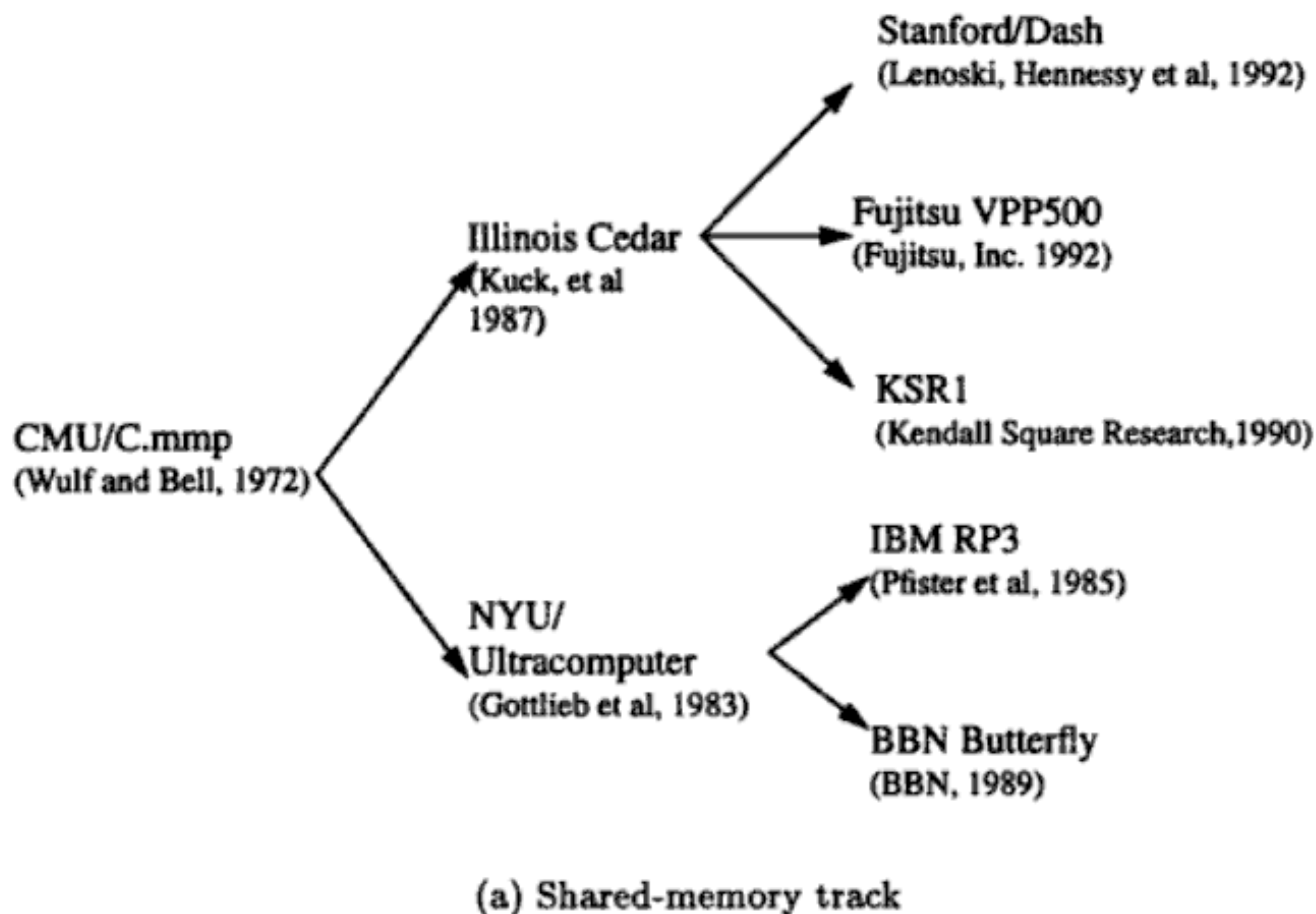
1.5.1 Multiple-Processor Tracks

Generally speaking, a multiple-processor system can be either a shared-memory multiprocessor or a distributed-memory multicomputer as modeled in Section 1.2. Bell listed these machines at the leaf nodes of the taxonomy tree (Fig. 1.10). Instead of a horizontal listing, we show a historical development along each important track of the taxonomy.

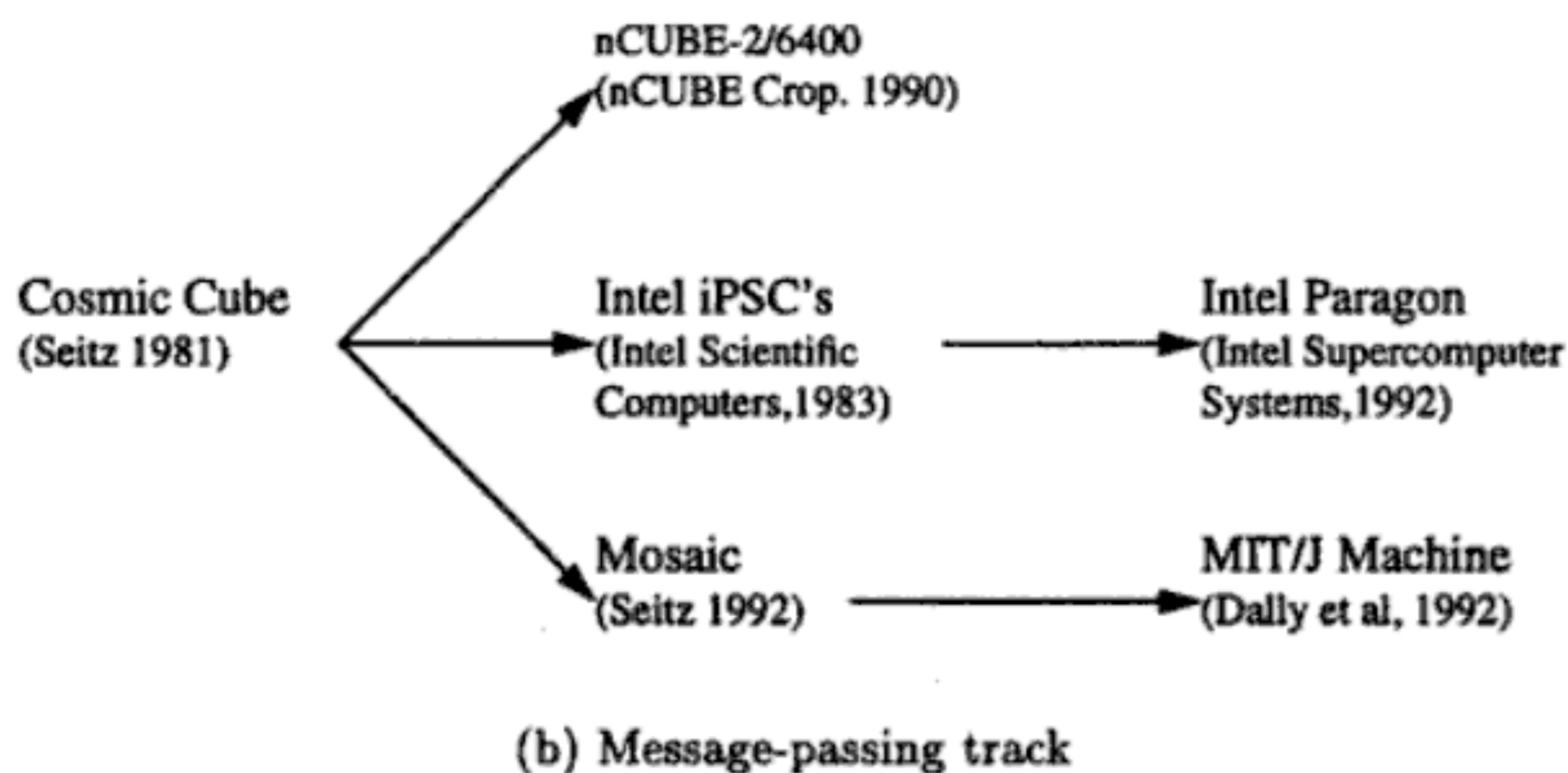
Shared-Memory Track Figure. 1.17a shows a track of multiprocessor development employing a single address space in the entire system. The track started with the C.mmp system developed at Carnegie-Mellon University (Wulf and Bell, 1972). The C.mmp was an UMA multiprocessor. Sixteen PDP 11/40 processors are interconnected to 16 shared-memory modules via a crossbar switch. A special interprocessor interrupt bus is provided for fast interprocess communication, besides the shared memory. The C.mmp project pioneered shared-memory multiprocessor development, not only in the crossbar architecture but also in the multiprocessor operating system (Hydra) development.

Both the NYU Ultracomputer project (Gottlieb et al., 1983) and the Illinois Cedar project (Kuck et al., 1987) were developed with a single address space. Both systems used multistage networks as a system interconnect. The major achievements in the Cedar project are in parallel compilers and performance benchmarking experiments. The Ultracomputer developed the combining network for fast synchronization among multiple processors to be studied in Chapter 7.

Copyrighted material



(a) Shared-memory track



(b) Message-passing track

Figure 1.17 Two multiple-processor tracks with and without shared memory.

The Stanford Dash (Lenoski, Hennessy et al., 1992) is a NUMA multiprocessor with distributed memories forming a global address space. Cache coherence is enforced with distributed directories. The KSR-1 is a typical COMA model. The Fujitsu VPP 500 is a 222-processor system with a crossbar interconnect. The shared memories are distributed to all processor nodes. We will study the Dash and the KSR-1 in Chapter 9 and the VPP500 in Chapter 8.

Following the Ultracomputer are two large-scale multiprocessors, both using multistage networks but with different interstage connections to be studied in Chapters 2 and 7. Among the systems listed in Fig. 1.17a, only the KSR-1, VPP500, and BBN Butterfly (BBN Advanced Computers, 1989) are commercial products. The rest are

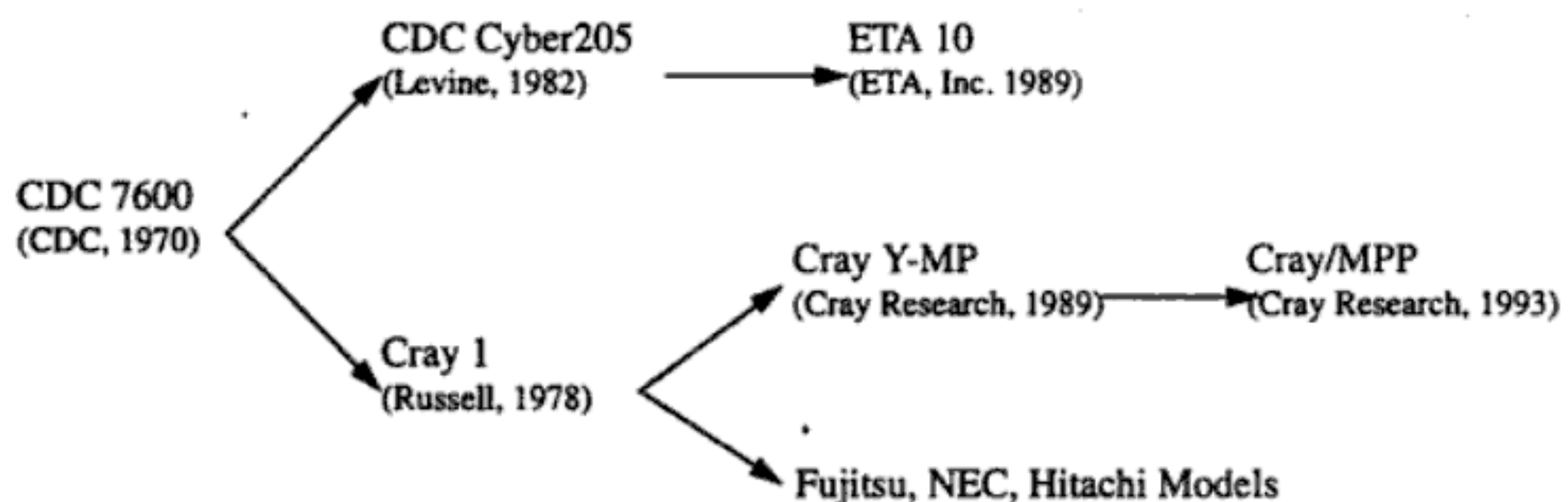
Copyrighted material

research systems; only prototypes have been built in laboratories.

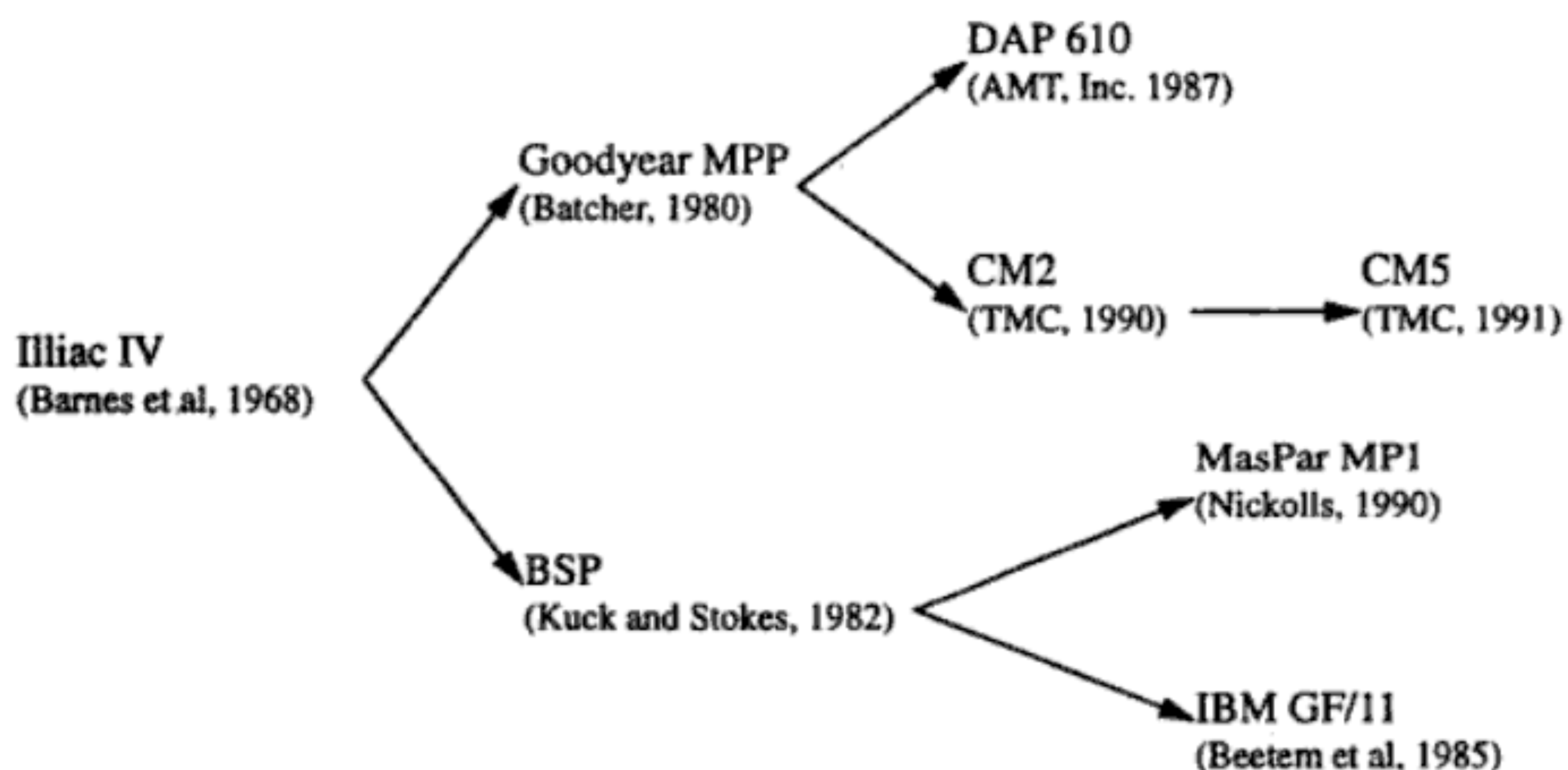
Message-Passing Track The Cosmic Cube (Seitz et al., 1981) pioneered the development of message-passing multicomputers (Fig. 1.17b). Since then, Intel has produced a series of medium-grain hypercube computers (the iPSCs). The nCUBE 2 also assumes a hypercube configuration. The latest Intel system is the Paragon (1992) to be studied in Chapter 7. On the research track, the Mosaic C (Seitz, 1992) and the MIT J-Machine (Dally et al., 1992) are two fine-grain multicomputers to be studied in Chapter 9.

1.5.2 Multivector and SIMD Tracks

The multivector track is shown in Fig. 1.18a, and the SIMD track in Fig. 1.18b.



(a) Multivector track



(b) SIMD track

Figure 1.18 Multivector and SIMD tracks.

Copyrighted material

Both tracks are used for concurrent scalar/vector processing. Detailed studies can be found in Chapter 8.

Multivector Track These are traditional vector supercomputers. The CDC 7600 was the first vector dual-processor system. Two subtracks were derived from the CDC 7600. The Cray and Japanese supercomputers all followed the register-to-register architecture. Cray 1 pioneered the multivector development in 1978. The latest Cray/MPP is a massively parallel system with distributed shared memory. It is supposed to work as a back-end accelerator engine compatible with the existing Cray Y-MP Series.

The other subtrack used memory-to-memory architecture in building vector supercomputers. We have identified only the CDC Cyber 205 and its successor the ETA10 here. Since the production of both machines has been discontinued now, we list them here simply for completeness in tracking different supercomputer architectures.

The SIMD Track The Illiac IV pioneered the construction of SIMD computers, even the array processor concept can be traced back far earlier to the 1960s. The subtrack, consisting of the Goodyear MPP, the AMT/DAP610, and the TMC/CM-2, are all SIMD machines built with bit-slice PEs. The CM-5 is a synchronized MIMD machine executing in a multiple-SIMD mode.

The other subtrack corresponds to medium-grain SIMD computers using word-wide PEs. The BSP (Kuck and Stokes, 1982) was a shared-memory SIMD machine built with 16 processors updating a group of 17 memory modules synchronously. The GF11 (Beetem et al., 1985) was developed at the IBM Watson Laboratory for scientific simulation research use. The MasPar MP1 is the only medium-grain SIMD computer currently in production use. We will describe the CM-2, MasPar MP1, and CM-5 in Chapter 8.

1.5.3 Multithreaded and Dataflow Tracks

These are two research tracks (Fig. 1.19) that have been mainly experimented with in laboratories. Both tracks will be studied in Chapter 9. The following introduction covers only basic definitions and milestone systems built today.

The conventional von Neumann machines are built with processors that execute a single context by each processor at a time. In other words, each processor maintains a single thread of control with limited hardware resources. In a multithreaded architecture, each processor can execute multiple contexts at the same time. The term *multithreading* implies that there are multiple threads of control in each processor. Multithreading offers an effective mechanism for hiding long latency in building large-scale multiprocessors.

As shown in Fig. 1.19a, the multithreading idea was pioneered by Burton Smith (1978) in the HEP system which extended the concept of scoreboarding of multiple functional units in the CDC 6400. The latest multithreaded multiprocessor projects are the Tera computer (Alverson, Smith et al., 1990) and the MIT Alewife (Agarwal et al., 1989) to be studied in Section 9.4. Until then, all multiprocessors studied use single-threaded processors as building blocks.

Copyrighted material

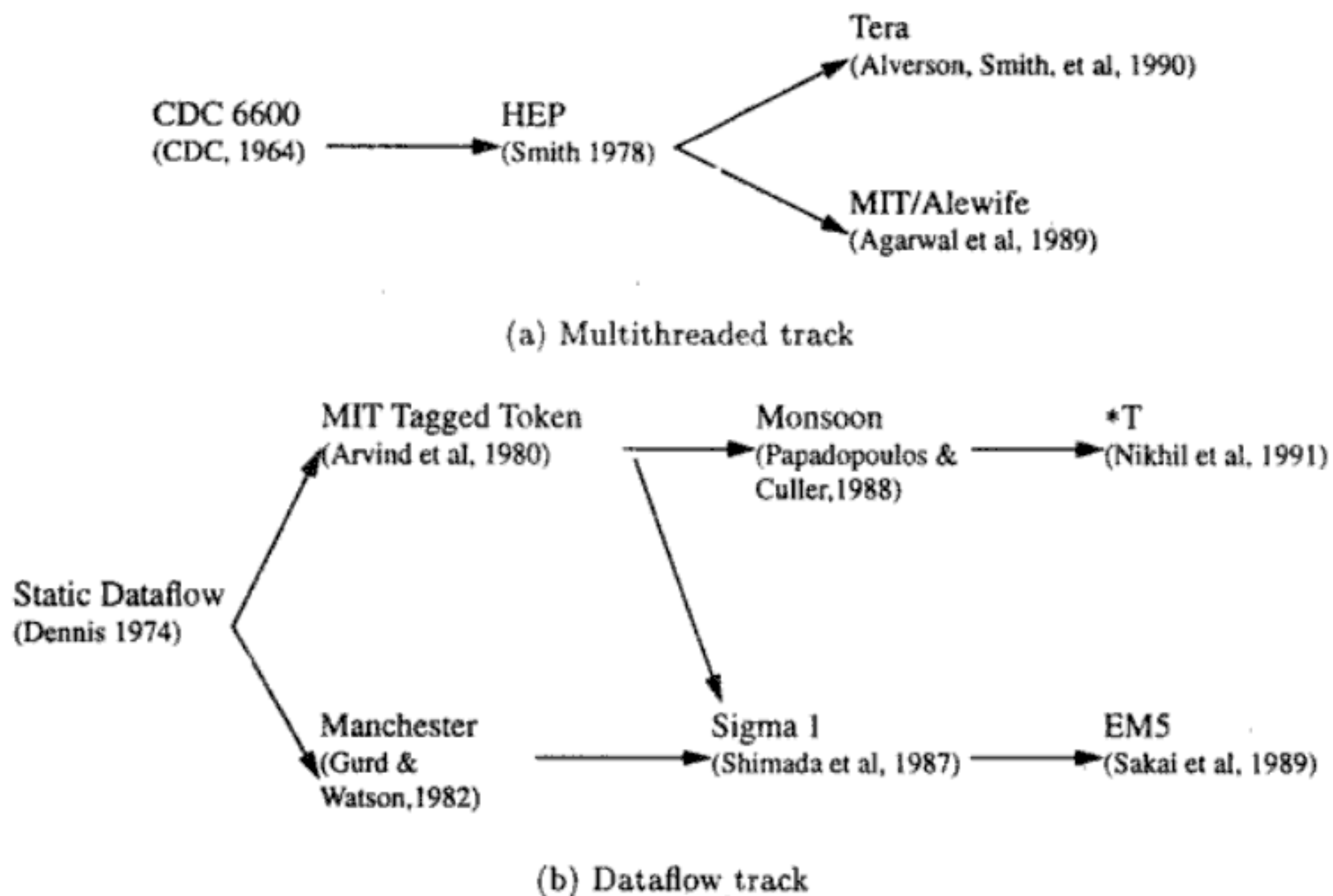


Figure 1.19 Multithreaded and dataflow tracks.

The Dataflow Track We will introduce the basic concepts of dataflow computers in Section 2.3. Some experimental dataflow systems are described in Section 9.5. The key idea is to use a dataflow mechanism, instead of a control-flow mechanism as in von Neumann machines, to direct the program flow. Fine-grain, instruction-level parallelism is exploited in dataflow computers.

As listed in Fig. 1.19b, the dataflow concept was pioneered by Jack Dennis (1974) with a "static" architecture. The concept later inspired the development of "dynamic" dataflow computers. A series of tagged-token architectures was developed at MIT by Arvind and coworkers. We will describe the tagged-token architecture in Section 2.3.1 and then the *T prototype (Nikhil et al., 1991) in Section 9.5.3.

Anther important subtrack of dynamic dataflow computer is represented by the Manchester machine (Gurd and Watson, 1982). The ETL Sigma 1 (Shimada et al., 1987) and EM5 have evolved from the MIT and Manchester machines. We will study the EM5 (Sakai et al., 1989) in Section 9.5.2. These dataflow machines are still in the research stage.

1.6 Bibliographic Notes and Exercises

Various machine architectures were classified by [Flynn72]. A recent collection of papers on architectural alternatives for exploiting parallelism can be found in the

Copyrighted material

tutorial [Lilja92]. [Almasi89] and Gottlieb provided a thorough survey of research done on parallel computing up to 1989. [Hwang89a] and DeGroot presented critical reviews on parallel processing techniques developed for supercomputers and artificial intelligence.

[Hennessy90] and Patterson provided an excellent treatment of uniprocessor computer design. For a treatment of earlier parallel processing computers, readers are referred to [Hwang84] and Briggs. The layered classification of parallel computers was proposed in [Ni91]. [Bell92] introduced the MIMD taxonomy. Systolic array was introduced by [Kung78] and Leiserson. [Prasanna Kumar87] and Raghavendra proposed the mesh architecture with broadcast buses for parallel computing.

Multiprocessor issues were characterized by [Gajski85] and Pier and by [Dubois88], Scheurich, and Briggs. Multicomputer technology was assessed by [Athas88] and Seitz. An introduction to existing parallel computers can be found in [Trew91] and Wilson. Key references of various computer systems, listed in Bell's taxonomy (Fig. 1.10) and in different architectural development tracks (Figs. 1.17 through 1.19), are identified in the figures as well as in the bibliography. Additional references to these case-study machines can be found in later chapters. A collection of reviews, a bibliography, and indexes of resources in parallel systems can be found in [ACM91] with an introduction by Charles Seitz.

A comparative study of NUMA and COMA multiprocessor models can be found in [Stenström92], Joe, and Gupta. SIMD machines were modeled by [Siegel79]. The RAM model was proposed by [Sheperdson63] and Sturgis. The PRAM model and variations were studied in [Fortune78] and Wyllie, [Snir82], and [Karp88]. The theory of NP-completeness is covered in the book by [Cormen90], Leiserson, and Rivest. The VLSI complexity model was introduced in [Thompson80] and interpreted by [Ullman84] and by [Seitz90] subsequently.

Readers are referred to the following journals and conference records for information on recent developments:

- *Journal of Parallel and Distributed Computing* (Academic Press, since 1983).
- *Journal of Parallel Computing* (North Holland, Amsterdam, since 1984).
- *IEEE Transactions on Parallel and Distributed Systems* (IEEE Computer Society, since 1990).
- *International Conference on Parallel Processing* (Pennsylvania State University, since 1972).
- *International Symposium on Computer Architecture* (IEEE Computer Society, since 1972).
- *Symposium on the Frontiers of Massively Parallel Computation* (IEEE Computer Society, since 1986).
- *International Conference on Supercomputing* (ACM, since 1987).
- *Symposium on Architectural Support for Programming Languages and Operating Systems* (ACM, since 1975).
- *Symposium on Parallel Algorithms and Architectures* (ACM, since 1989).
- *International Parallel Processing Symposium* (IEEE Computer Society, since 1986).

Copyrighted material

- *IEEE Symposium on Parallel and Distributed Processing* (IEEE Computer Society, since 1989).

Exercises

Problem 1.1 A 40-MHz processor was used to execute a benchmark program with the following instruction mix and clock cycle counts:

Instruction type	Instruction count	Clock cycle count
Integer arithmetic	45000	1
Data transfer	32000	2
Floating point	15000	2
Control transfer	8000	2

Determine the effective CPI, MIPS rate, and execution time for this program.

Problem 1.2 Explain how instruction set, compiler technology, CPU implementation and control, and cache and memory hierarchy affect the CPU performance and justify the effects in terms of program length, clock rate, and effective CPI.

Problem 1.3 A workstation uses a 15-MHz processor with a claimed 10-MIPS rating to execute a given program mix. Assume a one-cycle delay for each memory access.

(a) What is the effective CPI of this computer?

(b) Suppose the processor is being upgraded with a 30-MHz clock. However, the speed of the memory subsystem remains unchanged, and consequently two clock cycles are needed per memory access. If 30% of the instructions require one memory access and another 5% require two memory accesses per instruction, what is the performance of the upgraded processor with a compatible instruction set and equal instruction counts in the given program mix?

Problem 1.4 Consider the execution of an object code with 200,000 instructions on a 40-MHz processor. The program consists of four major types of instructions. The instruction mix and the number of cycles (CPI) needed for each instruction type are given below based on the result of a program trace experiment:

Instruction type	CPI	Instruction mix
Arithmetic and logic	1	60%
Load/store with cache hit	2	18%
Branch	4	12%
Memory reference with cache miss	8	10%

Copyrighted material

(a) Calculate the average CPI when the program is executed on a uniprocessor with the above trace results.

(b) Calculate the corresponding MIPS rate based on the CPI obtained in part (a).

Problem 1.5 Indicate whether each of the following statements is *true* or *false* and justify your answer with reasoning and supportive or counter examples:

(a) The CPU computations and I/O operations cannot be overlapped in a multiprogrammed computer.

(b) Synchronization of all PEs in an SIMD computer is done by hardware rather than by software as is often done in most MIMD computers.

(c) As far as programmability is concerned, shared-memory multiprocessors offer simpler interprocessor communication support than that offered by a message-passing multicomputer.

(d) In an MIMD computer, all processors must execute the same instruction at the same time synchronously.

(e) As far as scalability is concerned, multicomputers with distributed memory are more scalable than shared-memory multiprocessors.

Problem 1.6 The execution times (in seconds) of four programs on three computers are given below:

	Execution Time (in seconds)		
Program	Computer A	Computer B	Computer C
Program 1	1	10	20
Program 2	1000	100	20
Program 3	500	1000	50
Program 4	100	800	100

Assume that 100,000,000 instructions were executed in each of the four programs. Calculate the MIPS rating of each program on each of the three machines. Based on these ratings, can you draw a clear conclusion regarding the relative performance of the three computers? Give reasons if you find a way to rank them statistically.

Problem 1.7 Characterize the architectural operations of SIMD and MIMD computers. Distinguish between multiprocessors and multicomputers based on their structures, resource sharing, and interprocessor communications. Also, explain the differences among UMA, NUMA, and COMA, and NORMA computers.

Problem 1.8 The following code segment, consisting of six instructions, needs to be executed 64 times for the evaluation of vector arithmetic expression: $D(I) = A(I) + B(I) \times C(I)$ for $0 \leq I \leq 63$.

Load R1, B(I) $\quad$ $/R1 \leftarrow \text{Memory}\ (\alpha + I)/$

Copyrighted material

Load R2, C(I)	/R2 $\leftarrow$ Memory $(\beta + \text{I})$/
Multiply R1, R2	/R1 $\leftarrow$ (R1) $\times$ (R2)/
Load R3, A(I)	/R3 $\leftarrow$ Memory $(\gamma + \text{I})$/
Add R3, R1	/R3 $\leftarrow$ (R3) + (R1)/
Store D(I), R3	/Memory $(\theta + \text{I}) \leftarrow$ (R3)/

where R1, R2, and R3 are CPU registers, (R1) is the content of R1, α, β, γ, and θ are the starting memory addresses of arrays B(I), C(I), A(I), and D(I), respectively. Assume four clock cycles for each Load or Store, two cycles for the Add, and eight cycles for the Multiply on either a uniprocessor or a single PE in an SIMD machine.

(a) Calculate the total number of CPU cycles needed to execute the above code segment repeatedly 64 times on an SISD uniprocessor computer sequentially, ignoring all other time delays.

(b) Consider the use of an SIMD computer with 64 PEs to execute the above vector operations in six synchronized vector instructions over 64-component vector data and both driven by the same-speed clock. Calculate the total execution time on the SIMD machine, ignoring instruction broadcast and other delays.

(c) What is the speedup gain of the SIMD computer over the SISD computer?

Problem 1.9 Prove that the best parallel algorithm written for an n-processor EREW-PRAM model can be no more than $O(\log n)$ times slower than any algorithm for a CRCW model of PRAM having the same number of processors.

Problem 1.10 Consider the multiplication of two n-bit binary integers using a 1.2-μm CMOS multiplier chip. Prove the lower bound $AT^2 > kn^2$, where A is the chip area, T is the execution time, n is the word length, and k is a technology-dependent constant.

Problem 1.11 Compare the PRAM models with physical models of real parallel computers in each of the following categories:

(a) Which PRAM variant can best model SIMD machines and how?

(b) Repeat the question in part (a) for shared-memory MIMD machines.

Problem 1.12 Answer the following questions related to the architectural development tracks presented in Section 1.5:

(a) For the shared-memory track (Fig. 1.17), explain the trend in physical memory organizations from the earlier system (C.mmp) to more recent systems (such as Dash, etc.).

(b) Distinguish between medium-grain and fine-grain multicomputers in their architectures and programming requirements.

(c) Distinguish between register-to-register and memory-to-memory architectures for building conventional multivector supercomputers.

(d) Distinguish between single-threaded and multithreaded processor architectures.

Copyrighted material

Problem 1.13 Design an algorithm to find the maximum of n numbers in $O(\log n)$ time on an EREW-PRAM model. Assume that initially each location holds one input value. Explain how you would make the algorithm processor time optimal.

Problem 1.14 Develop two algorithms for fast multiplication of two $n \times n$ matrices with a system of p processors, where $1 \leq p \leq n^3/\log n$. Choose an appropriate PRAM machine model to prove that the matrix multiplication can be done in $T = O(n^3/p)$ time.

(a) Prove that $T = O(n^2)$ if $p = n$. The corresponding algorithm must be shown, similar to that in Example 1.5.

(b) Show the parallel algorithm with $T = O(n)$ if $p = n^2$.

Problem 1.15 Match each of the following eight computer systems: KSR-1, RP3, Paragon, Dash, CM-2, VPP500, EM-5, and Tera, with one of the best descriptions listed below. The mapping is a one-to-one correspondence.

(a) A massively parallel system built with multiple-context processors and a 3-D torus architecture.

(b) A data-parallel computer built with bit-slice PEs interconnected by a hypercube/mesh network.

(c) A ring-connected multiprocessor using a cache-only memory architecture.

(d) An experimental multiprocessor built with a dynamic dataflow architecture.

(e) A crossbar-connected multiprocessor built with distributed processor/memory nodes forming a single address space.

(f) A multicomputer built with commercial microprocessors with multiple address spaces.

(g) A scalable multiprocessor built with distributed shared memory and coherent caches.

(h) An MIMD computer built with a large multistage switching network.

Copyrighted material

Chapter 2

Program and Network Properties

This chapter covers fundamental properties of program behavior and introduces major classes of interconnection networks. We begin with a study of computational granularity, conditions for program partitioning, matching software with hardware, program flow mechanisms, and compilation support for parallelism. Interconnection architectures introduced include static and dynamic networks. Network complexity, communication bandwidth, and data-routing capabilities are discussed.

2.1 Conditions of Parallelism

The exploitation of parallelism has created a new dimension in computer science. In order to move parallel processing into the mainstream of computing, H.T. Kung (1991) has identified the need to make significant progress in three key areas: *computation models* for parallel computing, *interprocessor communication* in parallel architectures, and *system integration* for incorporating parallel systems into general computing environments.

A theoretical treatment of parallelism is thus needed to build a basis for the above challenges. In practice, parallelism appears in various forms in a computing environment. All forms can be attributed to levels of parallelism, computational granularity, time and space complexities, communication latencies, scheduling policies, and load balancing. Very often, tradeoffs exist among time, space, performance, and cost factors.

2.1.1 Data and Resource Dependences

The ability to execute several program segments in parallel requires each segment to be independent of the other segments. The independence comes in various forms as defined below separately. For simplicity, to illustrate the idea, we consider the dependence relations among instructions in a program. In general, each code segment may contain one or more statements.

We use a *dependence graph* to describe the relations. The nodes of a dependence graph correspond to the program statements (instructions), and the directed edges

Copyrighted material

with different labels show the ordered relations among the statements. The analysis of dependence graphs shows where opportunity exists for parallelization and vectorization.

Data Dependence The ordering relationship between statements is indicated by the data dependence. Five types of data dependence are defined below:

(1) *Flow dependence*: A statement S2 is *flow-dependent* on statement S1 if an execution path exists from S1 to S2 and if at least one output (variables assigned) of S1 feeds in as input (operands to be used) to S2. Flow dependence is denoted as S1 $\longrightarrow$ S2.

(2) *Antidependence*: Statement S2 is *antidependent* on statement S1 if S2 follows S1 in program order and if the output of S2 overlaps the input to S1. A direct arrow crossed with a bar as in S1 $\nrightarrow$ S2 indicates antidependence from S1 to S2.

(3) *Output dependence*: Two statements are *output-dependent* if they produce (write) the same output variable. S1 $\circ\!\!\rightarrow$ S2 indicates output dependence from S1 to S2.

(4) *I/O dependence*: Read and write are I/O statements. I/O dependence occurs not because the same variable is involved but because the same file is referenced by both I/O statements.

(5) *Unknown dependence*: The dependence relation between two statements cannot be determined in the following situations:

- The subscript of a variable is itself subscribed (indirect addressing).
- The subscript does not contain the loop index variable.
- A variable appears more than once with subscripts having different coefficients of the loop variable.
- The subscript is nonlinear in the loop index variable.

When one or more of these conditions exist, a conservative assumption is to claim unknown dependence among the statements involved.

Example 2.1 Data dependence in programs

Consider the following code fragment of four instructions:

S1:	Load R1, A	/R1 ← Memory(A)/
S2:	Add R2, R1	/R2 ← (R1) + (R2) /
S3:	Move R1, R3	/R1 ← (R3)/
S4:	Store B, R1	/Memory(B) ← (R1)/

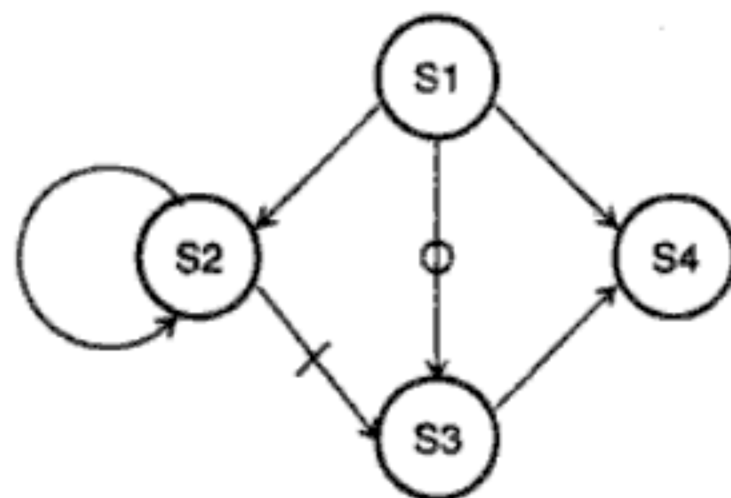
As illustrated in Fig. 2.1a, S2 is flow-dependent on S1 because the variable A is passed via the register R1. S3 is antidependent on S2 because of potential conflicts in register content in R1. S3 is output-dependent on S1 because they both modify the same register R1. Other data dependence relationships can be similarly revealed on a pairwise basis. Note that dependence is a partial ordering relation; that is, the members of not every pair of statements are related. For example, the statements S2 and S4 in the above program are totally *independent*.

Next, we consider a code fragment involving I/O operations:

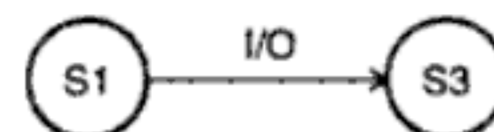
Copyrighted material

S1:	Read (4), A(I)	/Read array A from tape unit 4/
S2:	Rewind(4)	/Rewind tape unit 4 /
S3:	Write (4), B(I)	/Write array B into tape unit 4/
S4:	Rewind (4)	/Rewind tape unit 4/

As shown in Fig. 2.1b, the read/write statements, S1 and S3, are I/O-dependent on each other because they both access the same file from tape unit 4. The above data dependence relations should not be arbitrarily violated during program execution. Otherwise, erroneous results may be produced with changed program order. The order in which statements are executed in a program is often well defined. Repetitive runs should produce identical results. On a multiprocessor system, the program order may or may not be preserved, depending on the memory model used. Determinism yielding predictable results can be controlled by a programmer as well as by constrained modification of writable data in a shared memory.



(a) Dependence graph



(b) I/O dependence caused by accessing the same file by the read and write statements

Figure 2.1 Data and I/O dependences in the program of Example 2.1.

■

Control Dependence This refers to the situation where the order of execution of statements cannot be determined before run time. For example, conditional statements (IF in Fortran) will not be resolved until run time. Different paths taken after a conditional branch may introduce or eliminate data dependence among instructions. Dependence may also exist between operations performed in successive iterations of a looping procedure. In the following, we show one loop example with and another without control-dependent iterations. The successive iterations of the following loop are *control-independent*:

```
    Do 20 I = 1, N
       A(I) = C(I)
       IF (A(I) .LT. 0) A(I) = 1
20  Continue
```

The following loop has *control-dependent* iterations:

Copyrighted material

```
      Do 10 I = 1, N
         IF (A(I - 1) .EQ. 0) A(I) = 0
   10 Continue
```

Control dependence often prohibits parallelism from being exploited. Compiler techniques are needed to get around the control dependence in order to exploit more parallelism.

Resource Dependence This is different from data or control dependence, which demands the independence of the work to be done. *Resource dependence* is concerned with the conflicts in using shared resources, such as integer units, floating-point units, registers, and memory areas, among parallel events. When the conflicting resource is an ALU, we call it *ALU dependence*.

If the conflicts involve workplace storage, we call it *storage dependence*. In the case of storage dependence, each task must work on independent storage locations or use protected access (such as locks or monitors to be described in Chapter 11) to shared writable data.

The transformation of a sequentially coded program into a parallel executable form can be done manually by the programmer using explicit parallelism, or by a compiler detecting implicit parallelism automatically. In both approaches, the decomposition of programs is the primary objective.

Program partitioning determines whether a given program can be partitioned or split into pieces that can execute in parallel or follow a certain prespecified order of execution. Some programs are inherently sequential in nature and thus cannot be decomposed into parallel branches. The detection of parallelism in programs requires a check of the various dependence relations.

Bernstein's Conditions In 1966, Bernstein revealed a set of conditions based on which two processes can execute in parallel. A *process* is a software entity corresponding to the abstraction of a program fragment defined at various processing levels. We define the *input set* I_i of a process P_i as the set of all input variables needed to execute the process.

Similarly, the *output set* O_i consists of all output variables generated after execution of the process P_i. Input variables are essentially operands which can be fetched from memory or registers, and output variables are the results to be stored in working registers or memory locations.

Now, consider two processes P_1 and P_2 with their input sets I_1 and I_2 and output sets O_1 and O_2, respectively. These two processes can execute in parallel and are denoted $P_1 \parallel P_2$ if they are independent and do not create confusing results.

Formally, these conditions are stated as follows:

$$\left.\begin{array}{rcl} I_1 \cap O_2 & = & \emptyset \\ I_2 \cap O_1 & = & \emptyset \\ O_1 \cap O_2 & = & \emptyset \end{array}\right\} \tag{2.1}$$

Copyrighted material

These three equations are known as *Bernstein's conditions*. The input set I_i is also called the *read set* or the *domain* of P_i by other authors. Similarly, the output set O_i has been called the *write set* or the *range* of a process P_i. In terms of data dependences, Bernstein's conditions simply imply that two processes can execute in parallel if they are flow-independent, antiindependent, and output-independent.

The parallel execution of two processes produces the same results regardless of whether they are executed sequentially in any order or in parallel. This is possible only if the output of one process will not be used as input to the other process. Furthermore, the two processes will not modify (write) the same set of variables, either in memory or in the registers.

In general, a set of processes, $P_1, P_2, ..., P_k$, can execute in parallel if Bernstein's conditions are satisfied on a pairwise basis; that is, $P_1 \parallel P_2 \parallel P_3 \parallel \cdots \parallel P_k$ if and only if $P_i \parallel P_j$ for all $i \neq j$. This is exemplified by the following program illustrated in Fig. 2.2.

Example 2.2 Detection of parallelism in a program using Bernstein's conditions

Consider the simple case in which each process is a single HLL statement. We want to detect the parallelism embedded in the following five instructions labeled P_1, P_2, P_3, P_4, and P_5 in program order.

$$\left.\begin{array}{lcl} P_1: & C & = & D \times E \\ P_2: & M & = & G + C \\ P_3: & A & = & B + C \\ P_4: & C & = & L + M \\ P_5: & F & = & G \div E \end{array}\right\} \qquad (2.2)$$

Assume that each statement requires one step to execute. No pipelining is considered here. The dependence graph shown in Fig. 2.2a demonstrates flow dependence as well as resource dependence. In sequential execution, five steps are needed (Fig. 2.2b).

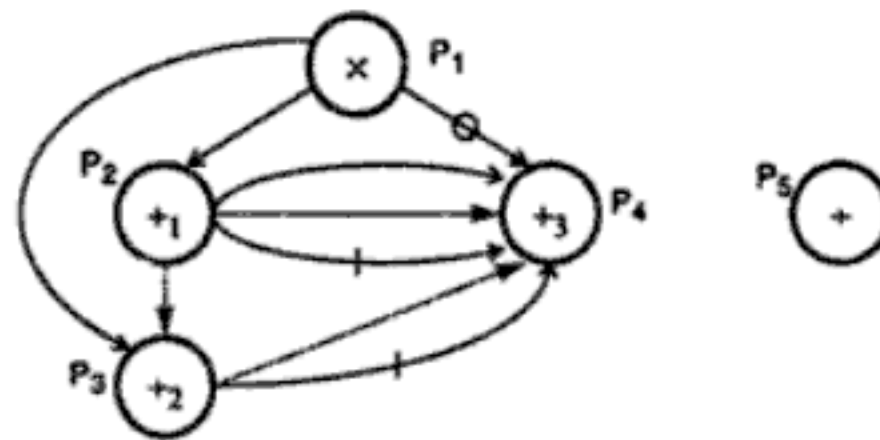
If two adders are available simultaneously, the parallel execution requires only three steps in Fig. 2.2c. Pairwise, there are 10 pairs of statements to check against Bernstein's conditions. Only 5 pairs, $P_1 \parallel P_5, P_2 \parallel P_3, P_2 \parallel P_5, P_5 \parallel P_3$, and $P_4 \parallel P_5$, can execute in parallel as revealed in Fig. 2.2a if there are no resource conflicts. Collectively, only $P_2 \parallel P_3 \parallel P_5$ is possible (Fig. 2.2c) because $P_2 \parallel P_3$, $P_3 \parallel P_5$, and $P_5 \parallel P_2$ are all possible.

■

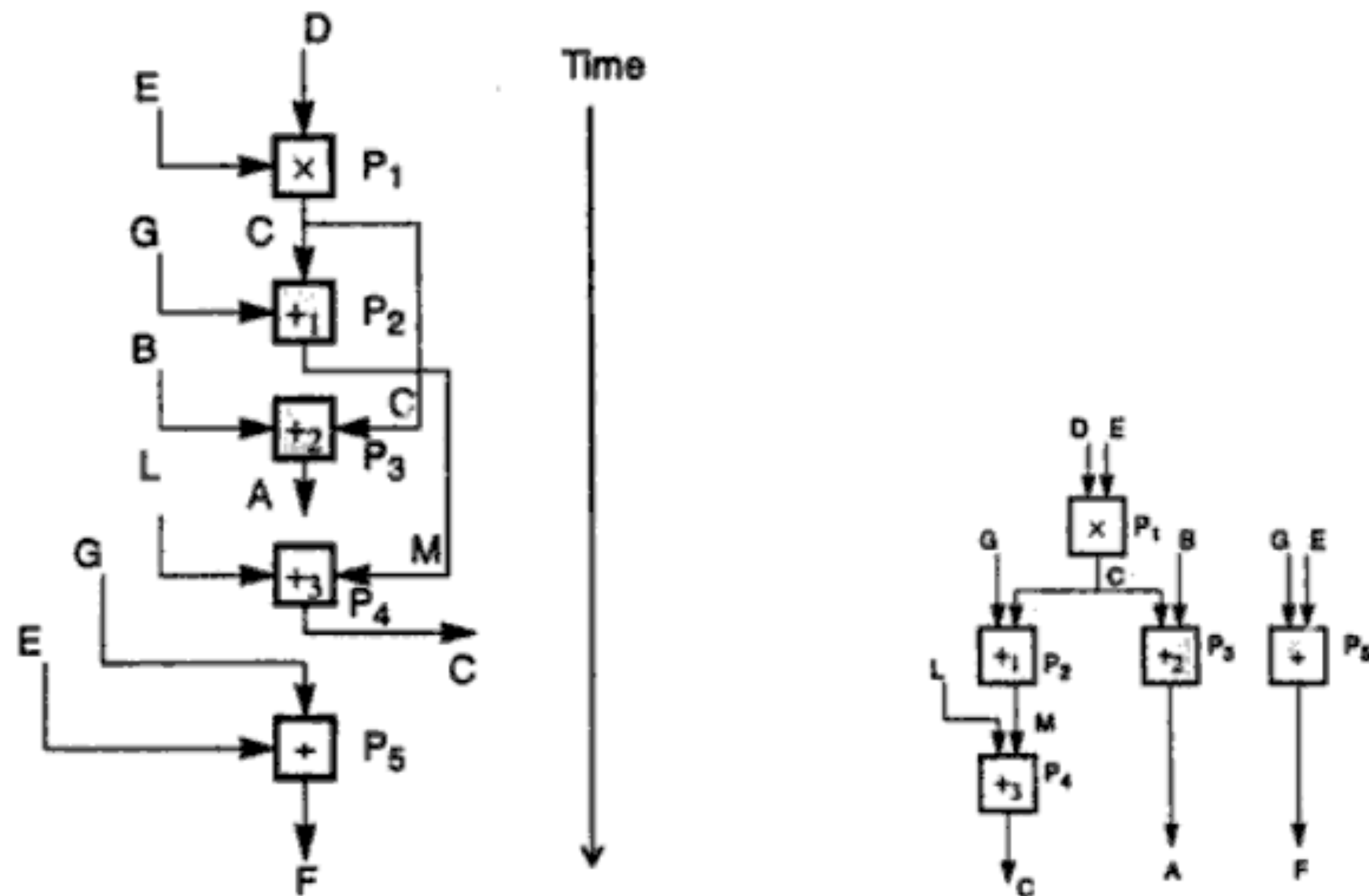
In general, the parallelism relation $\parallel$ is commutative; i.e., $P_i \parallel P_j$ implies $P_j \parallel P_i$. But the relation is not transitive; i.e., $P_i \parallel P_j$ and $P_j \parallel P_k$ do not necessarily guarantee $P_i \parallel P_k$. For example, we have $P_1 \parallel P_5$ and $P_5 \parallel P_2$, but $P_1 \nparallel P_2$, where $\nparallel$ means P_1 and P_2 cannot execute in parallel. In other words, the order in which P_1 and P_2 are executed will make a difference in the computational results.

Therefore, $\parallel$ is not an equivalence relation. However, $P_i \parallel P_j \parallel P_k$ implies associativity; i.e., $(P_i \parallel P_j) \parallel P_k = P_i \parallel (P_j \parallel P_k)$, since the order in which the parallel

Copyrighted material



(a) A dependence graph showing both data dependence (solid arrows) and resource dependence (dashed arrows)



(b) Sequential execution in five steps, assuming one step per statement (no pipelining)

(c) Parallel execution in three steps, assuming two adders are available per step

Figure 2.2 Detection of parallelism in the program of Example 2.2.

executable processes are executed should not make any difference in the output sets. It should be noted that the condition $I_i \cap I_j \neq \emptyset$ does not prevent parallelism between P_i and P_j.

Violations of any one or more of the three conditions in Eq. 2.1 prohibits parallelism between two processes. In general, violation of any one or more of the $3n(n-1)/2$ Bernstein's conditions among n processes prohibits parallelism collectively or partially. Many program constructs may prohibit parallelism.

Any statements or processes which depend on run-time conditions are not transformed to parallel form. These include IF statements or conditional branches. Recursive computations in successive iterations also prohibit parallelism. In general, data dependence, control dependence, and resource dependence all prevent parallelism from being exploitable.

The statement-level dependence can be generalized to higher levels, such as code

Copyrighted material

segment, subroutine, process, task, and program levels. The dependence of two higher-level objects can be inferred from the dependence of statements in the corresponding objects. The goals of analyzing the data dependence, control dependence, and resource dependence in a code are to identify opportunities for parallelization or vectorization.

Very often program restructuring or code transformations need to be performed before such opportunities can be revealed. The dependence relations are used in instruction issue and pipeline scheduling operations described in Chapter 6. We wish to build intelligent compilers to detect parallelism automatically. Further discussion of compiler techniques is given in Chapter 10.

2.1.2 Hardware and Software Parallelism

For implementation of parallelism, we need special hardware and software support. In this section, we address these support issues. We first distinguish between hardware and software parallelism. The mismatch problem between hardware and software is discussed. Then we describe the fundamental concept of compilation support needed to close the gap between hardware and software.

Details of special hardware functions and software support for parallelism will be treated in the remaining chapters. The key idea being conveyed is that parallelism cannot be achieved free. Besides theoretical conditioning, joint efforts between hardware designers and software programmers are needed to exploit parallelism in upgrading computer performance.

Hardware Parallelism This refers to the type of parallelism defined by the machine architecture and hardware multiplicity. Hardware parallelism is often a function of cost and performance tradeoffs. It displays the resource utilization patterns of simultaneously executable operations. It can also indicate the peak performance of the processor resources.

One way to characterize the parallelism in a processor is by the number of instruction issues per machine cycle. If a processor issues k instructions per machine cycle, then it is called a *k-issue* processor.

A conventional processor takes one or more machine cycles to issue a single instruction. These types of processors are called *one-issue* machines, with a single instruction pipeline in the processor. In a modern processor, two or more instructions can be issued per machine cycle.

For example, the Intel i960CA is a three-issue processor with one arithmetic, one memory access, and one branch instruction issued per cycle. The IBM RISC/System 6000 is a four-issue processor capable of issuing one arithmetic, one memory access, one floating-point, and one branch operation per cycle.

A multiprocessor system built with n k-issue processors should be able to handle a maximum number of nk threads of instructions simultaneously.

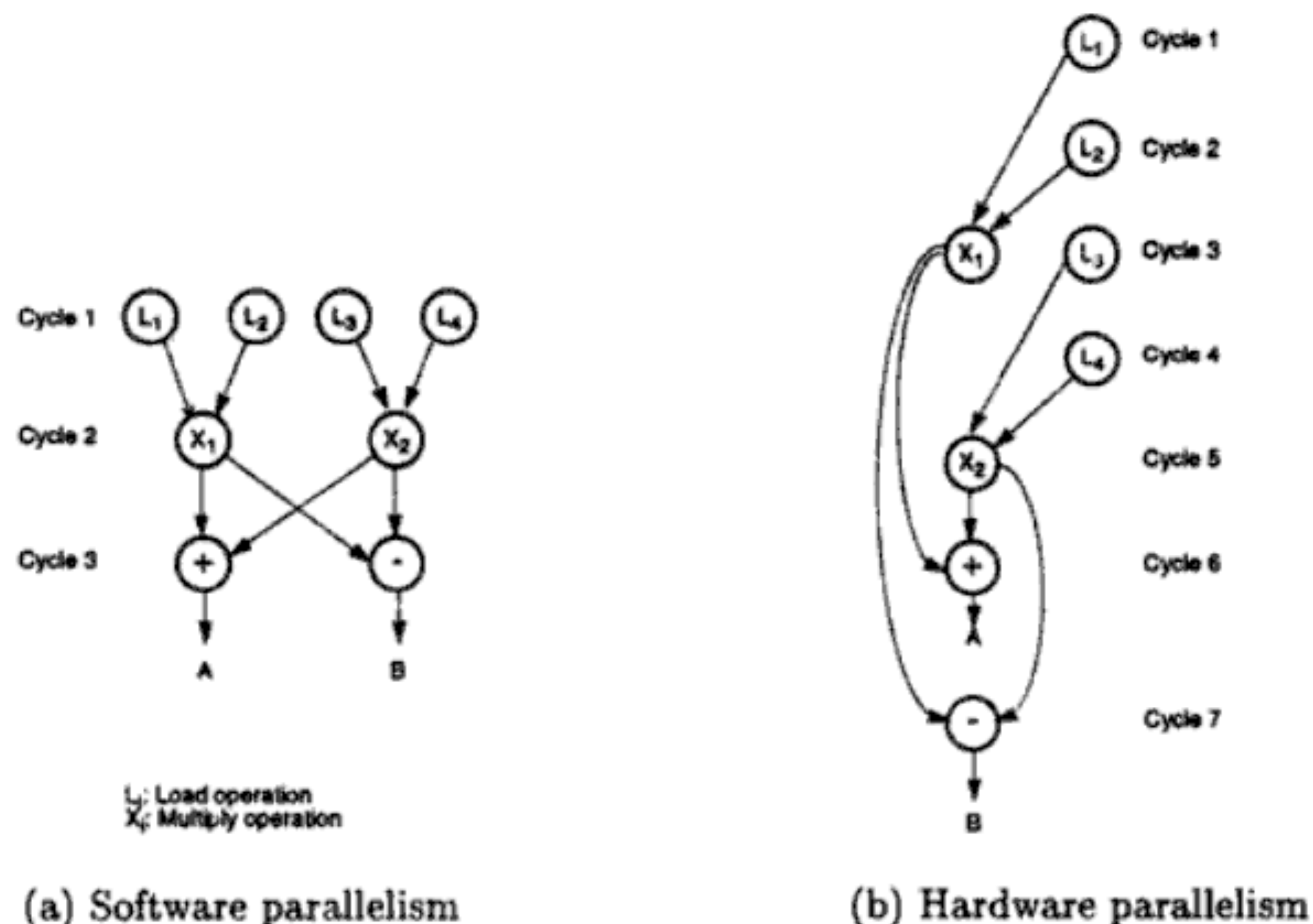
Software Parallelism This type of parallelism is defined by the control and data dependence of programs. The degree of parallelism is revealed in the program profile or in the program flow graph. Software parallelism is a function of algorithm, program-

Copyrighted material

ming style, and compiler optimization. The program flow graph displays the patterns of simultaneously executable operations. Parallelism in a program varies during the execution period. It often limits the sustained performance of the processor.

Example 2.3 Mismatch between software parallelism and hardware parallelism (Wen-Mei Hwu, 1991)

Consider the example program graph in Fig. 2.3a. There are eight instructions (four *loads* and four *arithmetic* operations) to be executed in three consecutive machine cycles. Four *load* operations are performed in the first cycle, followed by two *multiply* operations in the second cycle and two *add/subtract* operations in the third cycle. Therefore, the parallelism varies from 4 to 2 in three cycles. The average software parallelism is equal to $8/3 = 2.67$ instructions per cycle in this example program.



(a) Software parallelism (b) Hardware parallelism

Figure 2.3 Executing an example program by a two-issue superscalar processor.

Now consider execution of the same program by a two-issue processor which can execute one memory access (*load* or *write*) and one arithmetic (*add*, *subtract*, *multiply*, etc.) operation simultaneously. With this hardware restriction, the program must execute in seven machine cycles as shown in Fig. 2.3b. Therefore, the *hardware parallelism* displays an average value of $8/7 = 1.14$ instructions executed per cycle. This demonstrates a mismatch between the software parallelism and the hardware parallelism.

Let us try to match the software parallelism shown in Fig. 2.3a in a hardware platform of a dual-processor system, where single-issue processors are used.

Copyrighted material

The achievable hardware parallelism is shown in Fig. 2.4, where L/S stands for *load/store* operations. Note that six processor cycles are needed to execute the 12 instructions by two processors. S_1 and S_2 are two inserted *store* operations, and l_5 and l_6 are two inserted *load* operations. These added instructions are needed for interprocessor communication through the shared memory.

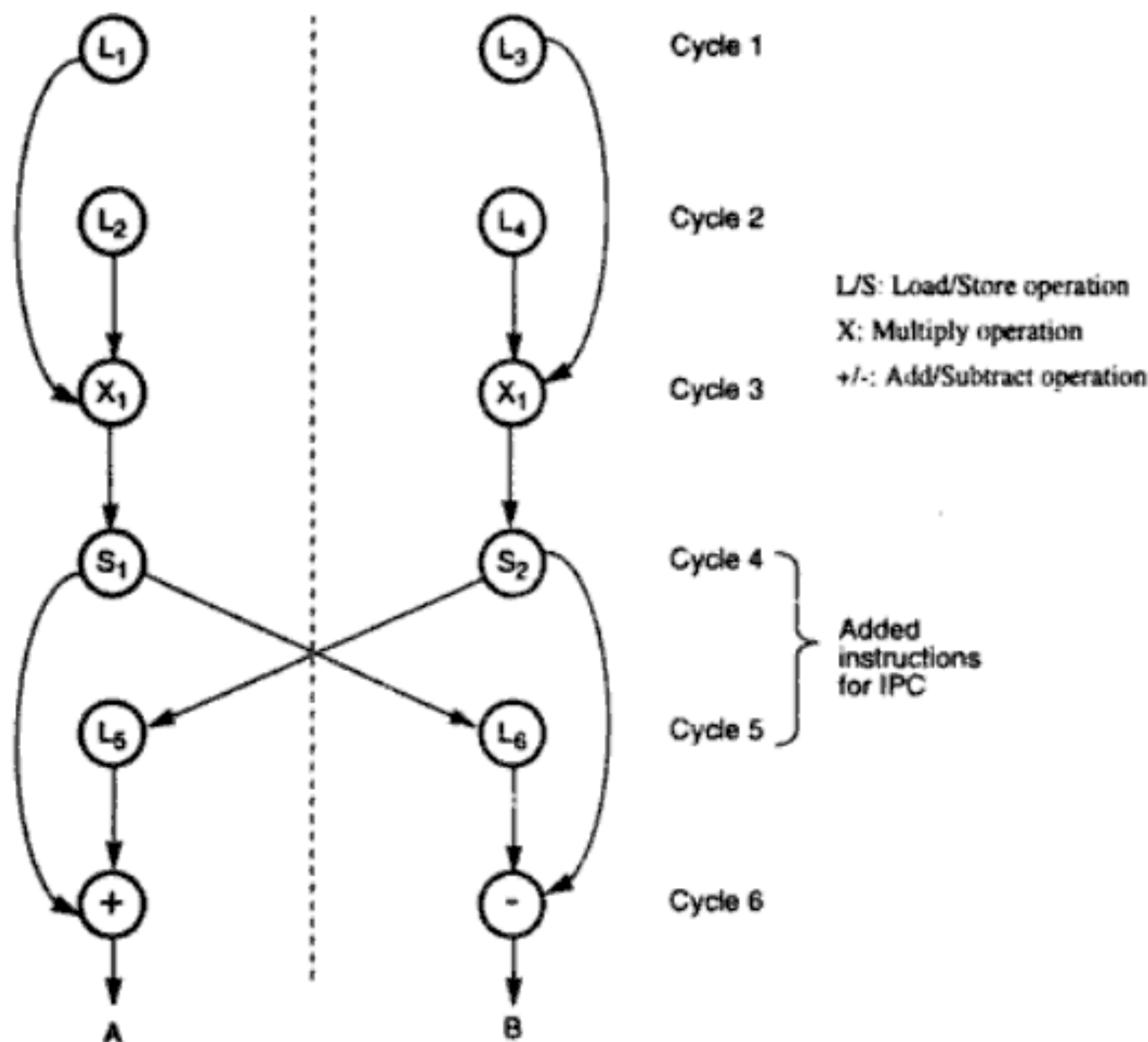


Figure 2.4 Dual-processor execution of the program in Fig. 2.3a.

■

Of the many types of software parallelism, two are most frequently cited as important to parallel programming: The first is *control parallelism*, which allows two or more operations to be performed simultaneously. The second type has been called *data parallelism*, in which almost the same operation is performed over many data elements by many processors simultaneously.

Control parallelism, appearing in the form of pipelining or multiple functional units, is limited by the pipeline length and by the multiplicity of functional units. Both pipelining and functional parallelism are handled by the hardware; programmers need take no special actions to invoke them.

Data parallelism offers the highest potential for concurrency. It is practiced in both SIMD and MIMD modes on MPP systems. Data parallel code is easier to write and to debug than control parallel code. Synchronization in SIMD data parallelism is handled by the hardware. Data parallelism exploits parallelism in proportion to the quantity

Copyrighted material

of data involved. Thus data parallel computations appeal to scaled problems, in which the performance of a MPP does not drop sharply with small sequential fraction in the program.

To solve the mismatch problem between software parallelism and hardware parallelism, one approach is to develop compilation support, and the other is through hardware redesign for more efficient exploitation by an intelligent compiler. These two approaches must cooperate with each other to produce the best result.

Hardware processors can be better designed to exploit parallelism by an optimizing compiler. Pioneer work in processor technology with this objective can be found in the IBM 801, Stanford MIPS, and Berkeley RISC. Most processors use a large register file and sustained instruction pipelining to execute nearly one instruction per cycle. The large register file supports fast access to temporary values generated by an optimizing compiler. The registers are exploited by the code optimizer and global register allocator in such a compiler.

The instruction scheduler exploits the pipeline hardware by filling *branch* and *load* delay slots. In superscalar and superpipelining, hardware and software branch prediction, multiple instruction issue, speculative execution, high bandwidth instruction cache, and support for dynamic scheduling are needed to facilitate the detection of parallelism opportunities. The architecture must be designed interactively with the compiler.

2.1.3 The Role of Compilers

Compiler techniques are used to exploit hardware features to improve performance. The pioneer work on the IBM PL.8 and Stanford MIPS compilers has aimed for this goal. Other optimizing compilers for exploiting parallelism include the CDC STACKLIB, Cray CFT, Illinois Parafrase, Rice PFC, Yale Bulldog, and Illinois IMPACT.

In Chapter 10, we will study loop transformation, software pipelining, and features developed in existing optimizing compilers for supporting parallelism. Interaction between compiler and architecture design is a necessity in modern computer development. Most existing processors issue one instruction per cycle and provide a few registers. This may cause excessive spilling of temporary results from the available registers. Therefore, more software parallelism may not improve performance in conventional scalar processors.

There exists a vicious cycle of limited hardware support and the use of a naive compiler. To break the cycle, one must design the compiler and the hardware jointly at the same time. Interaction between the two can lead to a better solution to the mismatch problem between software and hardware parallelism.

The general guideline is to increase the flexibility in hardware parallelism and to exploit software parallelism in control-intensive programs. Hardware and software design tradeoffs also exist in terms of cost, complexity, expandability, compatibility, and performance. Compiling for multiprocessors is much more involved than for uniprocessors. Both granularity and communication latency play important roles in the code optimization and scheduling process.

Copyrighted material

2.2 Program Partitioning and Scheduling

This section introduces the basic definitions of computational granularity or level of parallelism in programs. Communication latency and scheduling issues are illustrated with programming examples.

2.2.1 Grain Sizes and Latency

Grain size or *granularity* is a measure of the amount of computation involved in a software process. The simplest measure is to count the number of instructions in a grain (program segment). Grain size determines the basic program segment chosen for parallel processing. Grain sizes are commonly described as *fine*, *medium*, or *coarse*, depending on the processing levels involved.

Latency is a time measure of the communication overhead incurred between machine subsystems. For example, the *memory latency* is the time required by a processor to access the memory. The time required for two processes to synchronize with each other is called the *synchronization latency*. Computational granularity and communication latency are closely related. We reveal their relationship below.

Parallelism has been exploited at various processing levels. As illustrated in Fig. 2.5, five levels of program execution represent different computational grain sizes and changing communication and control requirements. The lower the level, the finer the granularity of the software processes.

In general, the execution of a program may involve a combination of these levels. The actual combination depends on the application, formulation, algorithm, language, program, compilation support, and hardware limitations. We characterize below the parallelism levels and review their implementation issues from the viewpoints of a programmer and of a compiler writer.

Instruction Level At instruction or statement level, a typical grain contains less than 20 instructions, called *fine grain* in Fig. 2.5. Depending on individual programs, fine-grain parallelism at this level may range from two to thousands. Butler et al. (1991) has shown that single-instruction-stream parallelism is greater than two. Wall (1991) finds that the average parallelism at instruction level is around five, rarely exceeding seven, in an ordinary program. For scientific applications, Kumar (1988) has measured the average parallelism in the range of 500 to 3000 Fortran statements executing concurrently in an idealized environment.

The advantage of fine-grain computation lies in the abundance of parallelism. The exploitation of fine-grain parallelism can be assisted by an optimizing compiler which should be able to automatically detect parallelism and translate the source code to a parallel form which can be recognized by the run-time system. Instruction-level parallelism is rather tedious for an ordinary programmer to detect in a source code.

Loop Level This corresponds to the iterative loop operations. A typical loop contains less than 500 instructions. Some loop operations, if independent in successive iterations, can be vectorized for pipelined execution or for lock-step execution on SIMD machines.

Copyrighted material

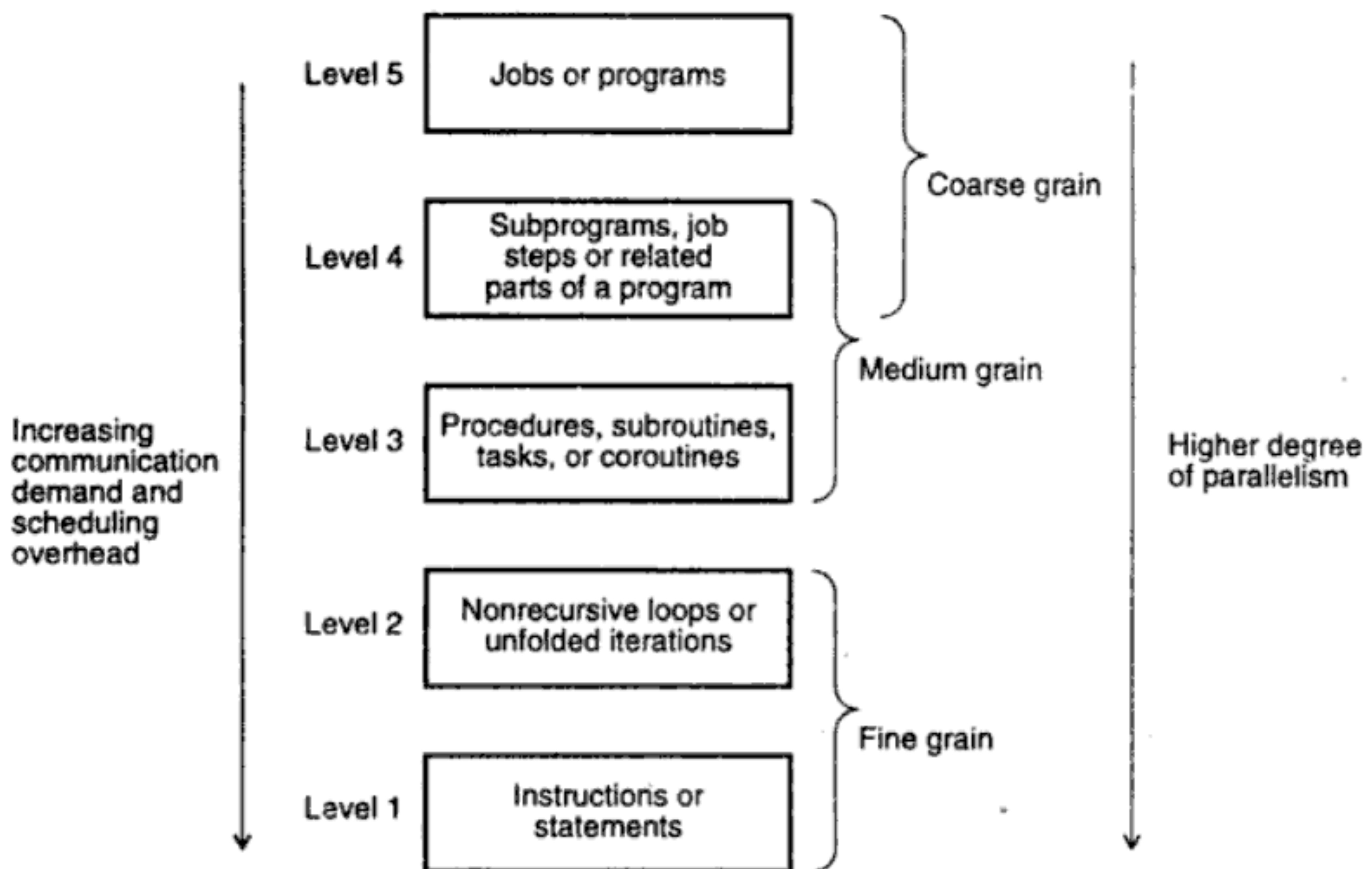


Figure 2.5 Levels of parallelism in program execution on modern computers. (Reprinted from Hwang, *Proc. IEEE*, October 1987)

Some loop operations can be self-scheduled for parallel execution on MIMD machines.

Loop-level parallelism is the most optimized program construct to execute on a parallel or vector computer. However, recursive loops are rather difficult to parallelize. Vector processing is mostly exploited at the loop level (level 2 in Fig. 2.5) by a vectorizing compiler. The loop level is still considered a fine grain of computation.

Procedure Level This level corresponds to medium-grain size at the task, procedural, subroutine, and coroutine levels. A typical grain at this level contains less than 2000 instructions. Detection of parallelism at this level is much more difficult than at the finer-grain levels. Interprocedural dependence analysis is much more involved and history-sensitive.

The communication requirement is often less compared with that required in MIMD execution mode. SPMD execution mode is a special case at this level. Multitasking also belongs in this category. Significant efforts by programmers may be needed to restructure a program at this level, and some compiler assistance is also needed.

Subprogram Level This corresponds to the level of job steps and related subprograms. The grain size may typically contain thousands of instructions. Job steps can overlap across different jobs. Subprograms can be scheduled for different processors in

Copyrighted material

SPMD or MPMD mode, often on message-passing multicomputers.

Multiprogramming on a uniprocessor or on a multiprocessor is conducted at this level. In the past, parallelism at this level has been exploited by algorithm designers or programmers, rather than by compilers. We do not have good compilers for exploiting medium- or coarse-grain parallelism at present.

Job (Program) Level This corresponds to the parallel execution of essentially independent jobs (programs) on a parallel computer. The grain size can be as high as tens of thousands of instructions in a single program. For supercomputers with a small number of very powerful processors, such coarse-grain parallelism is practical. Job-level parallelism is handled by the program loader and by the operating system in general. Time-sharing or space-sharing multiprocessors explore this level of parallelism. In fact, both time and space sharing are extensions of multiprogramming.

To summarize, fine-grain parallelism is often exploited at instruction or loop levels, preferably assisted by a parallelizing or vectorizing compiler. Medium-grain parallelism at the task or job step demands significant roles for the programmer as well as compilers. Coarse-grain parallelism at the program level relies heavily on an effective OS and on the efficiency of the algorithm used. Shared-variable communication is often used to support fine-grain and medium-grain computations.

Message-passing multicomputers have been used for medium- and coarse-grain computations. In general, the finer the grain size, the higher the potential for parallelism and the higher the communication and scheduling overhead. Fine grain provides a higher degree of parallelism, but heavier communication overhead, as compared with coarse-grain computations. Massive parallelism is often explored at the fine-grain level, such as data parallelism on SIMD or MIMD computers.

Communication Latency By balancing granularity and latency, one can achieve better performance of a computer system. Various latencies are attributed to machine architecture, implementing technology, and communication patterns involved. The architecture and technology affect the design choices for latency tolerance between subsystems. In fact, latency imposes a limiting factor on the scalability of the machine size. For example, memory latency increases with respect to memory capacity. Thus memory cannot be increased indefinitely without exceeding the tolerance level of the access latency. Various latency hiding or tolerating techniques will be studied in Chapter 9.

The latency incurred with interprocessor communication is another important parameter for a system designer to minimize. Besides signal delays in the data path, IPC latency is also affected by the communication patterns involved. In general, n tasks communicating with each other may require $n(n-1)/2$ communication links among them. Thus the complexity grows quadratically. This leads to a communication bound which limits the number of processors allowed in a large computer system.

Communication patterns are determined by the algorithms used as well as by the architectural support provided. Frequently encountered patterns include *permutations* and *broadcast*, *multicast*, and *conference* (many-to-many) communications. The communication demand may limit the granularity or parallelism. Very often tradeoffs do exist between the two.

Copyrighted material

The communication issue thus involves the reduction of latency or complexity, the prevention of deadlock, minimizing blocking in communication patterns, and the tradeoff between parallelism and communication overhead. We will study techniques that minimize communication latency, prevent deadlock, and optimize grain size throughout the book.

2.2.2 Grain Packing and Scheduling

Two fundamental questions to ask in parallel programming are: (i) How can we partition a program into parallel branches, program modules, microtasks, or grains to yield the shortest possible execution time? and (ii) What is the optimal size of concurrent grains in a computation?

This grain-size problem demands determination of both the number and the size of grains (or microtasks) in a parallel program. Of course, the solution is both problem-dependent and machine-dependent. The goal is to produce a short schedule for fast execution of subdivided program modules.

There exists a tradeoff between parallelism and scheduling/synchronization overhead. The time complexity involves both computation and communication overheads The program partitioning involves the algorithm designer, programmer, compiler, operating system support, etc. We describe below a *grain packing* approach introduced by Kruatrachue and Lewis (1988) for parallel programming applications.

Example 2.4 Program graph before and after grain packing (Kruatrachue and Lewis, 1988)

The basic concept of program partitioning is introduced below. In Fig. 2.6, we show an example *program graph* in two different grain sizes. A program graph shows the structure of a program. It is very similar to the dependence graph introduced in Section 2.1.1. Each node in the program graph corresponds to a computational unit in the program. The *grain size* is measured by the number of basic machine cycles (including both processor and memory cycles) needed to execute all the operations within the node.

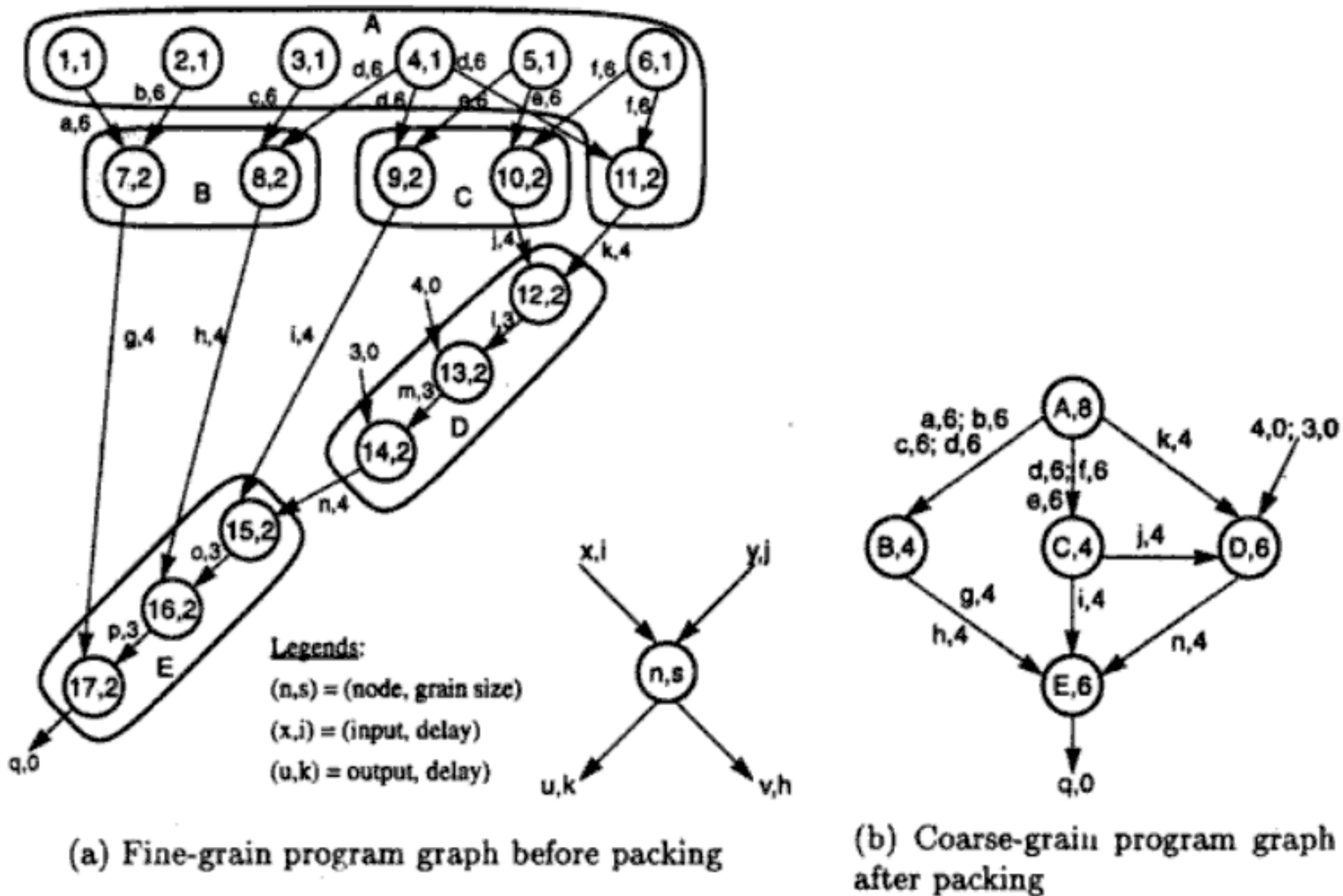
We denote each node in Fig. 2.6 by a pair (n, s), where n is the *node name* (id) and s is the grain size of the node. Thus grain size reflects the number of computations involved in a program segment. Fine-grain nodes have a smaller grain size, and coarse-grain nodes have a larger grain size.

The edge label (v, d) between two end nodes specifies the output variable v from the source node or the input variable to the destination node, and the communication delay d between them. This delay includes all the path delays and memory latency involved.

There are 17 nodes in the fine-grain program graph (Fig. 2.6a) and 5 in the coarse-grain program graph (Fig. 2.6b). The coarse-grain node is obtained by combining (grouping) multiple fine-grain nodes. The fine grain corresponds to the following program:

Var $a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q$

Copyrighted material



(a) Fine-grain program graph before packing

(b) Coarse-grain program graph after packing

Figure 2.6 A program graph before and after grain packing in Example 2.4. (Modified from Kruatrachue and Lewis, *IEEE Software*, Jan. 1988)

Begin

1.	$a := 1$	10.	$j := e \times f$
2.	$b := 2$	11.	$k := d \times f$
3.	$c := 3$	12.	$l := j \times k$
4.	$d := 4$	13.	$m := 4 \times l$
5.	$e := 5$	14.	$n := 3 \times m$
6.	$f := 6$	15.	$o := n \times i$
7.	$g := a \times b$	16.	$p := o \times h$
8.	$h := c \times d$	17.	$q := p \times q$
9.	$i := d \times e$		

End

Nodes 1, 2, 3, 4, 5, and 6 are memory reference (data fetch) operations. Each takes one cycle to address and six cycles to fetch from memory. All remaining nodes (7 to 17) are CPU operations, each requiring two cycles to complete. After packing, the coarse-grain nodes have larger grain sizes ranging from 4 to 8 as shown.

The node (A,8) in Fig. 2.6b is obtained by combining the nodes (1,1), (2,1), (3,1), (4,1), (5,1), (6,1), and (11,2) in Fig. 2.6a. The grain size, 8, of node A is the summation of all grain sizes $(1 + 1 + 1 + 1 + 1 + 1 + 2 = 8)$ being combined. ■

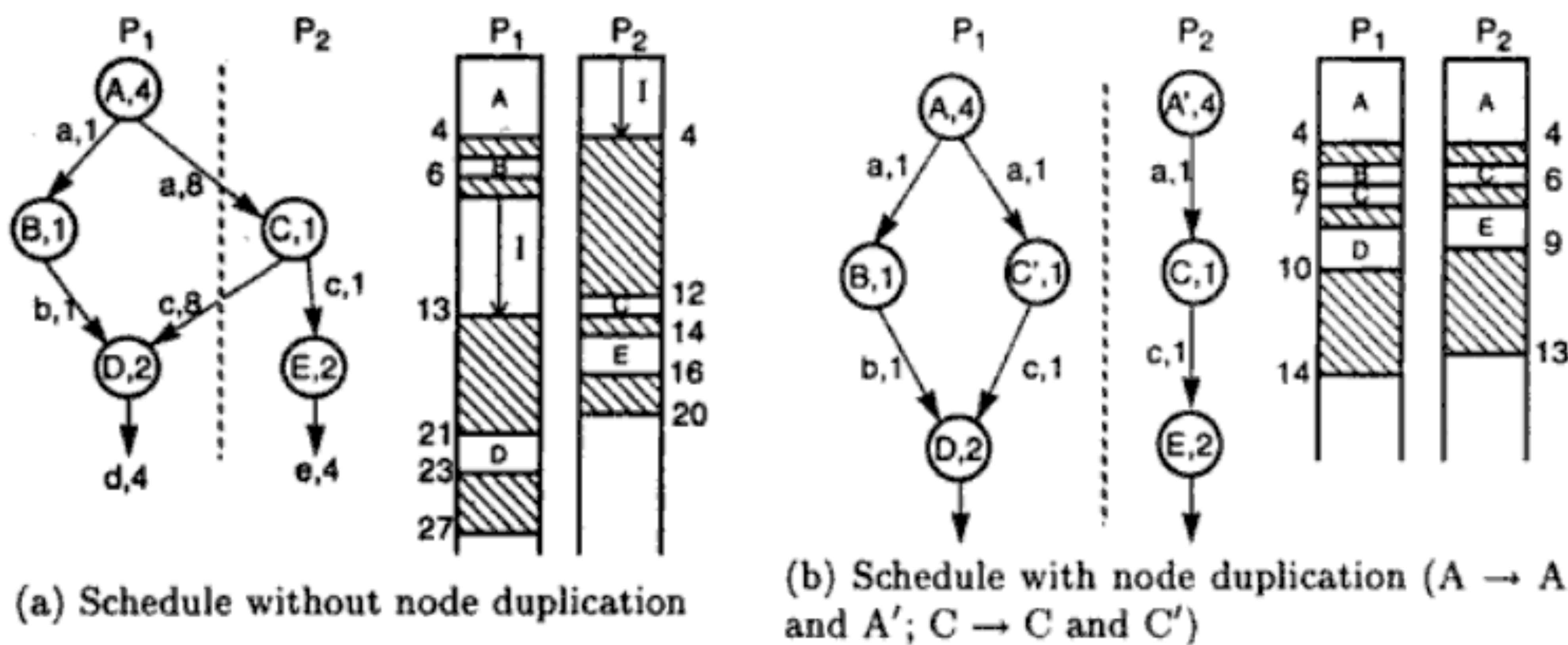
Copyrighted material

2.2.3 Static Multiprocessor Scheduling

Grain packing may not always produce a shorter schedule. In general, dynamic multiprocessor scheduling is an NP-hard problem. Very often heuristics are used to yield suboptimal solutions. We introduce below the basic concepts behind multiprocessor scheduling using static schemes.

Node Duplication In order to eliminate the idle time and to further reduce the communication delays among processors, one can duplicate some of the nodes in more than one processor.

Figure 2.8a shows a schedule without duplicating any of the five nodes. This schedule contains idle time as well as long interprocessor delays (8 units) between P1 and P2. In Fig. 2.8b, node A is duplicated into A′ and assigned to P2 besides retaining the original copy A in P1. Similarly, a duplicated node C′ is copied into P1 besides the original node C in P2. The new schedule shown in Fig. 2.8b is almost 50% shorter than that in Fig. 2.8a. The reduction in schedule time is caused by elimination of the (a, 8) and (c, 8) delays between the two processors.



(a) Schedule without node duplication

(b) Schedule with node duplication (A → A and A′; C → C and C′)

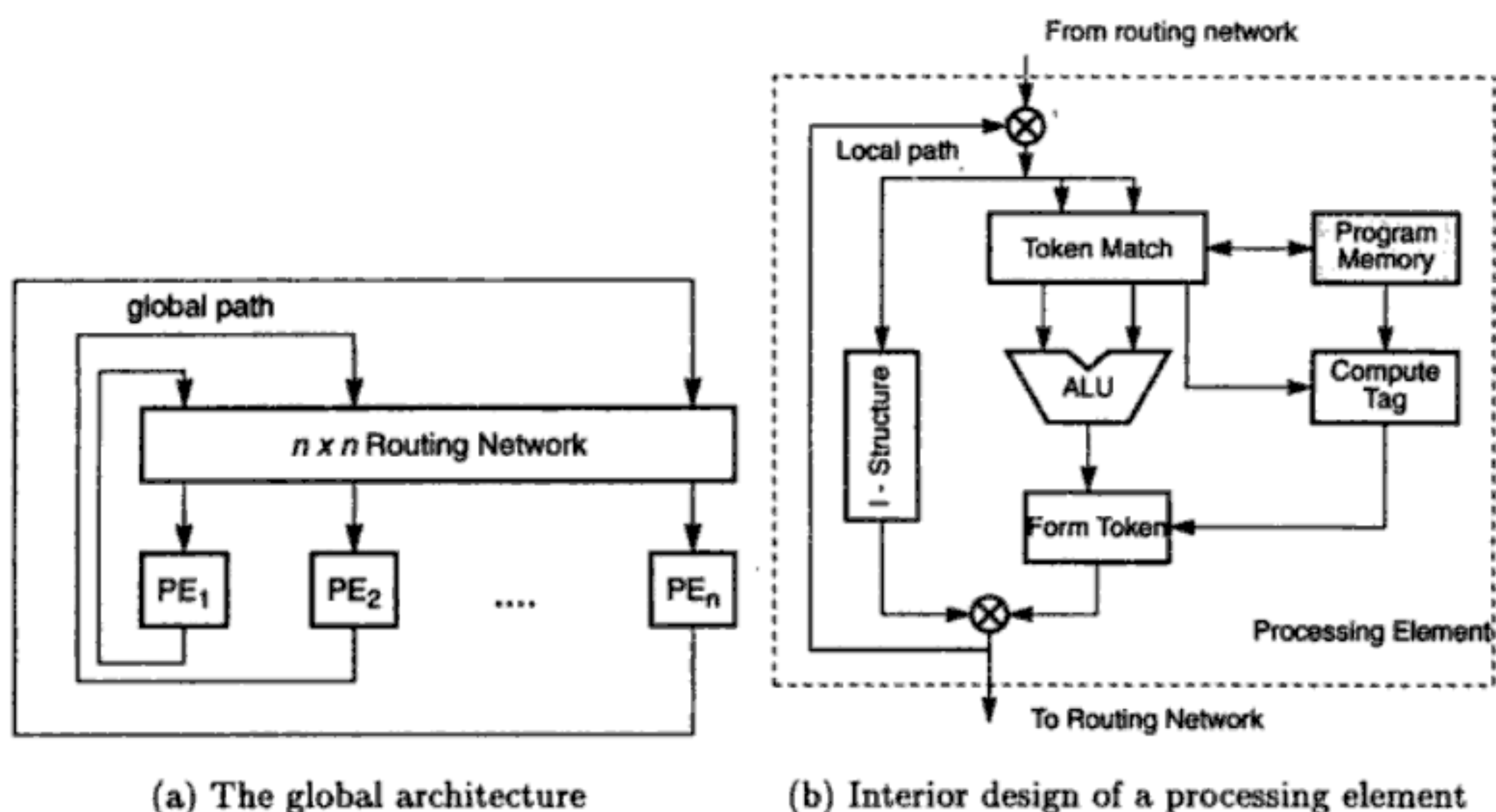
Figure 2.8 Node-duplication scheduling to eliminate communication delays between processors. (I: idle time; shaded areas: communication delays)

Grain packing and node duplication are often used jointly to determine the best grain size and corresponding schedule. Four major steps are involved in the grain determination and the process of scheduling optimization:

Step 1. Construct a fine-grain program graph.
Step 2. Schedule the fine-grain computation.
Step 3. Grain packing to produce the coarse grains.
Step 4. Generate a parallel schedule based on the packed graph.

The purpose of multiprocessor scheduling is to obtain a minimal time schedule for the computations involved. The following example clarifies this concept.

Copyrighted material



(a) The global architecture (b) Interior design of a processing element

Figure 2.12 The MIT tagged-token dataflow computer. (Adapted from Arvind and Iannucci, 1986 with permission)

other PEs through the routing network. All internal token circulation operations are pipelined without blocking.

One can think of the instruction address in a dataflow computer as replacing the program counter, and the context identifier replacing the frame base register in a control flow computer. It is the machine's job to match up data with the same tag to needy instructions. In so doing, new data will be produced with a new tag indicating the successor instruction(s). Thus, each instruction represents a synchronization operation. New tokens are formed and circulated along the PE pipeline for reuse or to other PEs through the global path, which is also pipelined.

Another synchronization mechanism, called the *I-structure*, is provided within each PE. The I-structure is a tagged memory unit for overlapped usage of a data structure by both the producer and consumer processes. Each word of I-structure uses a 2-bit tag indicating whether the word is *empty*, is *full*, or has *pending read* requests. The use of I-structure is a retreat from the pure dataflow approach. The purpose is to reduce excessive copying of large data structures in dataflow operations.

Example 2.6 Comparison of dataflow and control-flow computers (Gajski, Padua, Kuck, and Kuhn, 1982)

The dataflow graph in Fig. 2.13a shows that 24 instructions are to be executed (8 *divides*, 8 *multiplies*, and 8 *adds*). A dataflow graph is similar to a dependence graph or program graph. The only difference is that data tokens are passed around the edges in a dataflow graph. Assume that each *add*, *multiply*, and *divide* requires 1, 2,

Copyrighted material

PE data routing in SIMD computers.

Before we analyze various network topologies, let us define several parameters often used to estimate the complexity, communication efficiency, and cost of a network. In general, a network is represented by the graph of a finite number of nodes linked by directed or undirected edges. The number of nodes in the graph is called the *network size.*

Node Degree and Network Diameter The number of edges (links or channels) incident on a node is called the *node degree d.* In the case of unidirectional channels, the number of channels into a node is the *in degree*, and that out of a node is the *out degree.* Then the node degree is the sum of the two. The node degree reflects the number of I/O ports required per node, and thus the cost of a node. Therefore, the node degree should be kept a constant, as small as possible in order to reduce cost. A constant node degree is very much desired to achieve modularity in building blocks for scalable systems.

The *diameter D* of a network is the maximum shortest path between any two nodes. The path length is measured by the number of links traversed. The network diameter indicates the maximum number of distinct hops between any two nodes, thus providing a figure of communication merit for the network. Therefore, the network diameter should be as small as possible from a communication point of view.

Bisection Width When a given network is cut into two equal halves, the minimum number of edges (channels) along the cut is called the *channel bisection width b.* In the case of a communication network, each edge corresponds to a *channel* with w bit wires. Then the *wire bisection width* is $B = bw$. This parameter B reflects the wiring density of a network. When B is fixed, the *channel width* (in bits) $w = B/b$. Thus the bisection width provides a good indicator of the maximum communication bandwidth along the bisection of a network. All other cross sections should be bounded by the bisection width.

Another quantitative parameter is the *wire length* (or channel length) between nodes. This may affect the signal latency, clock skewing, or power requirements. We label a network *symmetric* if the topology is the same looking from any node. Symmetric networks are easier to implement or to program. Whether the nodes are homogeneous, the channels are buffered, or some of the nodes are switches are other useful properties for characterizing the structure of a network.

Data-Routing Functions A data-routing network is used for inter-PE data exchange. This routing network can be static, such as the hypercube routing network used in the TMC/CM-2, or dynamic such as the multistage network used in the IBM GF11. In the case of a multicomputer network, the data routing is achieved through message passing. Hardware routers are used to route messages among multiple computer nodes.

We specify below some primitive data-routing functions implementable on an inter-PE routing network. The versatility of a routing network will reduce the time needed for data exchange and thus can significantly improve the system performance.

Commonly seen data-routing functions among the PEs include *shifting*, *rotation*,

Copyrighted material

bidirectional. It is symmetric with a constant node degree of 2. The diameter is $\lfloor N/2 \rfloor$ for a bidirectional ring, and N for unidirectional ring.

The IBM *token ring* has this topology, in which messages circulate along the ring until they reach the destination with a matching token. Pipelined or packet-switched, rings have been implemented in the CDC Cyberplus multiprocessor (1985) and in the KSR-1 computer system (1992) for interprocessor communications.

By increasing the node degree from 2 to 3 or 4, we obtain two *chordal rings* as shown in Figs. 2.16c and 2.16d, respectively. One and two extra links are added to produce the two chordal rings, respectively. In general, the more links added, the higher the node degree and the shorter the network diameter.

Comparing the 16-node ring (Fig. 2.16b) with the two chordal rings (Figs. 2.16c and 2.16d), the network diameter drops from 8 to 5 and to 3, respectively. In the extreme, the *completely connected network* in Fig. 2.16f has a node degree of 15 with the shortest possible diameter of 1.

Barrel Shifter As shown in Fig. 2.16e for a network of $N = 16$ nodes, the *barrel shifter* is obtained from the ring by adding extra links from each node to those nodes having a distance equal to an integer power of 2. This implies that node i is connected to node j if $|j - i| = 2^r$ for some $r = 0, 1, 2, \cdots, n-1$ and the network size is $N = 2^n$. Such a barrel shifter has a node degree of $d = 2n - 1$ and a diameter $D = n/2$.

Obviously, the connectivity in the barrel shifter is increased over that of any chordal ring of lower node degree. For $N = 16$, the barrel shifter has a node degree of 7 with a diameter of 2. But the barrel shifter complexity is still much lower than that of the completely connected network (Fig. 2.16f).

Tree and Star A *binary tree* of 31 nodes in five levels is shown in Fig. 2.17a. In general, a k-level, completely balanced binary tree should have $N = 2^k - 1$ nodes. The maximum node degree is 3 and the diameter is $2(k-1)$. With a constant node degree, the binary tree is a scalable architecture. However, the diameter is rather long.

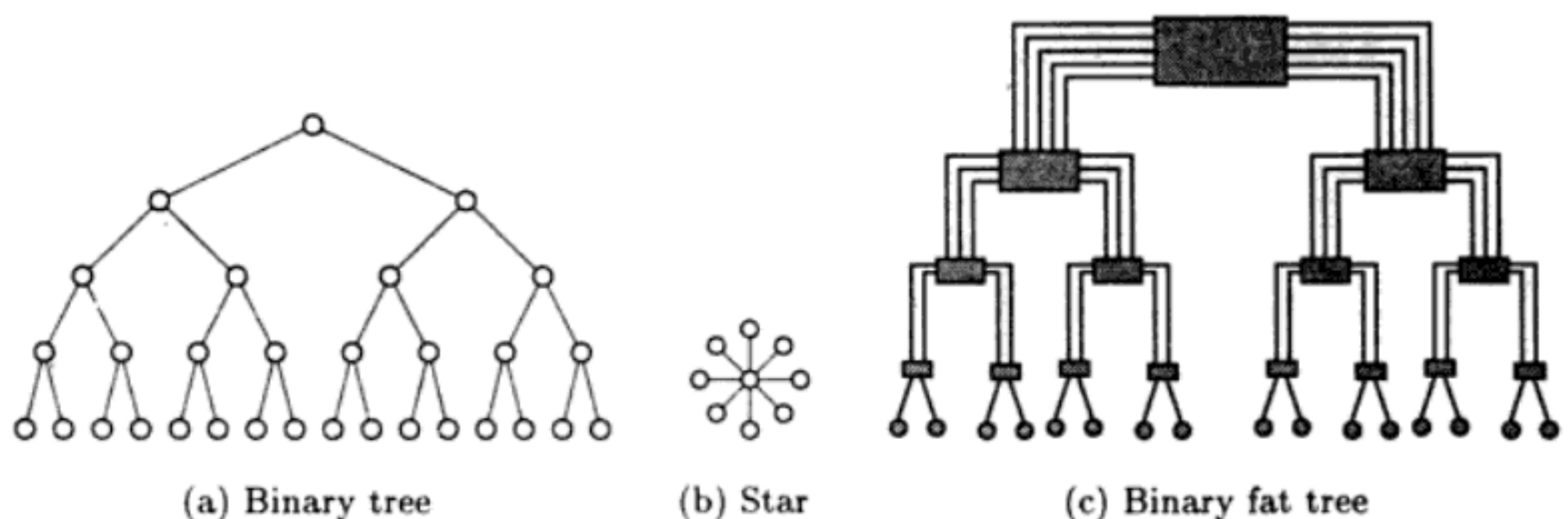
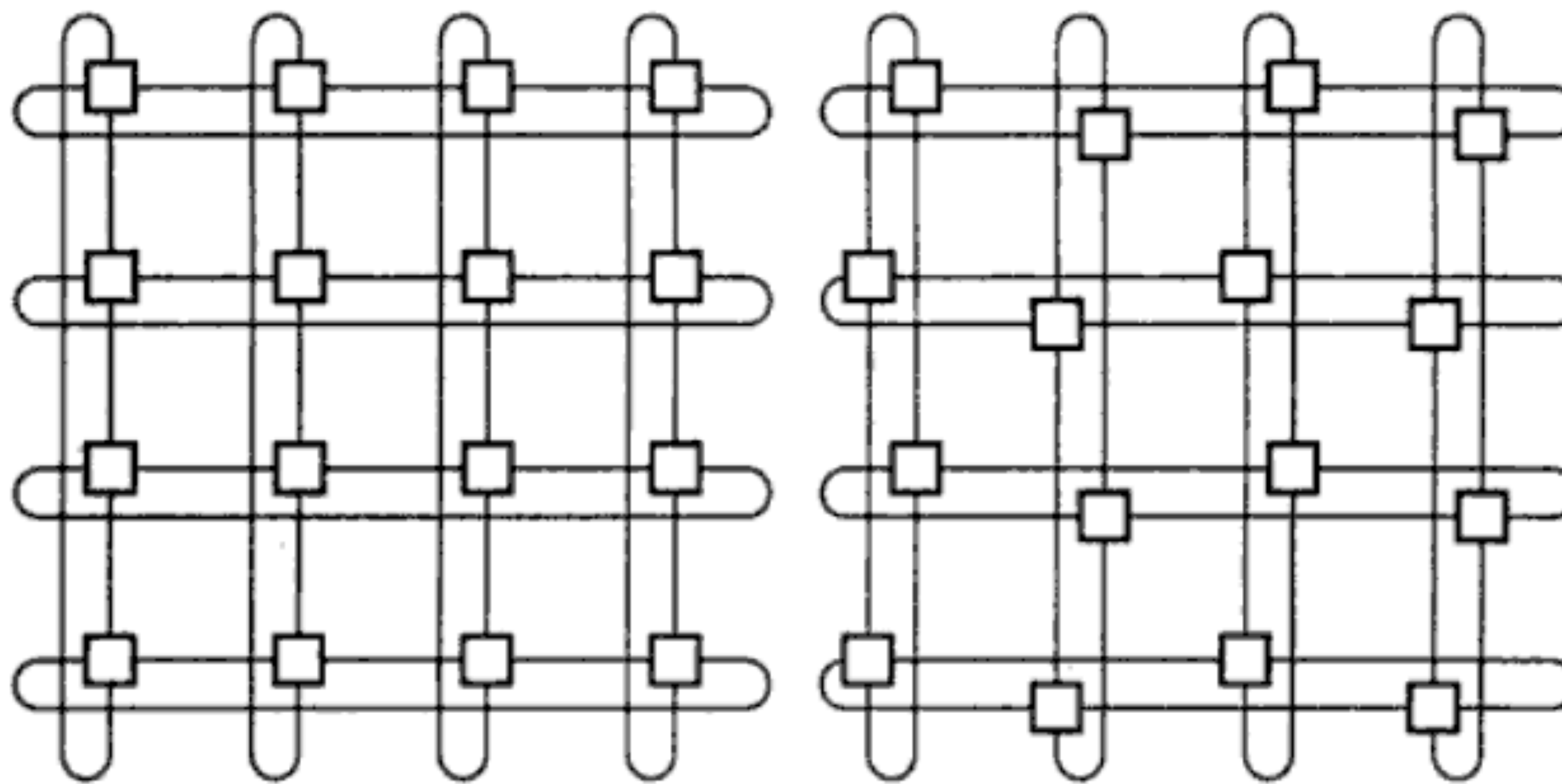


Figure 2.17 Tree, star, and fat tree.

Copyrighted material

is embedded in a plane.



(a) Traditional torus (a 4-ary 2-cube) (b) A torus with folded connections

Figure 2.21 Folded connections to equalize the wire length in a torus network. (Courtesy of W. Dally; reprinted with permission from *IEEE Trans. Computers*, June 1990)

William Dally (1990) has revealed a number of interesting properties of k-ary n-cube networks. The cost of such a network is dominated by the amount of wire, rather by the number of switches required. Under the assumption of constant wire bisection, low-dimensional networks with wide channels provide lower latency, less contention, and higher hot-spot throughput than higher-dimensional networks with narrow channels.

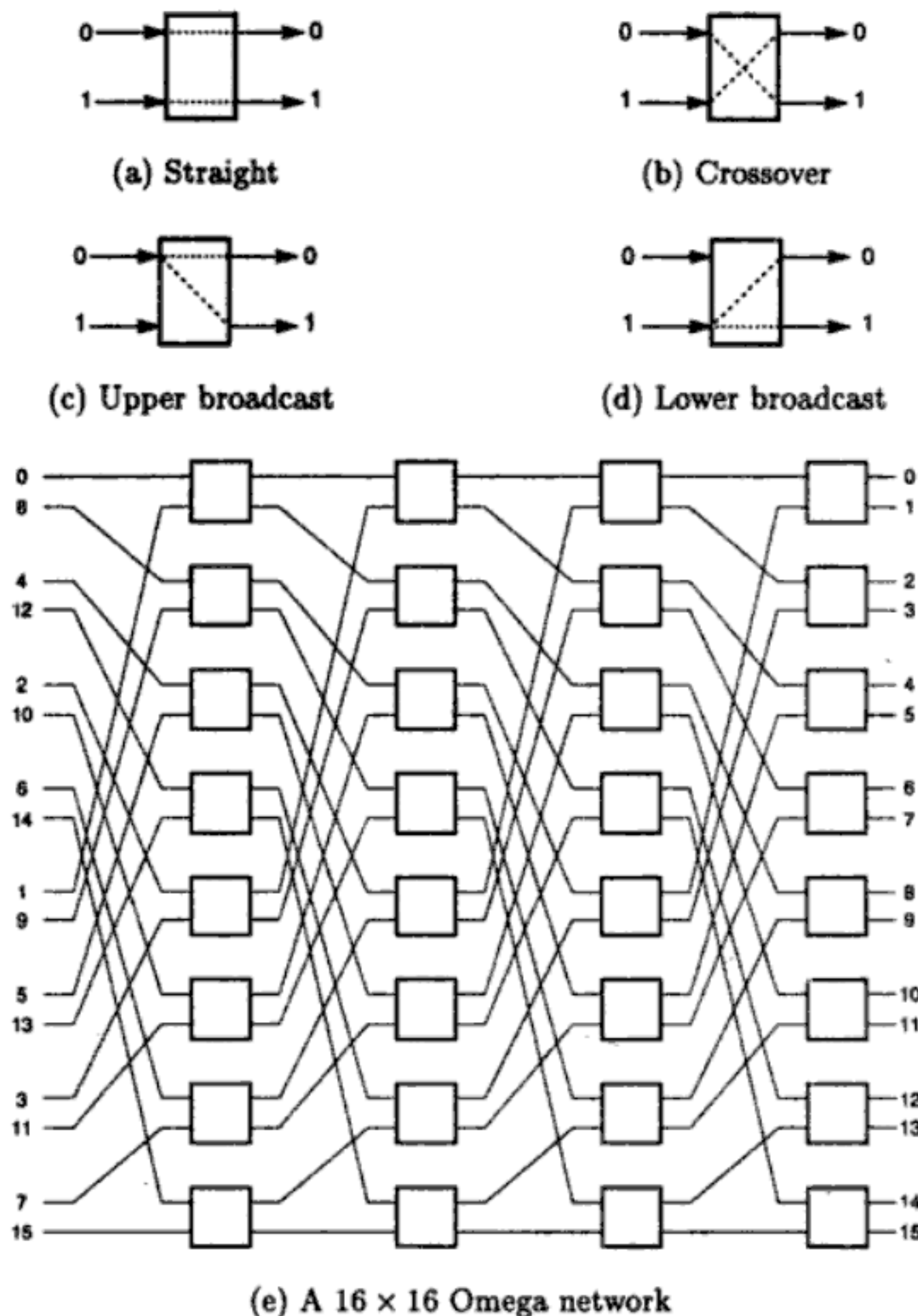
Network Throughput The *network throughput* is defined as the total number of messages the network can handle per unit time. One method of estimating throughput is to calculate the capacity of a network, the total number of messages that can be in the network at once. Typically, the maximum throughput of a network is some fraction of its capacity.

A *hot spot* is a pair of nodes that accounts for a disproportionately large portion of the total network traffic. Hot-spot traffic can degrade performance of the entire network by causing congestion. The *hot-spot throughput* of a network is the maximum rate at which messages can be sent from one specific node P_i to another specific node P_j.

Low-dimensional networks operate better under nonuniform loads because they have more resource sharing. In a high-dimensional network, wires are assigned to particular dimensions and cannot be shared between dimensions. For example, in a binary n-cube, it is possible for a wire to be saturated while a physically adjacent wire assigned to a different dimension remains idle. In a torus, all physically adjacent wires are combined into a single channel which is shared by all messages.

Minimum network latency is achieved when the network radix k and dimension n

Copyrighted material



(e) A 16×16 Omega network

Figure 2.24 The use of 2×2 switches and perfect shuffle as an interstage connection pattern to construct a 16×16 Omega network. (Courtesy of Duncan Lawrie; reprinted with permission from *IEEE Trans. Computers*, Dec. 1975)

In general, an n-input Omega network requires $\log_2 n$ stages of 2×2 switches. Each stage requires $n/2$ switch modules. In total, the network uses $n \log_2 n/2$ switches. Each switch module is individually controlled.

Various combinations of the switch states implement different permutations, broadcast, or other connections from the inputs to the outputs. The interconnection capabilities of the Omega and other networks will be further studied in Chapter 7.

Baseline Network Wu and Feng (1980) have studied the relationship among a

Copyrighted material

Exercises

Problem 2.1 Define the following terms related to parallelism and dependence relations:

(a) Computational granularity.
(b) Communication latency.
(c) Flow dependence.
(d) Antidependence.
(e) Output dependence.
(f) I/O dependence.
(g) Control dependence.
(h) Resource dependence.
(i) Bernstein conditions.
(j) Degree of parallelism.

Problem 2.2 Define the following terms for various system interconnect architectures:

(a) Node degree.
(b) Network diameter.
(c) Bisection bandwidth.
(d) Static connection networks.
(e) Dynamic connection networks.
(f) Nonblocking networks.
(g) Multicast and broadcast.
(h) Mesh versus torus.
(i) Symmetry in networks.
(j) Multistage networks.
(k) Crossbar networks.
(l) Digital buses.

Problem 2.3 Answer the following questions on program flow mechanisms and computer models:

(a) Compare control-flow, dataflow, and reduction computers in terms of the program flow mechanism used.
(b) Comment on the advantages and disadvantages in control complexity, potential for parallelism, and cost-effectiveness of the above computer models.
(c) What are the differences between string reduction and graph reduction machines?

Problem 2.4 Perform a data dependence analysis on each of the following Fortran program fragments. Show the dependence graphs among the statements with justification.

(a)

```
S1:    A = B + D
S2:    C = A × 3
S3:    A = A + C
S4:    E = A / 2
```

(b)

```
S1:    X = SIN(Y)
S2:    Z = X + W
S3:    Y = −2.5 × W
S4:    X = COS(Z)
```

(c) Determine the data dependences in the same and adjacent iterations of the following Do-loop.

Copyrighted material

equivalence among the Omega, Flip, and Baseline networks.

(a) Prove that the Omega network (Fig. 2.24) is topologically equivalent to the Baseline network (Fig. 2.25b) .

(b) The *Flip network* (Fig. 2.27) is constructed using inverse perfect shuffle (Fig. 2.14b) for interstage connections. Prove that the Flip network is topologically equivalent to the Baseline network.

(c) Based on the results obtained in (a) and (b), prove the topological equivalence between the Flip network and the Omega network.

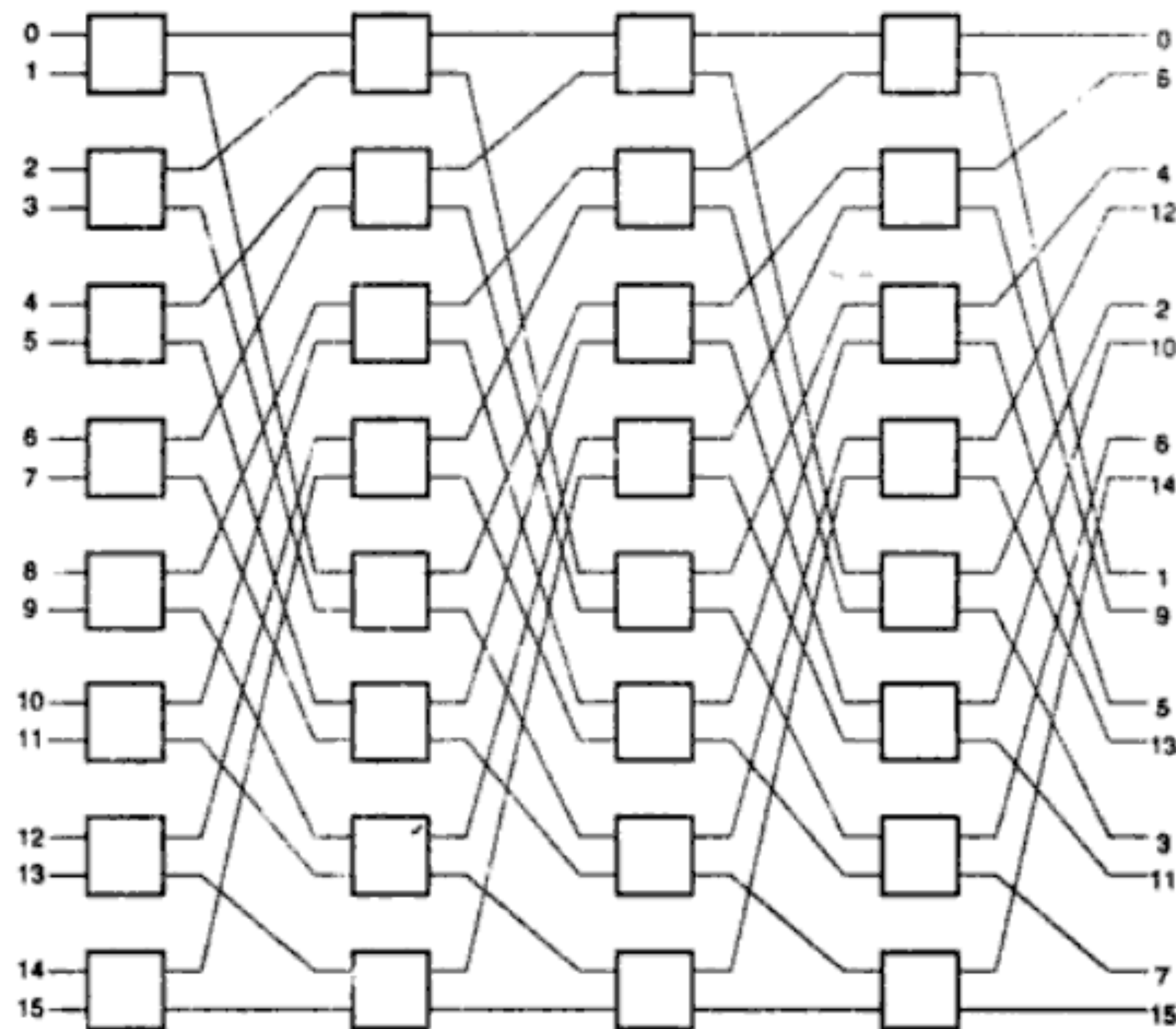


Figure 2.27 A 16×16 **Flip network.** (Courtesy of Ken Batcher; reprinted from *Proc. Int. Conf. Parallel Processing*, 1976)

Problem 2.16 Answer the following questions for the k-ary n-cube network:

(a) How many nodes are there?

(b) What is the network diameter?

(c) What is the bisection bandwidth?

(d) What is the node degree?

(e) Explain the graph-theoretic relationship among k-ary n-cube networks and rings, meshes, tori, binary n-cubes, and Omega networks.

(f) Explain the difference between a conventional torus and a folded torus.

(g) Under the assumption of constant wire bisection, why do low-dimensional networks

Copyrighted material

In discrete form, we have

$$A = \left(\sum_{i=1}^{m} i \cdot t_i\right) \Big/ \left(\sum_{i=1}^{m} t_i\right) \tag{3.4}$$

Example 3.1 Example parallelism profile and average parallelism of a divide-and-conquer algorithm (Sun and Ni, 1993)

As illustrated in Fig. 3.1, the parallelism profile of an example divide-and-conquer algorithm increases from 1 to its peak value $m = 8$ and then decreases to 0 during the observation period (t_1, t_2).

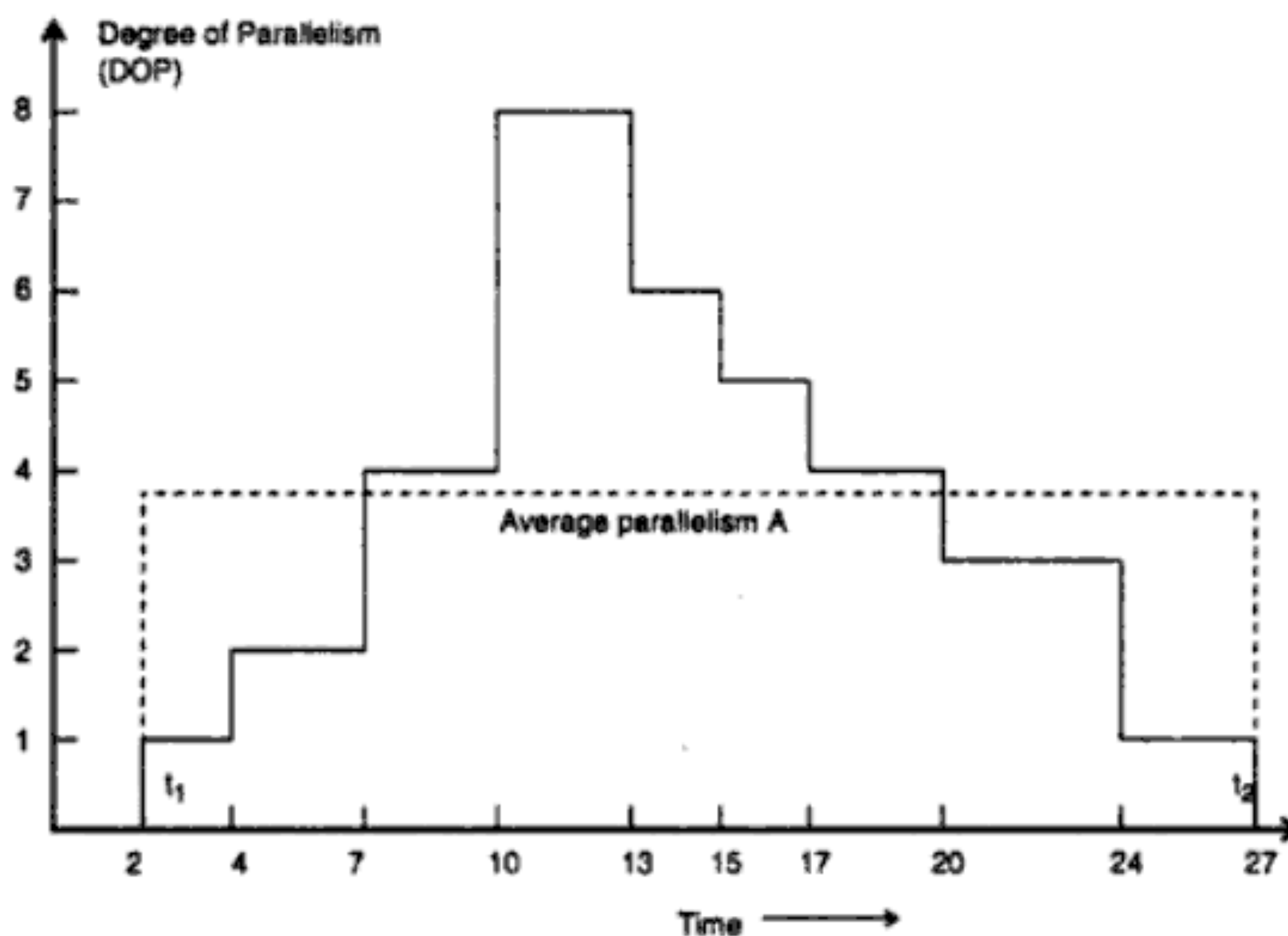


Figure 3.1 Parallelism profile of a divide-and-conquer algorithm.

In Fig. 3.1, the average parallelism $A = (1 \times 5 + 2 \times 3 + 3 \times 4 + 4 \times 6 + 5 \times 2 + 6 \times 2 + 8 \times 3)/(5 + 3 + 4 + 6 + 2 + 2 + 3) = 93/25 = 3.72$. In fact, the total workload $W = A\,\Delta\,(t_2 - t_1)$, and A is an upper bound of the asymptotic speedup to be defined below.

■

Available Parallelism There is a wide range of potential parallelism in application programs. Engineering and scientific codes exhibit a high DOP due to data parallelism. Manoj Kumar (1988) has reported that computation-intensive codes may execute 500 to 3500 arithmetic operations concurrently in each clock cycle in an idealized environment. Nicolau and Fisher (1984) reported that standard Fortran programs averaged about a

Copyrighted material

upper-bounded by $1/\alpha$, regardless of how many processors are employed.

In Fig. 3.3, we plot Eq. 3.16 as a function of n for four values of α. When $\alpha = 0$, the ideal speedup is achieved. As the value of α increases from 0.01 to 0.1 to 0.9, the speedup performance drops sharply.

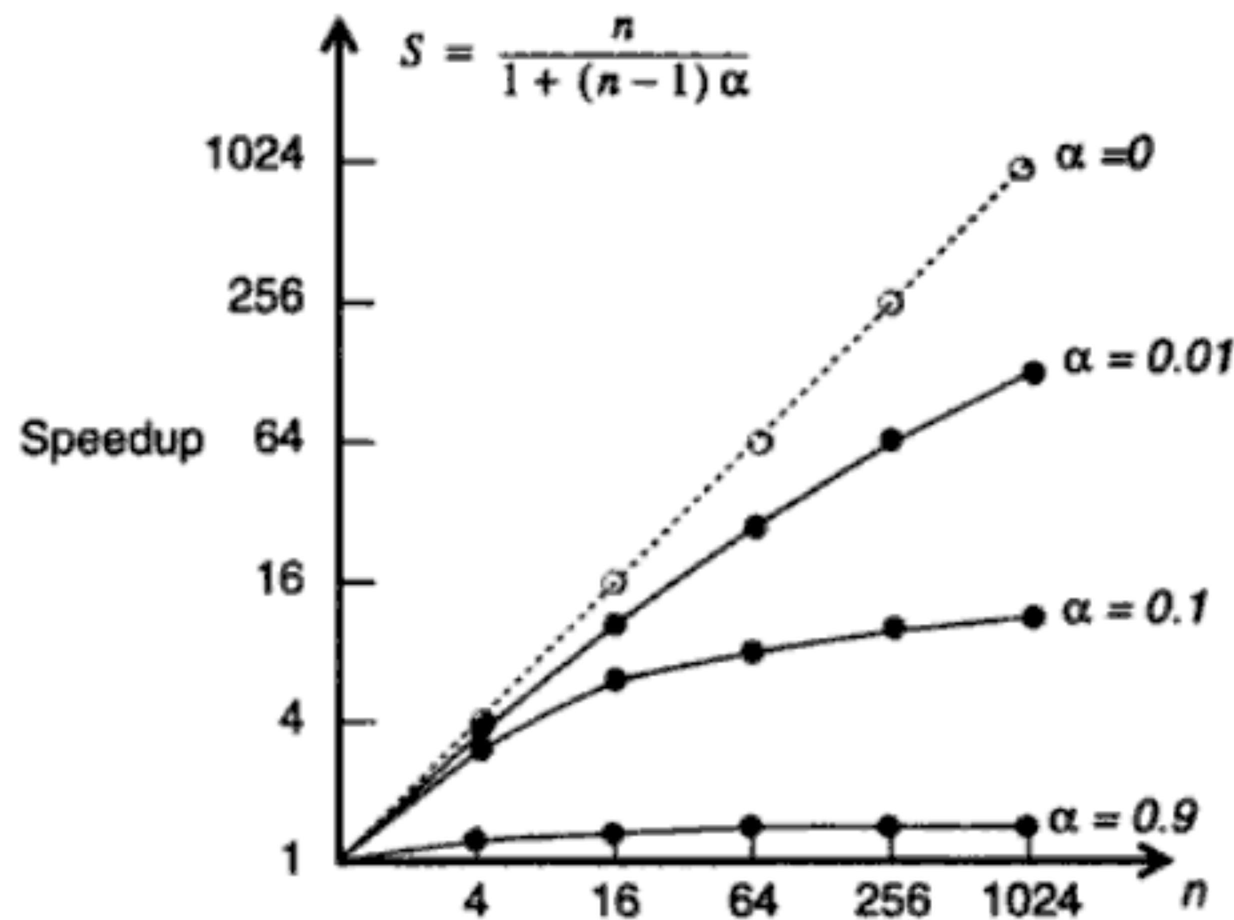


Figure 3.3 Speedup performance with respect to the probability distribution $\pi = (\alpha, 0, ..., 0, 1 - \alpha)$ where α is the fraction of sequential bottleneck

For many years, Amdahl's law has painted a very gloomy and pessimistic picture for parallel processing. That is, the system performance cannot be high as long as the serial fraction α exists. We will further examine Amdahl's law in Section 3.3.1 from the perspective of workload growth.

3.1.3 Efficiency, Utilization, and Quality

Ruby Lee (1980) has defined several parameters for evaluating parallel computations. These are fundamental concepts in parallel processing. Tradeoffs among these performance factors are often encountered in real-life applications.

System Efficiency Let $O(n)$ be the total number of unit operations performed by an n-processor system and $T(n)$ be the execution time in unit time steps. In general, $T(n) < O(n)$ if more than one operation is performed by n processors per unit time, where $n \geq 2$. Assume $T(1) = O(1)$ in a uniprocessor system. The *speedup factor* is defined as

$$S(n) = T(1)/T(n) \tag{3.17}$$

The *system efficiency* for an n-processor system is defined by

$$E(n) = \frac{S(n)}{n} = \frac{T(1)}{nT(n)} \tag{3.18}$$

Copyrighted material

Table 3.2 Dhrystone Version 1.1 Results and Relative MIPS Performance of Some Systems

System	KDhrystone/s	Relative MIPS
VAX 11/780	1.7	1.0
VAX 8600	6.4	3.7
Cray 1S	14.8	8.4
IBM 3081	15.0	8.5
Sun4/260(SPARC)	16.8	9.8
Cray X/MP	18.5	10.5
SPARC, 16.7 MHz	19.0	10.8
IBM 3090/200	31.2	17.8
M88100, 20 MHz	35.7	20.3
MIPS R3000, 25 MHz	43.1	24.5
Intel i860, 33.3 MHz	72.5	41.2
MC68040,25MHz	40.0	23.5
i80486, 25MHz	24.0	14.1
AMD 29000,25MHz	21.1	12.4
NS 32532,25MHz	18.3	10.7
IBM RS/6000,25MHz	60.7	35.7

Source: Weicker (1984), Cocke and Markstein (1990), and National Semiconductor (1988).

The TPS and KLIPS Ratings On-line transaction processing applications demand rapid, interactive processing for a large number of relatively simple transactions. They are typically supported by very large databases. Automated teller machines and airline reservation systems are familiar examples.

The throughput of computers for on-line transaction processing is often measured in *transactions per second* (TPS). Each transaction may involve a database search, query answering, and database update operations. Business computers should be designed to deliver a high TPS rate. The TP1 benchmark was originally proposed in 1985 for measuring the transaction processing of business application computers. This benchmark

Table 3.3 Some Whestone Results, Single Precision, Reported by DEC in January 1986

System	KWhestones/s
HP3000/930	2.84
IBM 4381/11	2.00
DEC 11/780	1.15
Sun 3/50	0.86
IBM RT/PC	0.20

Source: DEC, *Digital Review*, January 1986; all others: company claims.

Copyrighted material

puter Systems, the CM-5 by Thinking Machine Corporation, the KSR-1 by Kendall Square Research, the Fujitsu VPP500 System, the Tera computer, and the MIT *T system.

Recently, IBM announced an MPP project using thousands of IBM RS/6000 processors. Cray Research plans to develop an MPP system using Digital's Alpha processors as building blocks. These MPP projects are summarized in Table 3.5. We will study some of these systems in later chapters.

Table 3.5 Representative Massively Parallel Processing Systems

MPP System	Architecture, Technology, and Operational Features
Intel Paragon	A 2-D mesh-connected multicomputer, built with i860 XP processors and wormhole routers, targeted for 300 Gflops in peak performance.
IBM MPP Model	Use IBM RISC/6000 processors as building blocks, 50 Gflops peak expected for a 1024-processor configuration.
TMC CM-5	A universal architecture for SIMD/MIMD computing using SPARC PEs and custom-designed FPUs, control and data networks, 2 Tflops peak for 16K nodes.
Cray Research MPP Model	A 3D torus heterogeneous architecture using DEC Alpha chips with special communication support, global address space over physically distributed memory; first system offers 150 Gflops in a 1024-processor configuration in 1993; will eventually grow to Tflops with larger configurations.
Kendall Square Research KSR-1	An ALLCACHE ring-connected multiprocessor with custom-designed processors, 43 Gflops peak performance for a 1088-processor configuration.
Fujitsu VPP500	A crossbar-connected 222-PE MIMD vector system, with shared distributed memories using VP2000 as a host; peak performance = 355 Gflops; first delivery in September 1993.

3.2.2 Application Models of Parallel Computers

In general, if the workload or problem size s is kept unchanged as shown by curve α in Fig. 3.6a, then the efficiency E decreases rapidly as the machine size n increases. The reason is that the overhead h increases faster than the machine size. To maintain the efficiency at a desired level, one has to increase the machine size and problem size proportionally. Such a system is known as a *scalable computer* for solving *scaled problems*.

In the ideal case, we like to see a workload curve which is a linear function of n (curve γ in Fig. 3.6a). This implies linear scalability in problem size. If the linear workload curve is not achievable, the second choice is to achieve a sublinear scalability as close to linearity as possible, as illustrated by curve β in Fig. 3.6a.

Suppose that the workload follows an exponential growth pattern and becomes

Copyrighted material

isoefficiency function as follows:

$$f_E(n) = C \times h(s, n) \tag{3.24}$$

If the workload $w(s)$ grows as fast as $f_E(n)$ in Eq. 3.23, then a constant efficiency can be maintained for a given algorithm-architecture combination. Two examples are given below to illustrate the use of isoefficiency functions for scalability analysis.

Example 3.4 Scalability of matrix multiplication algorithms (Gupta and Kumar, 1992)

Four algorithms for matrix multiplication are compared below. The problem size s is represented by the matrix order. In other words, we consider the multiplication of two $s \times s$ matrices A and B to produce an output matrix C = A × B. The total workload involved is $w = O(s^3)$. The number of processors used is confined within $1 \le n \le s^3$. Some of the algorithms may use less than s^3 processors.

Table 3.6 Asymptotic Isoefficiency Functions of Four Matrix Multiplication Algorithms (Gupta and Kumar, 1992)

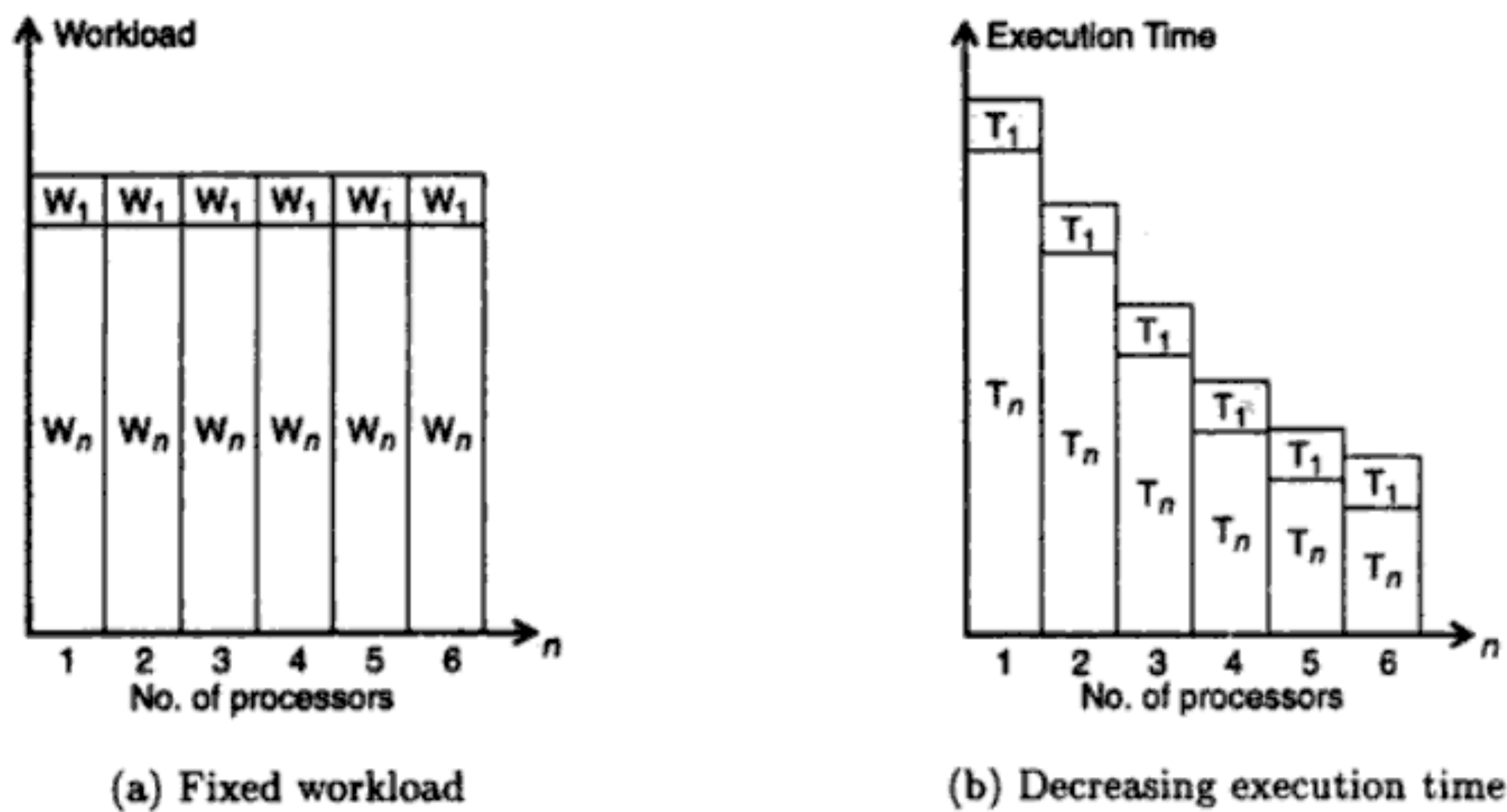
Matrix Multiplication Algorithm	Isoefficiency Function $f_E(n)$	Range of Applicability	Target Machine Architecture
Fox, Otto, and Hey (1987)	$O(n^{3/2})$	$1 \le n \le s^2$	A $\sqrt{n} \times \sqrt{n}$ torus
Berntsen (1989)	$O(n^2)$	$1 \le n \le s^{3/2}$	A hypercube with $n = 2^{3k}$ nodes
Gupta and Kumar (1992)	$O(n(\log n)^3)$	$1 \le n \le s^3$	A hypercube with $n = 2^{3k}$ nodes and $k < \frac{1}{3}\log s$
Dekel, Nassimi, and Sahni (1981)	$O(n \log n)$	$s^2 \le n \le s^3$	A hypercube with $n = s^3 = 2^{3k}$ nodes

Note: Two $s \times s$ matrices are multiplied.

The isoefficiency functions of the four algorithms are derived below based on equating the workload with the communication overhead (Eq. 3.23) in each algorithm. Details of these algorithms and corresponding architectures can be found in the original papers identified in Table 3.6 as well as in the paper by Gupta and Kumar (1992). The derivation of the communication overheads is left as an exercise in Problem 3.14.

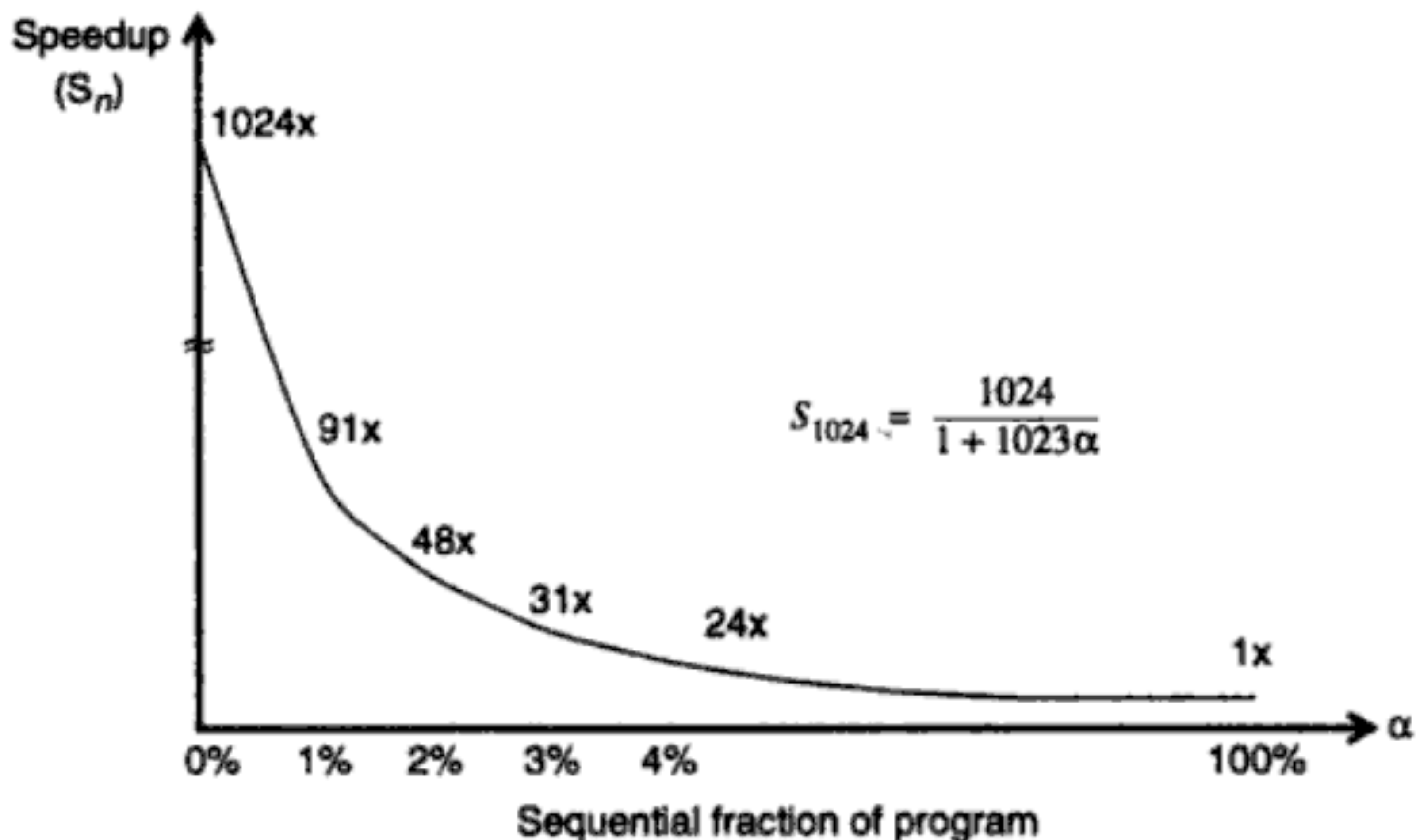
The Fox-Otto-Hey algorithm has a total overhead $h(s,n) = O(n \log n + s^2\sqrt{n})$. The workload $w = O(s^3) = O(n \log n + s^2\sqrt{n})$. Thus we must have $O(s^3) = O(n \log n)$ and $O(s) = O(\sqrt{n})$. Combining the two, we obtain the isoefficiency function $O(s^3) = O(n^{3/2})$, where $1 \le n \le s^2$ as shown in the first row of Table 3.6.

Copyrighted material



(a) Fixed workload

(b) Decreasing execution time



(c) Speedup with a fixed load

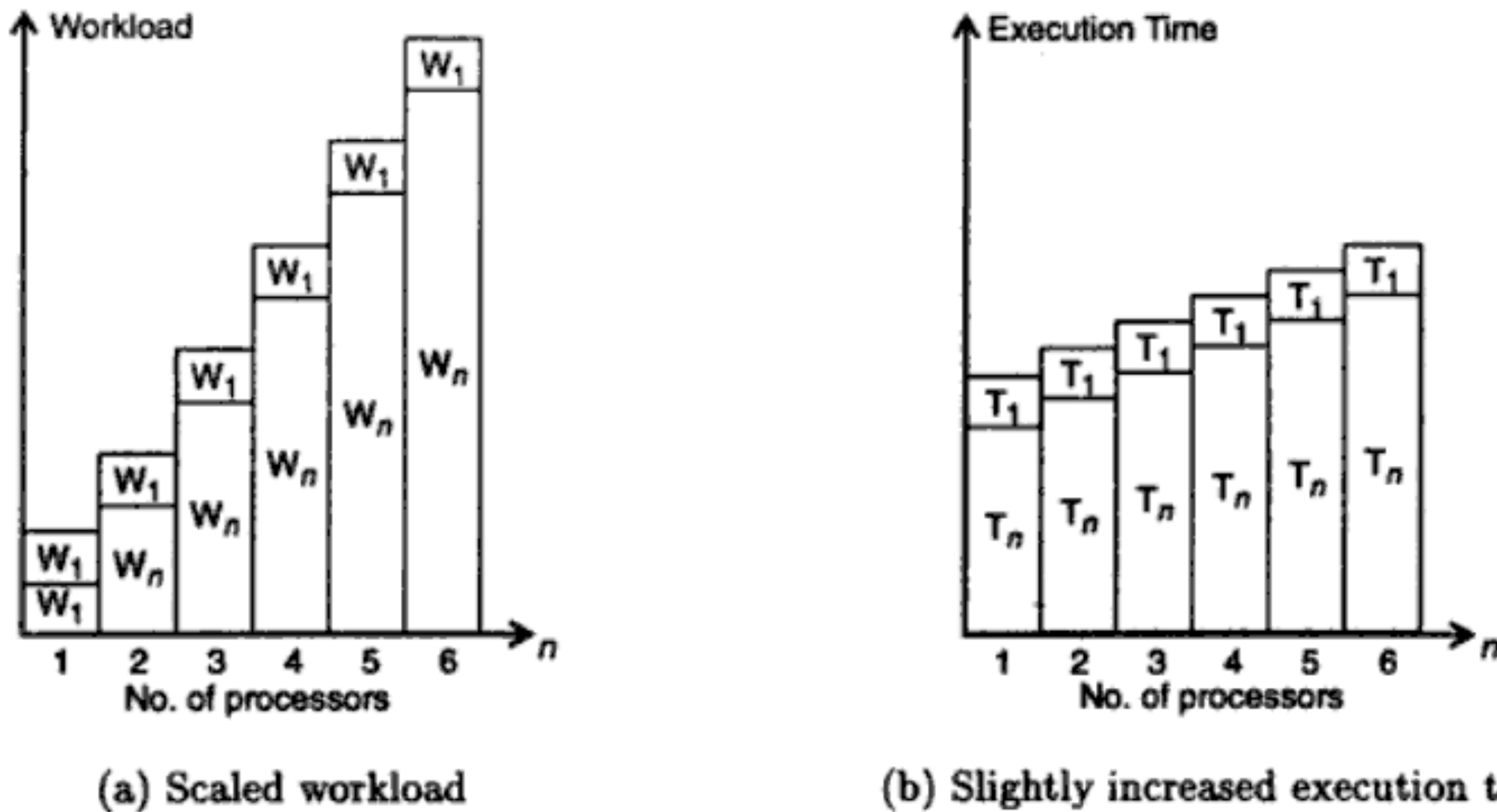
Figure 3.8 Fixed-load speedup model and Amdahl's law.

Scaling for Higher Accuracy Time-critical applications provided the major motivation leading to the development of the fixed-load speedup model and Amdahl's law. There are many other applications that emphasize accuracy more than minimum turnaround time. As the machine size is upgraded to obtain more computing power, we may want to increase the problem size in order to create a greater workload, producing more accurate solution and yet keeping the execution time unchanged.

Many scientific modeling and engineering simulation applications demand the solution of very large-scale matrix problems based on some partial differential equation (PDE) formulations discretized with a huge number of grid points. Representative ex-

Copyrighted material

than the memory requirement, as is often true in some scientific simulation and engineering applications, the fixed-memory model (Fig. 3.10) may yield an even higher speedup (i.e., $S_n^* \geq S_n' \geq S_n$) and better resource utilization.



(a) Scaled workload (b) Slightly increased execution time

Figure 3.10 Scaled speedup model using fixed memory (Courtesy of Sun and Ni; reprinted with permission from *ACM Supercomputing*, 1990)

The fixed-memory model also assumes a scaled workload and allows a slight increase in execution time. The increase in workload (problem size) is memory-bound. The growth in machine size is limited by increasing communication demands as the number of processors becomes very large. The fixed-time model can be moved very close to the fixed-memory model if available memory is fully utilized.

Example 3.6 Scaled matrix multiplication using global versus local computation models (Sun and Ni, 1993)

In scientific computations, matrix often represents some discretized data continuum. Enlarging the matrix size generally leads to a more accurate solution for the continuum. For matrices with dimension n, the number of computations involved in matrix multiplication is $2n^3$ and the memory requirement is roughly $M = 3n^2$.

As the memory increases n times in an n-processor multicomputer system, $nM = n \times 3n^2 = 3n^3$. If the enlarged matrix has a dimension of N, then $3n^3 = 3N^2$. Therefore, $N = n^{1.5}$. Thus $G(n) = n^{1.5}$, and the scaled workload $W_n^* = G(n)W_n = n^{1.5}W_n$. Using Eq. 3.35, we have

$$S^* = \frac{W_1 + n^{1.5}W_n}{W_1 + \frac{n^{1.5}W_n}{n}} = \frac{W_1 + n^{1.5}W_n}{W_1 + n^{0.5}W_n} \tag{3.36}$$

Copyrighted material

where $T_I(s,n)$ is the parallel execution time on the PRAM, ignoring all communication overhead. The scalability is defined as follows:

$$\Phi(s,n) = \frac{S(s,n)}{S_I(s,n)} = \frac{T_I(s,n)}{T(s,n)} \tag{3.40}$$

Intuitively, the larger the scalability, the better the performance that the given architecture can yield running the given algorithm. In the ideal case, $S_I(s,n) = n$, the scalability definition in Eq. 3.40 becomes identical to the efficiency definition given in Eq. 3.39.

Example 3.7 Scalability of various machine architectures for parity calculation (Nussbaum and Agarwal, 1991)

Table 3.7 shows the execution times, asymptotic speedups, and scalabilities (with respect to the EREW-PRAM model) of five representative interconnection architectures: linear array, meshes, hypercube, and Omega network, for running a parallel parity calculation.

Table 3.7 Scalability of Various Network-Based Architectures for the Parity Calculation

Metrics	Machine Architecture				
	Linear array	2-D mesh	3-D mesh	Hypercube	Omega Network
$T(s,n)$	$s^{1/2}$	$s^{1/3}$	$s^{1/4}$	$\log s$	$\log^2 s$
$S(s,n)$	$s^{1/2}$	$s^{2/3}$	$s^{3/4}$	$s / \log s$	$s / \log^2 s$
$\Phi(s,n)$	$\log s / s^{1/2}$	$\log s / s^{1/3}$	$\log s / s^{1/4}$	1	$1 / \log s$

This calculation examines s bits, determining whether the number of bits set is even or odd using a balanced binary tree. For this algorithm, $T(s,1) = s$, $T_I(s,n) = \log s$, and $S_I(s,n) = s/\log s$ for the ideal PRAM machine.

On real architectures, the parity algorithm's performance is limited by network diameter. For example, the linear array has a network diameter equal to $n-1$, yielding a total parallel running time of $s/n+n$. The optimal partition of the problem is to use $n = \sqrt{s}$ processors so that each processor performs the parity check on $\sqrt{s}$ bits locally. This partition gives the best match between computation costs and communication costs with $T(s,n) = s^{1/2}, S(s,n) = s^{1/2}$ and thus scalability $\Phi(s,n) = \log s/s^{1/2}$.

The 2D and 3D mesh architectures use a similar partition to match their own communication structure with the computational loads, yielding even better scala-

Copyrighted material

floating-point number.

For a balanced multicomputer, the computation time within each node and inter-node communication latency should be equal. Thus $0.07n^3\mu s$ equals $6n^2\mu s$ communication latency, implying that n has to be at least as large as 86. A node memory of capacity $86^3 \times 8 = 640\text{K} \times 8 = 5120$ Kwords = 5 megabytes is needed to hold each subdomain of data.

On the other hand, suppose each message exchange takes $2\mu s$ (one receive and one send) per word. The communication latency is doubled. We desire to scale up the problem size with an enlarged local memory of 32 megabytes. The subdomain dimension size n can be extended to at most 160, because $160^3 \times 8 = 32$ megabytes. This size problem requires 0.3 s of computation time and 2×0.15 s of send and receive time. Thus each iteration takes 0.6 $(0.3 + 0.3)$ s, resulting in a computation rate of 50 Mflops, which is only 50% of the peak speed of each node.

If the problem size n is further increased, the effective Mflops rate and efficiency will be improved. But this cannot be achieved unless the memory capacity is further enlarged. For a fixed memory capacity, the situation corresponds to the memory-bound region shown in Fig. 1.10c. Another risk of problem scaling is to exacerbate the limited I/O capability which is not demonstrated in this example.

■

To summarize the above studies on scalability, we realize that the machine size, problem size, and technology scalabilities are not necessarily orthogonal to each other. They must be considered jointly. In the next section, we will identify additional issues relating scalability studies to software compatibility, latency tolerance, machine programmability, and cost-effectiveness.

3.4.3 Research Issues and Solutions

Toward the development of truly scalable computers, much research needs to be done. In this section, we briefly identify several frontier research problems. Partial solutions to these problems will be studied in subsequent chapters.

The Problems When a computer is scaled up to become an MPP system, the following difficulties will arise:

- Memory-access latency becomes too long and too nonuniformly distributed to be considered tolerable.
- The IPC complexity or synchronization overhead becomes too high to be useful.
- The multicache inconsistency problem becomes out of control.
- The processor utilization rate deteriorates as the system size becomes large.
- Message passing (or page migration) becomes too time-consuming to benefit resource sharing in a large distributed system.
- Overall system performance becomes saturated with diminishing return as system size increases further.

Copyrighted material

next 32 iterations ($I = 33$ to 64), and so on. What are the execution time and speedup factors compared with part (a)? (Note that the computational workload, dictated by the J-loop, is unbalanced among the processors.)

(c) Modify the given program to facilitate a balanced parallel execution of all the computational workload over 32 processors. By a balanced load, we mean an equal number of additions assigned to each processor with respect to both loops.

(d) What is the minimum execution time resulting from the balanced parallel execution on 32 processors? What is the new speedup over the uniprocessor?

Problem 3.8 Consider the multiplications of two $n \times n$ matrices $A = (a_{ij})$ and $B = (b_{ij})$ on a scalar uniprocessor and on a multiprocessor, respectively. The matrix elements are floating-point numbers, initially stored in the main memory in row-major. The resulting product matrix $C = (c_{ij})$ where $C = A \times B$, should be stored back to memory in contiguous locations.

Assume a 2-address instruction format and an instruction set of your choice. Each load/store instruction takes, on the average, 4 cycles to complete. All ALU operations must be done sequentially on the processor with 2 cycles if no memory reference is required in the instruction. Otherwise, 4 cycles are added for each memory reference to fetch an operand. Branch-type instructions require, on the average, 2 cycles.

(a) Write a minimal-length assembly-language program to perform the matrix multiplication on a scalar processor with a load-store architecture and floating-point hardware.

(b) Calculate the total instruction count, the total number of cycles needed for the program execution, and the average cycles per instruction (CPI).

(c) What is the MIPS rate of this scalar machine, if the processor is driven by a 40-MHz clock?

(d) Suggest a partition of the above program to execute the divided program parts on an N-processor shared-memory system with minimum time. Assume $n = 1000N$. Estimate the potential speedup of the multiprocessor over the uniprocessor, assuming the same type of processors are used in both systems. Ignore the memory-access conflicts, synchronization and other overheads.

(e) Sketch a scheme to perform distributed matrix computations with distributed data sets on an N-node multicomputer with distributed memory. Each node has a computer equivalent to the scalar processor used in part (a).

(f) Specify the message-passing operations required in part (e). Suppose that, on the average, each message passing requires 100 processor cycles to complete. Estimate the total execution time on the multicomputer for the distributed matrix multiplication. Make appropriate assumptions if needed in your timing analysis.

Problem 3.9 Consider the interleaved execution of the four programs in Problem 1.6 on each of the three machines. Each program is executed in a particular mode with the measured MIPS rating.

Copyrighted material

Based on these trends, the mapping of processors in Fig. 4.1 reflects their implementation during the past decade. As time passes, some of the mapped ranges may move toward the lower right corner of the design space.

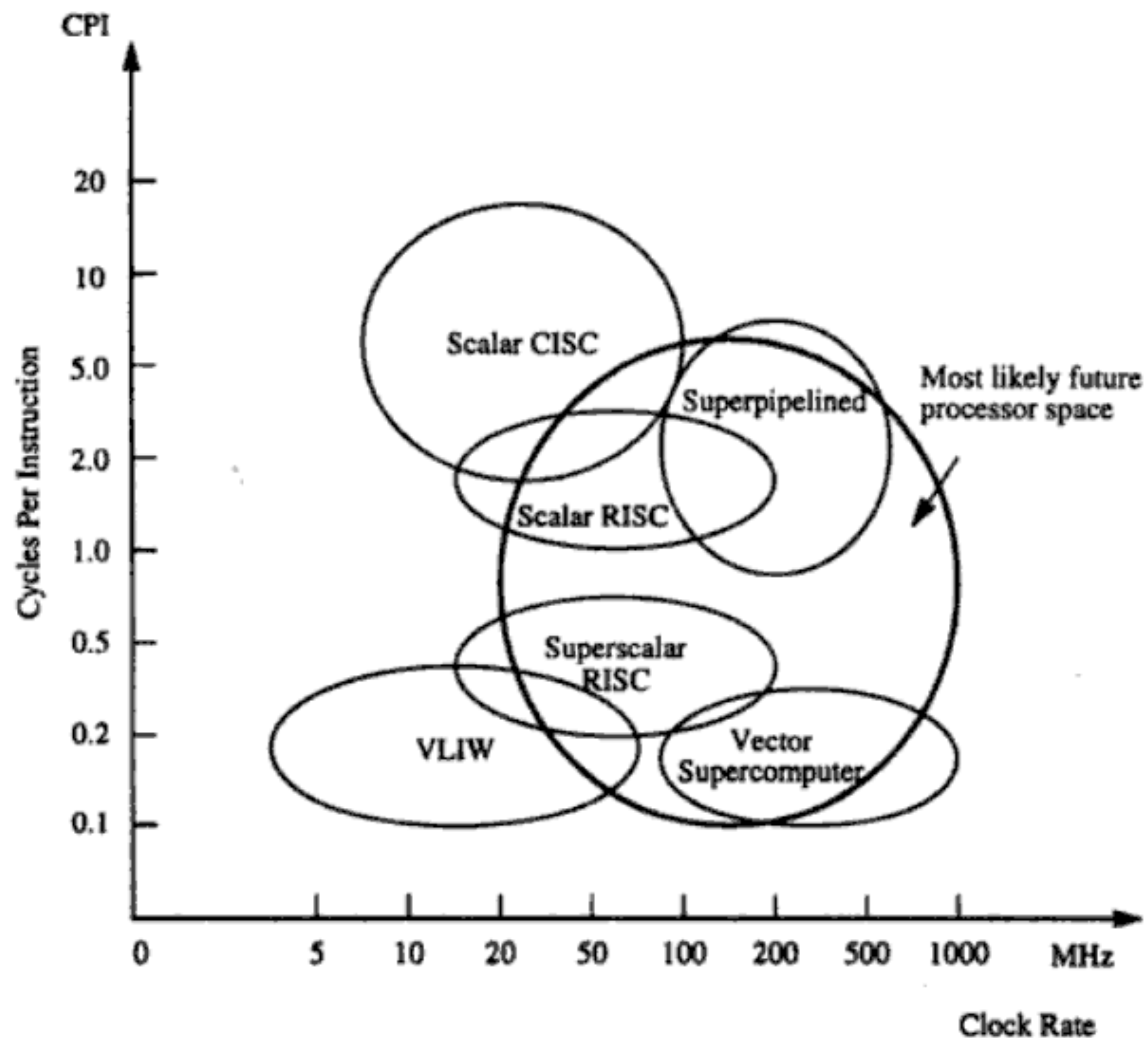


Figure 4.1 Design space of modern processor families.

The Design Space Conventional processors like the Intel i486, M68040, VAX/8600, IBM 390, etc., fall into the family known as *complex-instruction-set computing* (CISC) architecture. The typical clock rate of today's CISC processors ranges from 33 to 50 MHz. With microprogrammed control, the CPI of different CISC instructions varies from 1 to 20. Therefore, CISC processors are at the upper left of the design space.

Today's *reduced-instruction-set computing* (RISC) processors, such as the Intel i860, SPARC, MIPS R3000, IBM RS/6000, etc., have faster clock rates ranging from 20 to 120 MHz determined by the implementation technology employed. With the use of hardwired control, the CPI of most RISC instructions has been reduced to one to two cycles.

A special subclass of RISC processors are the *superscalar processors*, which allow multiple instructions to be issued simultaneously during each cycle. Thus the effective CPI of a superscalar processor should be lower than that of a generic scalar RISC pro-

Copyrighted material

making the instruction set very large and very complex. The growth of instruction sets was also encouraged by the popularity of microprogrammed control in the 1960s and 1970s. Even user-defined instruction sets were implemented using microcodes in some processors for special-purpose applications.

A typical CISC instruction set contains approximately 120 to 350 instructions using variable instruction/data formats, uses a small set of 8 to 24 *general-purpose registers* (GPRs), and executes a large number of memory reference operations based on more than a dozen addressing modes. Many HLL statements are directly implemented in hardware/firmware in a CISC architecture. This may simplify the compiler development, improve execution efficiency, and allow an extension from scalar instructions to vector and symbolic instructions.

Reduced Instruction Sets We started with RISC instruction sets and gradually moved to CISC instruction sets during the 1980s. After two decades of using CISC processors, computer users began to reevaluate the performance relationship between instruction-set architecture and available hardware/software technology.

Through many years of program tracing, computer scientists realized that only 25% of the instructions of a complex instruction set are frequently used about 95% of the time. This implies that about 75% of hardware-supported instructions often are not used at all. A natural question then popped up: Why should we waste valuable chip area for rarely used instructions?

With low-frequency elaborate instructions demanding long microcodes to execute them, it may be more advantageous to remove them completely from the hardware and rely on software to implement them. Even if the software implementation is slow, the net result will be still a plus due to their low frequency of appearance. Pushing rarely used instructions into software will vacate chip areas for building more powerful RISC or superscalar processors, even with on-chip caches or floating-point units.

A RISC instruction set typically contains less than 100 instructions with a fixed instruction format (32 bits). Only three to five simple addressing modes are used. Most instructions are register-based. Memory access is done by load/store instructions only. A large register file (at least 32) is used to improve fast context switching among multiple users, and most instructions execute in one cycle with hardwired control.

Because of the reduction in instruction-set complexity, the entire processor is implementable on a single VLSI chip. The resulting benefits include a higher clock rate and a lower CPI, which lead to higher MIPS ratings as reported on commercially available RISC/superscalar processors.

Architectural Distinctions Hardware features built into CISC and RISC processors are compared below. Figure 4.4 shows the architectural distinctions between modern CISC and traditional RISC. Some of the distinctions may disappear, because future processors may be designed with features from both types.

Conventional CISC architecture uses a unified cache for holding both instructions and data. Therefore, they must share the same data/instruction path. In a RISC processor, separate *instruction* and *data caches* are used with different access paths. However, exceptions do exist. In other words, CISC processors may also use split codes.

Copyrighted material

Table 4.3 Representative CISC Scalar Processors

Feature	Intel i486	Motorola MC68040	NS 32532
Instruction-set size and word length	157 instructions, 32 bits.	113 instructions, 32 bits.	63 instructions, 32 bits.
Addressing modes	12	18	9
Integer unit and GPRs	32-bit ALU with 8 registers.	32-bit ALU with 16 registers.	32-bit ALU with 8 registers.
On-chip cache(s) and MMUs	8-KB unified cache for both code and data.	4-KB code cache 4-KB data cache with separate MMUs.	512-B code cache 1-KB data cache.
Floating-point unit, registers, and function units	On-chip with 8 FP registers adder, multiplier, shifter.	On-chip with 3 pipeline stages, 8 80-bit FP registers.	Off-chip FPU NS 32381, or WTL 3164.
Pipeline stages	5	6	4
Protection level	4	2	2
Memory organization and TLB/ATC entries	Segmented paging with 4 KB/page and 32 entries in TLB.	Paging with 4 or 8 KB/page, 64 entries in each ATC.	Paging with 4 KB/page, 64 entries.
Technology, clock rate, packaging, and year introduced	CHMOS IV, 25 MHz, 33 MHz, 1.2M transistors, 168 pins, 1989.	0.8–μm HCMOS, 1.2M transistors, 20 MHz, 40 MHz, 179 pins, 1990.	1.25–μm CMOS 370K transistors, 30 MHz, 175 pins, 1987.
Claimed performance	24 MIPS at 25 MHz,	20 MIPS at 25 MHz, 30 MIPS at 60 MHz.	15 MIPS at 30 MHz.
Successors to watch	i586, i686.	MC68050, MC68066.	unknown.

modes. The instruction set includes data movement, integer, BCD, and floating-point arithmetic, logical, shifting, bit-field manipulation, cache maintenance, and multiprocessor communications, in addition to program and system control and memory management instructions.

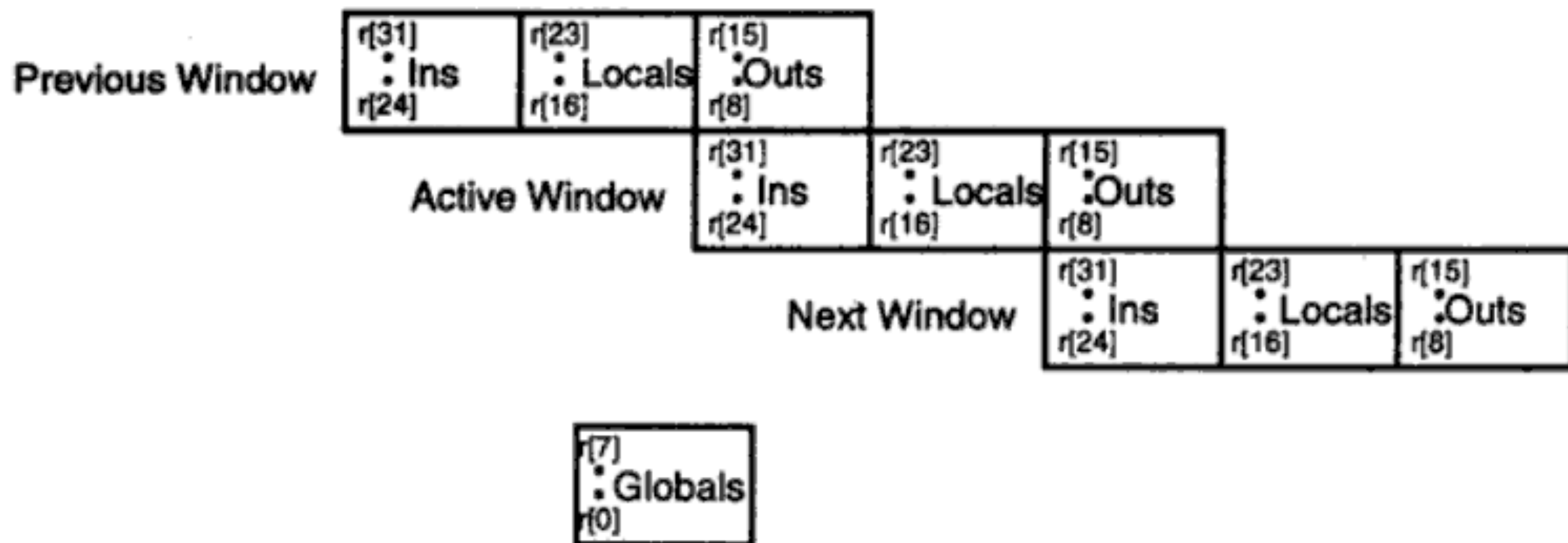
The integer unit is organized in a six-stage instruction pipeline. The floating-point unit consists of three pipeline stages (details to be studied in Section 6.4.1). All instructions are decoded by the integer unit. Floating-point instructions are forwarded to the floating-point unit for execution.

Separate instruction and data buses are used to and from the instruction and data memory units, respectively. Dual MMUs allow interleaved fetch of instructions and data from the main memory. Both the address bus and the data bus are 32 bits wide.

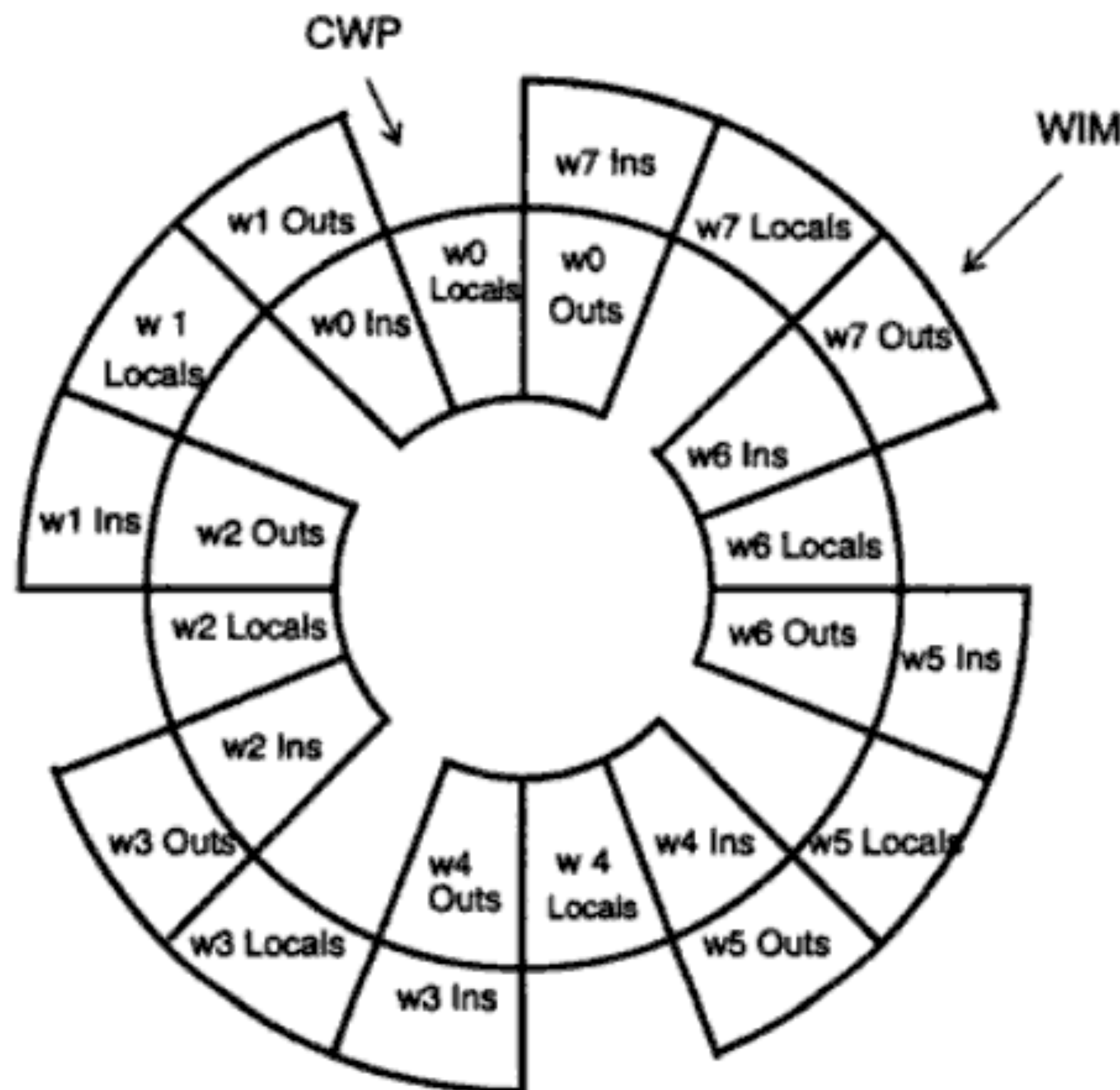
Three simultaneous memory requests can be generated by the dual MMUs,

Copyrighted material

Eight of these registers are *global registers* shared by all procedures, and the remaining 24 are *window registers* associated with only each procedure. The concept of using overlapped register windows is the most important feature introduced by the Berkeley RISC architecture.



(a) Three overlapping register windows and the global registers



(b) Eight register windows forming a circular stack

Figure 4.8 The concept of overlapping register windows in the SPARC architecture. (Courtesy of Sun Microsystems, Inc., 1987)

The concept is illustrated in Fig. 4.8 for eight overlapping windows (formed with 64 local registers and 64 overlapped registers) and eight globals with a total

Copyrighted material

4.2.1 Superscalar Processors

Superscalar processors are designed to exploit more instruction-level parallelism in user programs. Only independent instructions can be executed in parallel without causing a wait state. The amount of instruction-level parallelism varies widely depending on the type of code being executed.

It has been observed that the average value is around 2 for code without loop unrolling. Therefore, for these codes there is not much benefit gained from building a machine that can issue more then three instructions per cycle. The *instruction-issue degree* in a superscalar processor has thus been limited to 2 to 5 in practice.

Pipelining in Superscalar Processors The fundamental structure of a superscalar pipeline is illustrated in Fig. 4.11. This diagram shows the use of three instruction pipelines in parallel for a triple-issue processor. Superscalar processors were originally developed as an alternative to vector processors.

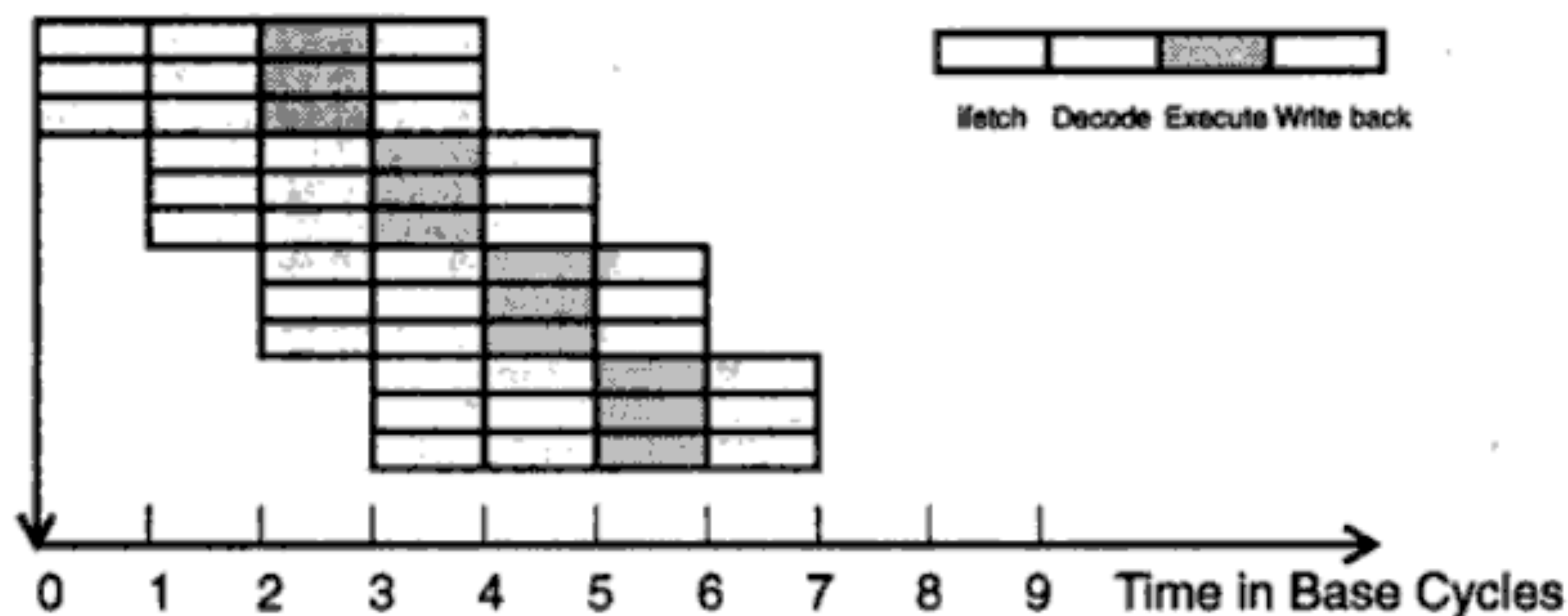


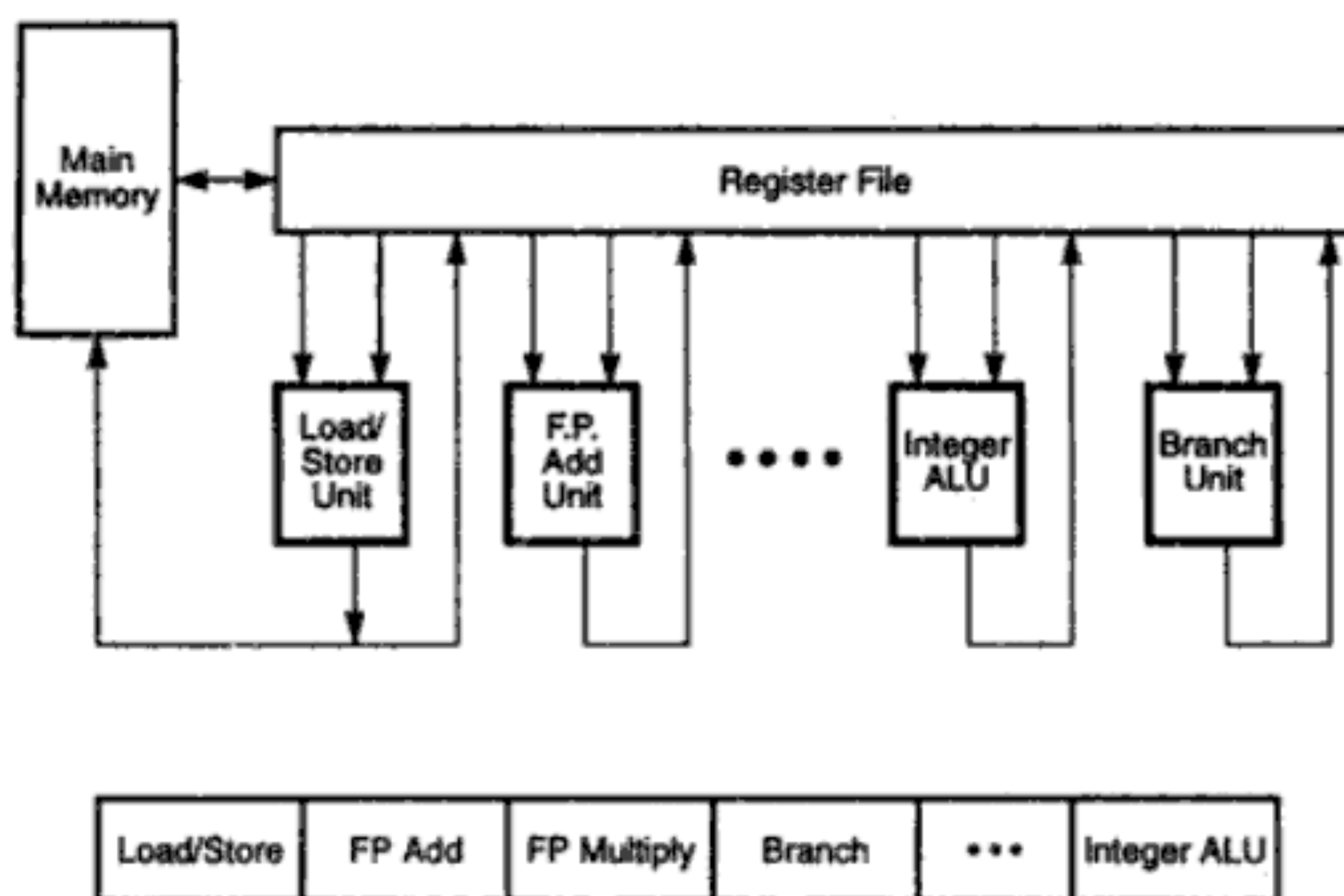
Figure 4.11 A superscalar processor of degree m = 3.

A superscalar processor of degree m can issue m instructions per cycle. In this sense, the base scalar processor, implemented either in RISC or CISC, has $m = 1$. In order to fully utilize a superscalar processor of degree m, m instructions must be executable in parallel. This situation may not be true in all clock cycles. In that case, some of the pipelines may be stalling in a wait state.

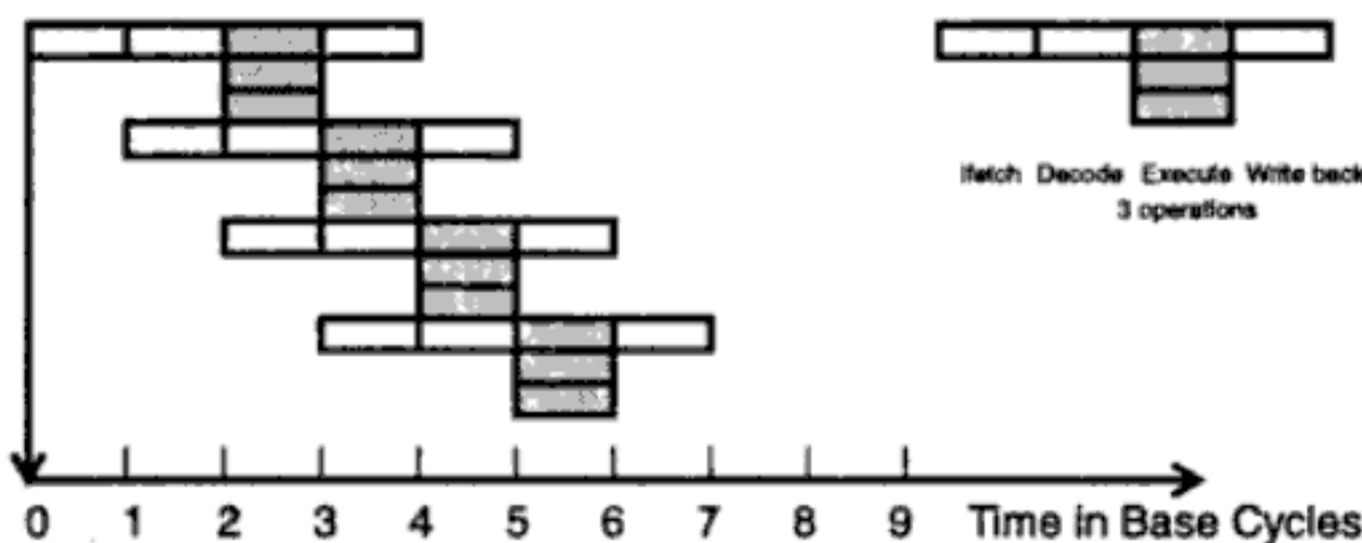
In a superscalar processor, the simple operation latency should require only one cycle, as in the base scalar processor. Due to the desire for a higher degree of instruction-level parallelism in programs, the superscalar processor depends more on an optimizing compiler to exploit parallelism.

In theory, a superscalar processor can attain the same performance as a machine with vector hardware. A superscalar machine that can issue a fixed-point, floating-point, load, and branch all in one cycle achieves the same effective parallelism as a *vector machine* which executes a vector load, chained into a vector add, with one element loaded and added per cycle. This will become more evident as we discuss *pipeline chaining* for vector processors in Chapters 6 and 8. A typical superscalar architecture

Copyrighted material



(a) A typical VLIW processor and instruction format



(b) VLIW execution with degree $m = 3$

Figure 4.14 The architecture of a very long instruction word (VLIW) processor and its pipeline operations. (Courtesy of Multiflow Computer, Inc., 1987)

to seven operations to be executed concurrently with 256 bits per VLIW instruction.

VLIW Opportunities In a VLIW architecture, random parallelism among scalar operations is exploited instead of regular or synchronous parallelism as in a vectorized supercomputer or in an SIMD computer. The success of a VLIW processor depends heavily on the efficiency in code compaction. The architecture is totally incompatible with that of any conventional general-purpose processor.

Furthermore, the instruction parallelism embedded in the compacted code may require a different latency to be executed by different functional units even though the instructions are issued at the same time. Therefore, different implementations of the same VLIW architecture may not be binary-compatible with each other.

By explicitly encoding parallelism in the long instruction, a VLIW processor can eliminate the hardware or software needed to detect parallelism. The main advantage

Copyrighted material

The Symbolics 3600 executes most Lisp instructions in one machine cycle. Integer instruction fetch operands form the stack buffer and the duplicate top of the stack in the scratch-pad memory. Floating-point addition, garbage collection, data type checking by the tag processor, and fixed-point addition can be carried out in parallel.

■

4.3 Memory Hierarchy Technology

In a typical computer configuration, the cost of memory, disks, printers, and other peripherals has far exceeded that of the central processor. We briefly introduce below the memory hierarchy and peripheral technology.

4.3.1 Hierarchical Memory Technology

Storage devices such as *registers*, *caches*, *main memory*, *disk devices*, and *tape units* are often organized as a hierarchy as depicted in Fig. 4.17. The memory technology and storage organization at each level are characterized by five parameters: the *access time* (t_i), *memory size* (s_i), *cost per byte* (c_i), *transfer bandwidth* (b_i), and *unit of transfer* (x_i).

The access time t_i refers to the round-trip time from the CPU to the ith-level memory. The memory size s_i is the number of bytes or words in level i. The cost of the ith-level memory is estimated by the product $c_i s_i$. The bandwidth b_i refers to the rate at which information is transferred between adjacent levels. The unit of transfer x_i refers to the grain size for data transfer between levels i and $i+1$.

Memory devices at a lower level are faster to access, smaller in size, and more expensive per byte, having a higher bandwidth and using a smaller unit of transfer as compared with those at a higher level. In other words, we have $t_{i-1} < t_i$, $s_{i-1} < s_i$, $c_{i-1} > c_i$, $b_{i-1} > b_i$, and $x_{i-1} < x_i$, for $i = 1, 2, 3$, and 4, in the hierarchy where $i = 0$ corresponds to the CPU register level. The cache is at level 1, main memory at level 2, the disks at level 3, and the tape unit at level 4. The physical memory design and operations of these levels are studied in subsequent sections and in Chapter 5.

Registers and Caches The register and the cache are parts of the processor complex, built either on the processor chip or on the processor board. Register assignment is often made by the compiler. Register transfer operations are directly controlled by the processor after instructions are decoded. Register transfer is conducted at processor speed, usually in one clock cycle.

Therefore, many designers would not consider registers a level of memory. We list them here for comparison purposes. The cache is controlled by the MMU and is programmer-transparent. The cache can also be implemented at one or multiple levels, depending on the speed and application requirements.

Main Memory The main memory is sometimes called the primary memory of a computer system. It is usually much larger than the cache and often implemented by

Copyrighted material

cache, copies of that word must be updated immediately or eventually at all higher levels. The hierarchy should be maintained as such. Frequently used information is often found in the lower levels in order to minimize the effective access time of the memory hierarchy. In general, there are two strategies for maintaining the coherence in a memory hierarchy.

The first method is called *write-through* (WT), which demands immediate update in M_{i+1} if a word is modified in M_i, for $i = 1, 2, ..., n-1$.

The second method is *write-back* (WB), which delays the update in M_{i+1} until the word being modified in M_i is replaced or removed from M_i. Memory replacement policies are studied in Section 4.4.3.

Locality of References The memory hierarchy was developed based on a program behavior known as *locality of references*. Memory references are generated by the CPU for either instruction or data access. These accesses tend to be clustered in certain regions in time, space, and ordering.

In other words, most programs act in favor of a certain portion of their address space at any time window. Hennessy and Patterson (1990) have pointed out a 90-10 rule which states that a typical program may spend 90% of its execution time on only 10% of the code such as the innermost loop of a nested looping operation.

There are three dimensions of the locality property: *temporal*, *spatial*, and *sequential*. During the lifetime of a software process, a number of pages are used dynamically. The references to these pages vary from time to time, however, they follow certain access patterns as illustrated in Fig. 4.19. These memory reference patterns are caused by the following locality properties:

(1) *Temporal locality* — Recently referenced items (instructions or data) are likely to be referenced again in the near future. This is often caused by special program constructs such as iterative loops, process stacks, temporary variables, or subroutines. Once a loop is entered or a subroutine is called, a small code segment will be referenced repeatedly many times. Thus temporal locality tends to cluster the access in the recently used areas.

(2) *Spatial locality* — This refers to the tendency for a process to access items whose addresses are near one another. For example, operations on tables or arrays involve accesses of a certain clustered area in the address space. Program segments, such as routines and macros, tend to be stored in the same neighborhood of the memory space.

(3) *Sequential locality* — In typical programs, the execution of instructions follows a sequential order (or the program order) unless branch instructions create out-of-order executions. The ratio of in-order execution to out-of-order execution is roughly 5 to 1 in ordinary programs. Besides, the access of a large data array also follows a sequential order.

Memory Design Implications The sequentiality in program behavior also contributes to the spatial locality because sequentially coded instructions and array elements are often stored in adjacent locations. Each type of locality affects the design of

Copyrighted material

The efficiency of the address translation process affects the performance of the virtual memory. Processor support is needed for precise interrupts and translator updates. Virtual memory is more difficult to implement in a multiprocessor, where additional problems such as coherence, protection, and consistency become much more involved. Two virtual memory models are studied below.

Private Virtual Memory The first model uses a *private virtual memory space* associated with each processor, as seen in the VAX/11 and in most UNIX systems (Fig. 4.20a). Each private virtual space is divided into pages. Virtual pages from different virtual spaces are mapped into the same physical memory shared by all processors.

The advantages of using private virtual memory include the use of a small processor address space (32 bits), protection on each page or on a per-process basis, and the use of private memory maps, which require no locking.

The shortcoming lies in the *synonym problem*, in which different virtual addresses in different/same virtual spaces point to the same physical page. Another problem is that the same virtual address in different virtual spaces may point to different pages in the main memory.

Shared Virtual Memory This model combines all the virtual address spaces into a single globally *shared virtual space* (Fig. 4.20b). Each processor is given a portion of the shared virtual memory to declare their addresses. Different processors may use disjoint spaces. Some areas of virtual space can be also shared by multiple processors.

Examples of machines using shared virtual memory include the IBM801, RT, RP3, System 38, the HP Spectrum, the Stanford Dash, MIT Alewife, Tera, etc. We will further study shared virtual memory in Chapter 9. Until then, all virtual memory systems discussed are assumed private unless otherwise specified.

The advantages in using shared virtual memory include the fact that all addresses are unique. However, each processor must be allowed to generate addresses larger than 32 bits, such as 46 bits for a 64-Tbyte (2^{46}-byte) address space. Synonyms are not allowed in a globally shared virtual memory.

The page table must allow shared accesses. Therefore, *mutual exclusion* (locking) is needed to enforce protected access. Segmentation is built on top of the paging system to confine each process to its own address space (segments). Global virtual memory makes the address translation process even longer.

4.4.2 TLB, Paging, and Segmentation

Both the virtual memory and physical memory are partitioned into fixed-length pages as illustrated in Fig. 4.18. The purpose of memory allocation is to allocate *pages* of virtual memory to the *page frames* of the physical memory.

Address Translation Mechanisms The process demands the translation of virtual addresses into physical addresses. Various schemes for virtual address translation are summarized in Fig. 4.21a. The translation demands the use of *translation maps* which can be implemented in various ways.

Copyrighted material

that address and, if it is found, uses the table index of the matching entry as the address of the desired page frame. A hashing table is used to search through the inverted PT. The size of an inverted PT is governed by the size of the physical space, while that of traditional PTs is determined by the size of the virtual space. Because of limited physical space, no multiple levels are needed for the inverted page table.

Example 4.8 Paging and segmentation in the Intel i486 processor

As with its predecessor in the 80x86 family, the i486 features both segmentation and paging compatibilities. Protected mode increases the linear address from 4 Gbytes (2^{32} bytes) to 64 Tbytes (2^{46} bytes) with four levels of protection. The maximal memory size in real mode is 1 Mbyte (2^{20} bytes). Protected mode allows the i486 to run all software from existing 8086, 80286, and 80386 processors. A segment can have any length from 1 byte to 4 Gbytes, the maximal physical memory size.

A segment can start at any base address, and storage overlapping between segments is allowed. The virtual address (Fig. 4.22a) has a 16-bit segment selector to determine the base address of the *linear address space* to be used with the i486 paging system.

The 32-bit offset specifies the internal address within a segment. The segment descriptor is used to specify access rights and segment size besides selection of the address of the first byte of the segment.

The paging feature is optional on the i486. It can be enabled or disabled by software control. When paging is enabled, the virtual address is first translated into a linear address and then into the physical address. When paging is disabled, the linear address and physical address are identical. When a 4-Gbyte segment is selected, the entire physical memory becomes one large segment, which means the segmentation mechanism is essentially disabled.

In this sense, the i486 can be used with four different memory organizations, *pure paging*, *pure segmentation*, *segmented paging*, or *pure physical addressing* without paging and segmentation.

A 32-entry TLB (Fig 4.22b) is used to convert the linear address directly into the physical address without resorting to the two-level paging scheme (Fig 4.22c). The standard page size on the i486 is 4 Kbytes = 2^{12} bytes. Four control registers are used to select between regular paging and page fault handling.

The page table directory (4 Kbytes) allows 1024 page directory entries. Each page table at the second level is 4 Kbytes and holds up to 1024 PTEs. The upper 20 linear address bits are compared to determine if there is a hit. The hit ratios of the TLB and of the page tables depend on program behavior and the efficiency of the update (page replacement) policies. A 98% hit ratio has been observed in TLB operations in the past.

Copyrighted material

Different cache organizations (Section 5.1) may offer different flexibilities in implementing some of the replacement algorithms. The cache memory is often associatively searched, while the main memory is randomly addressed.

Due to the difference between page allocation in main memory and block allocation in the cache, the cache hit ratio and memory page hit ratio are affected by the replacement policies differently. *Cache traces* are often needed to evaluate the cache performance. These considerations will be further discussed in Chapter 5.

4.5 Bibliographic Notes and Exercises

Advanced microprocessors were surveyed in the book by [Tabak91]. A tutorial on RISC computers was given by [Stallings90]. Superscalar and superpipelined machines were characterized by Jouppi and Wall [Jouppi89]. [Johnson91] provided an excellent book on superscalar processor design. The VLIW architecture was first developed by [Fisher83].

The Berkeley RISC was reported by Patterson and Sequin in [Patterson82]. A MIPS R2000 overview can be found in [Kane88]. The HP precision architecture has been assessed in [Lee89]. Optimizing compilers for SPARC appears in [Muchnick88]. A further description of the M68040 can be found in [Edenfield90].

A good source of information on the i860 can be found in [Margulis90]. The DEC Alpha architecture is described in [DEC92]. The latest MIPS R4000 was reported by Mirapuri, Woodacre, and Vasseghi [Mirapuri92]. The IBM RS/6000 was discussed in [IBM90]. The Hot-Chips Symposium Series [Hotchips91] usually present the latest developments in high-performance processor chips.

The virtual memory models were based on the tutorial by Dubois and Briggs [Dubois90c]. The book by [Hennessy90] and Patterson has treated the memory hierarchy design based on extensive program trace data. Distributed shared virtual memory was surveyed in [Nitzberg91] and Lo.

The concept of a working set was introduced by [Denning68]. A linear programming optimization of the memory hierarchy was reported in [Chow74]. [Crawford90] explained the paging and segmentation schemes in the i486. Inverted paging was described by [Chang88] and Mergen. [Cragon92b] has discussed memory systems for pipeline processor design.

Exercises

Problem 4.1 Define the following basic terms related to modern processor technology:

(a) Processor design space.
(b) Instruction issue latency.
(c) Instruction issue rate.
(d) Simple operation latency.
(e) Resource conflicts.
(f) Processor versus coprocessor.
(g) General-purpose registers.
(h) Addressing modes.
(i) Unified versus split caches.
(j) Hardwired versus microcoded control.

Copyrighted material

Chapter 5

Bus, Cache, and Shared Memory

This chapter describes the design and operational principles of bus, cache, and shared-memory organization. Backplane bus systems are studied first, including the latest Futurebus+ specifications. Cache addressing models and implementation schemes are described. We study memory interleaving, allocation schemes, and the sequential and weak consistency models for shared-memory systems. Other relaxed memory consistency models are given in Chapter 9.

5.1 Backplane Bus Systems

The system bus of a computer system operates on a contention basis. Several active devices such as processors may request use of the bus at the same time. However, only one of them can be granted access at a time. The *effective bandwidth* available to each processor is inversely proportional to the number of processors contending for the bus.

For this reason, most bus-based commercial multiprocessors are small in size. The simplicity and low cost of a bus system have made it attractive in building small multiprocessors ranging from 4 to 16 processors based on today's technology.

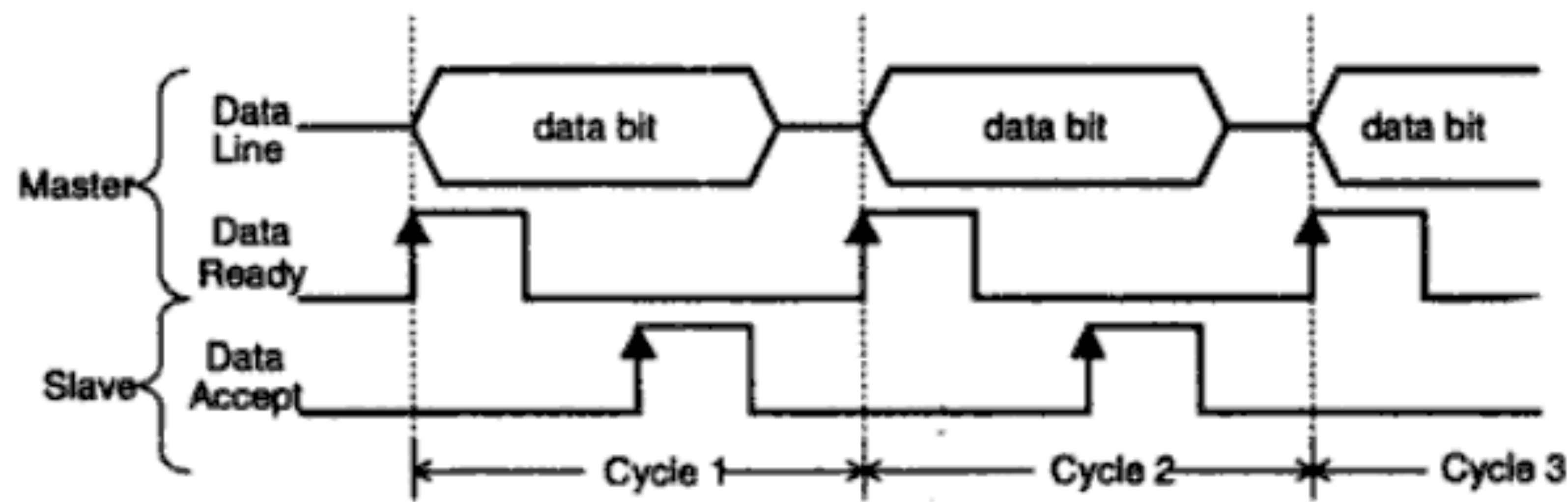
In this section, we specify system buses which are confined to a single backplane chassis. We concentrate on logical specification instead of physical implementation. Standard bus specifications should be both technology-independent and architecture-independent.

5.1.1 Backplane Bus Specification

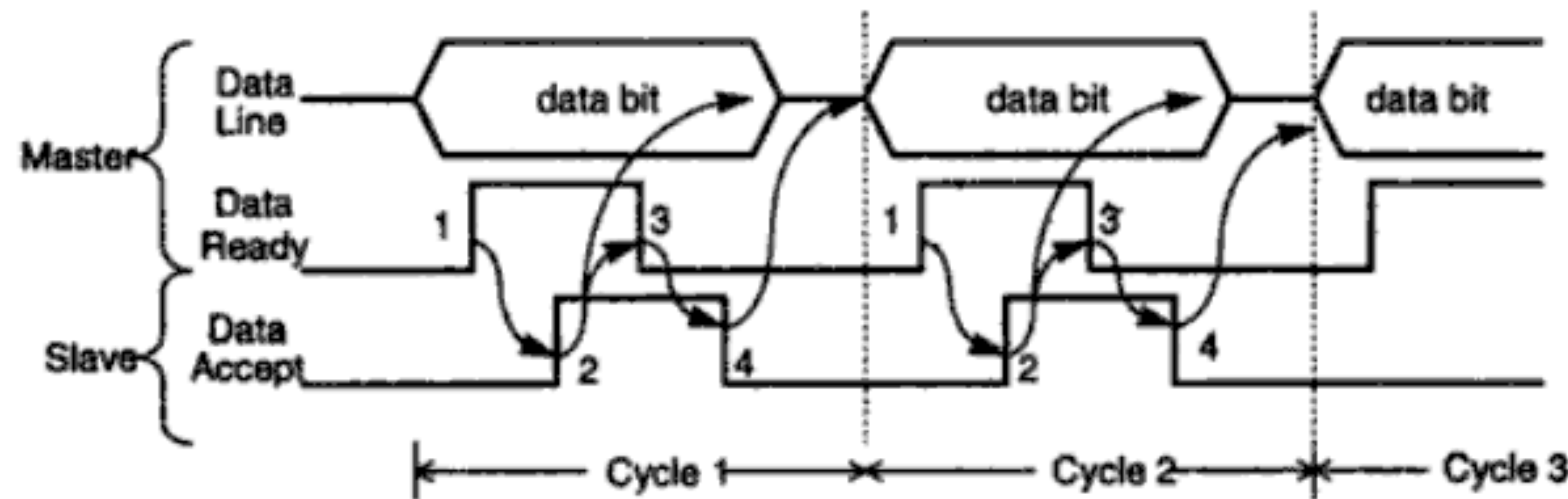
A backplane bus interconnects processors, data storage, and peripheral devices in a tightly coupled hardware configuration. The system bus must be designed to allow communication between devices on the bus without disturbing the internal activities of all the devices attached to the bus. Timing protocols must be established to arbitrate among multiple requests. Operational rules must be set to ensure orderly data transfers on the bus.

Signal lines on the backplane are often functionally grouped into several buses as

Copyrighted material



(a) Synchronous bus timing with fixed-length clock signals for all devices



(b) Asynchronous bus timing using a four-edge handshaking (interlocking) with variable-length signals for different-speed devices

Figure 5.3 Synchronous versus asynchronous bus timing protocols.

5.1.3 Arbitration, Transaction, and Interrupt

The process of selecting the next bus master is called *arbitration*. The duration of a master's control of the bus is called *bus tenure*. This arbitration process is designed to restrict tenure of the bus to one master at a time. Competing requests must be arbitrated on a fairness or priority basis.

Arbitration competition and bus transactions may take place concurrently on a parallel bus with separate lines for both purposes.

Central Arbitration As illustrated in Fig. 5.4a, a central arbitration scheme uses a central arbiter. Potential masters are daisy-chained in a cascade. A special signal line is used to propagate a *bus-grant* signal level from the first master (at slot 1) to the last master (at slot n).

Each potential master can send a bus request. However, all requests share the same *bus-request* line. As shown in Fig. 5.4b, the *bus-request* signals the rise of the *bus-grant* level, which in turn raises the *bus-busy* level.

A fixed priority is set in a daisy chain from left to right. Only when the devices on the left do not request bus control can a device be granted bus tenure. When the bus transaction is complete, the *bus-busy* level is lowered, which triggers the falling of the *bus grant* signal and the subsequent rising of the *bus-request* signal.

The advantage of this arbitration scheme is its simplicity. Additional devices can

Copyrighted material

(10) Direct support of snoopy cache-based multiprocessors with recursive protocols to support large systems interconnected by multiple buses.

(11) Compatible message-passing protocols with multicomputer connections and special application profiles and interface design guides provided.

Example 5.1 Signal lines in the proposed Futurebus+ standard

As illustrated in Fig. 5.6, the Futurebus+ consists of information, synchronization, bus arbitration, and handshake lines that can match different system designs. The 64-bit *address lines* are multiplexed with the lower-order 64-bit *data lines*. Additional data lines can be added to form a data path up to 256 bits wide. The *tag lines* are optional for extending the addressing/data modes.

The *command lines* carry command information from the master to one or more slaves. The *status lines* are used by slaves to respond to the master's commands. The *capability lines* are used to declare special bus transactions. Every byte of lines is protected by at least one parity-check line.

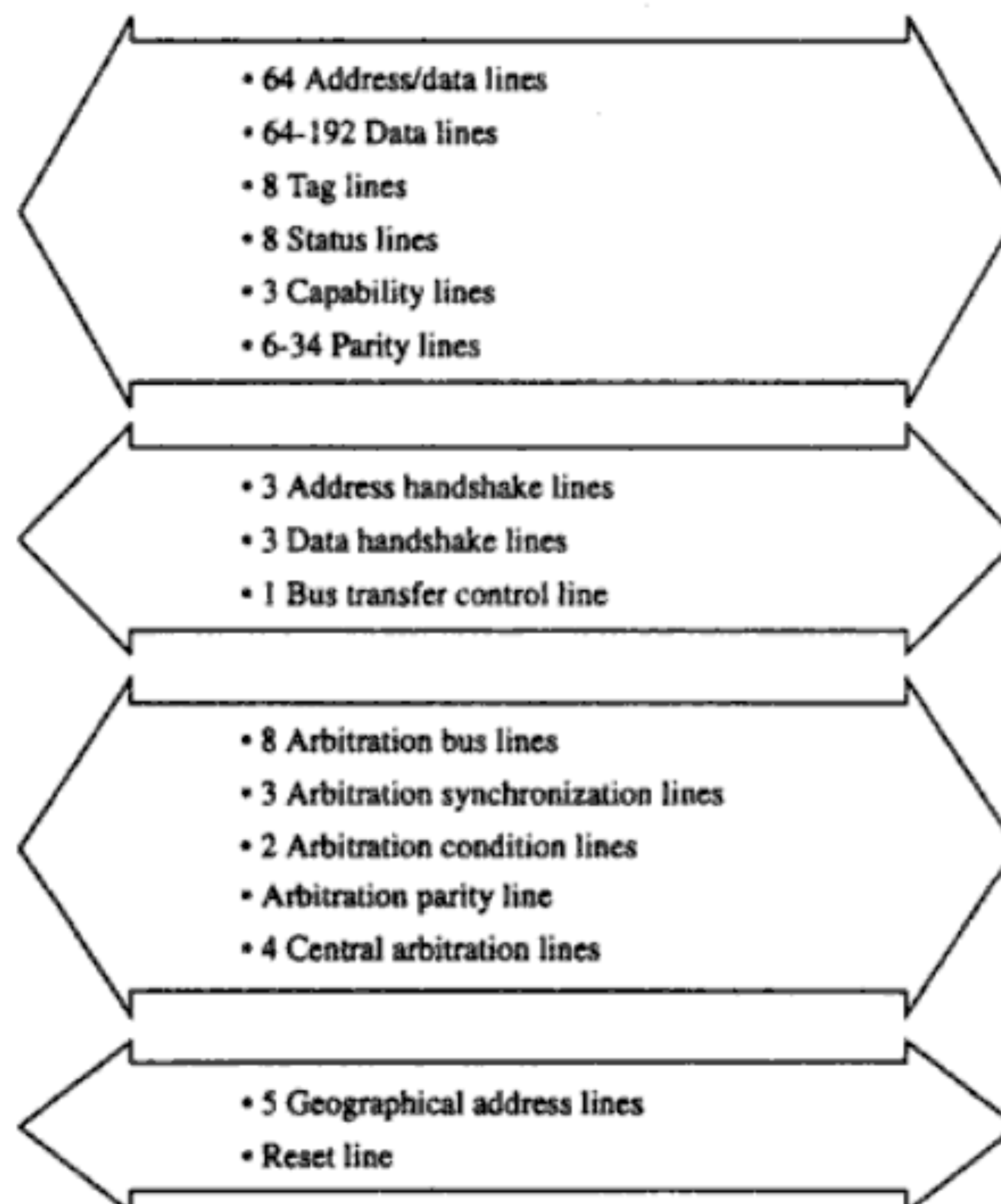
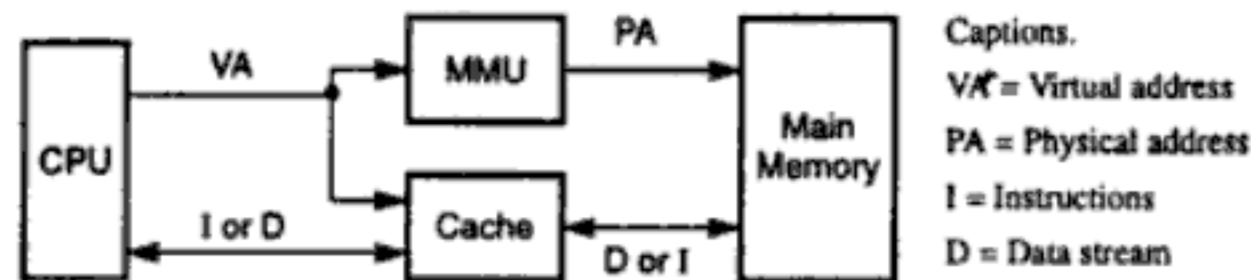


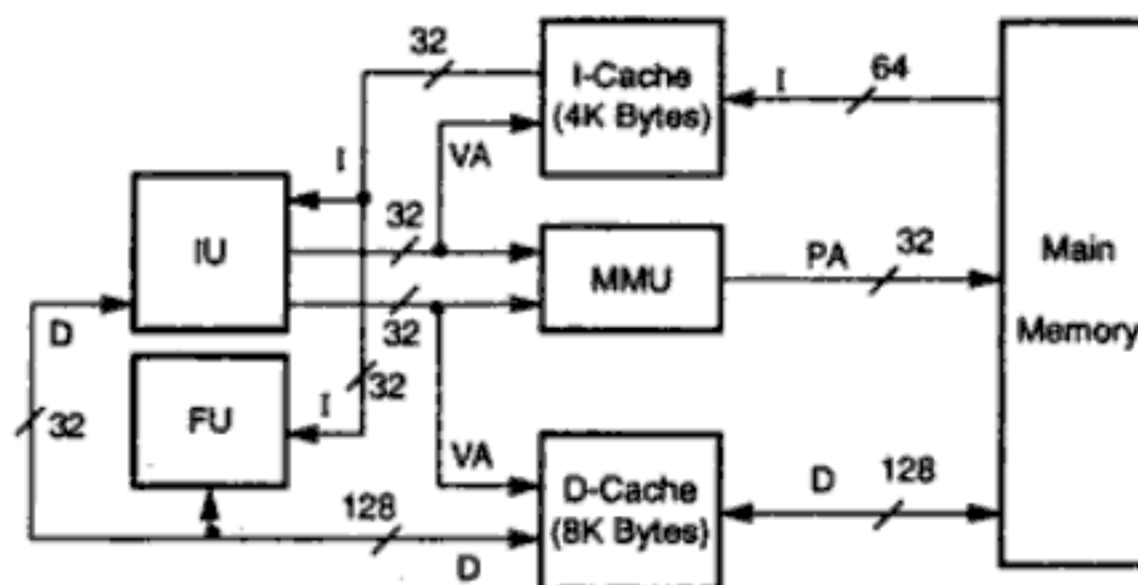
Figure 5.6 The Futurebus+ organization. (Reprinted with permission from IEEE Standard 896.1-1991, copyright © 1991 by IEEE, Inc.)

Copyrighted material

Virtual Address Caches When a cache is indexed or tagged with a virtual address as shown in Fig. 5.9, it is called a *virtual address cache*. In this model, both cache and MMU translation or validation are done in parallel. The physical address generated by the MMU can be saved in tags for later write-back but is not used during the cache lookup operations. The virtual address cache is motivated with its enhanced efficiency to access the cache faster, overlapping with the MMU translation as exemplified below.



(a) A unified cache accessed by virtual address



(b) A split cache accessed by virtual address as in the Intel i860 processor

Figure 5.9 Virtual address models for unified and split caches. (Courtesy of Intel Corporation, 1989)

Example 5.3 The virtual addressed split cache design in Intel i860

Figure 5.9b shows the virtual address design in the Intel i860 using split caches for data and instructions. Instructions are 32 bits wide. Virtual addresses generated by the integer unit (IU) are 32 bits wide, and so are the physical addresses generated by the MMU. The data cache is 8 Kbytes with a block size of 32 bytes. A two-way, set-associative cache organization (Section 5.2.3) is implemented with 128 sets in the D-cache and 64 sets in the I-cache.

■

The Aliasing Problem The major problem associated with a virtual address cache is *aliasing*, when different logically addressed data have the same index/tag in the cache. Multiple processes may use the same range of virtual addresses. This aliasing problem may create confusion if two or more processes access the same physical cache location.

Copyrighted material

sequentially using RAMs. Thus an *associative memory* (content-addressable memory, CAM) is needed to achieve a parallel comparison with all tags simultaneously. This demands a higher implementation cost for the cache. Therefore, a fully associative cache has been implemented only in moderate size, such as those used in microprocessor-based computer systems.

Figure 5.11b shows a four-way mapping example using a fully associative search. The tag is 4 bits long because 16 possible cache blocks can be destined for the same block frame. The major advantage of using full associativity is to allow the implementation of a better block replacement policy with reduced block contention. The major drawback lies in the expensive search process requiring a higher hardware cost.

5.2.3 Set-Associative and Sector Caches

Set-associative caches are the most popular cache designs built into commercial computers. Sector mapping caches offer a design alternative to set-associative caches. These two types of cache design are described below.

Set-Associative Cache This design offers a compromise between the two extreme cache designs based on direct mapping and full associativity. If properly designed, this cache may offer the best performance-cost ratio. Most high-performance computer systems are based on this approach. The idea is illustrated in Fig. 5.12.

In a *k-way associative* cache, the m cache block frames are divided into $v = m/k$ sets, with k blocks per set. Each set is identified by a d-bit *set number*, where $2^d = v$. The cache block tags are now reduced to $s - d$ bits. In practice, the *set size* k, or *associativity*, is chosen as 2, 4, 8, 16, or 64, depending on a tradeoff among block size w, cache size m, and other performance/cost factors.

Fully associative mapping can be visualized as having a single set (i.e., $v = 1$) or an m-way associativity. In a k-way associative search, the *tag* needs to be compared only with the k tags within the identified set, as shown in Fig. 5.12a. Since k is rather small in practice, the k-way associative search is much more economical than the full associativity.

In general, a block B_j can be mapped into any one of the available frames $\overline{B_f}$ in a set S_i defined below. The matched tag identifies the current block which resides in the frame.

$$B_j \rightarrow \overline{B_f} \in S_i \qquad \text{if} \quad j(\text{modulo } v) = i \tag{5.2}$$

Design Tradeoffs The set size (associativity) k and the number of sets v are inversely related by

$$m = v \times k \tag{5.3}$$

For a fixed cache size there exists a tradeoff between the set size and the number of sets.

The advantages of the set-associative cache include the following:

First, the block replacement algorithm needs to consider only a few blocks in the same set. Thus the replacement policy can be more economically implemented with limited choices, as compared with the fully associative cache.

Copyrighted material

Model 85, there are 16 sectors, each having 16 blocks. Each block has 64 bytes, giving a total of 1024 bytes in each sector and a total cache capacity of 16 Kbytes using a LRU block replacement policy.

■

5.2.4 Cache Performance Issues

The performance of a cache design concerns two related aspects: the *cycle count* and the *hit ratio*. The cycle count refers to the number of basic machine cycles needed for cache access, update, and coherence control. The hit ratio determines how effectively the cache can reduce the overall memory-access time. Tradeoffs do exist between these two aspects. Key factors affecting cache speed and hit ratio are discussed below.

Program trace-driven simulation and *analytical modeling* are two complementary approaches to studying cache performance. Both have to be applied together in order to provide a credible performance assessment.

Simulation studies present snapshots of program behavior and cache responses but they suffer from having a microscopic perspective.

Analytical models may deviate from reality under simplification. However, they provide some macroscopic and intuitive insight into the underlying processes.

Agreement between results generated from the two approaches allows one to draw a more credible conclusion. However, the generalization of any conclusion is limited by the finite-sized address traces and by the assumptions about address trace patterns. Simulation results can be used to verify the theoretical results, and analytical formulation can guide simulation experiments on a wider range of parameters.

Cycle Counts The cache speed is affected by the underlying static or dynamic RAM technology, the cache organization, and the cache hit ratios. The total cycle count should be predicated with appropriate cache hit ratios. This will affect various cache design decisions, as already seen in previous sections.

The cycle counts are not credible unless detailed simulation of all aspects of a memory hierarchy is performed. The write-through or write-back policies also affect the cycle count. Cache size, block size, set number, and associativity all affect the cycle count as illustrated in Fig. 5.14.

The cycle count is directly related to the hit ratio, which decreases almost linearly with increasing values of the above cache parameters. But the decreasing trend becomes flat and after a certain point turns into an increasing trend (the dashed line in Fig. 5.14a). This is caused primarily by the effect of the block size on the hit ratio, which will be discussed below.

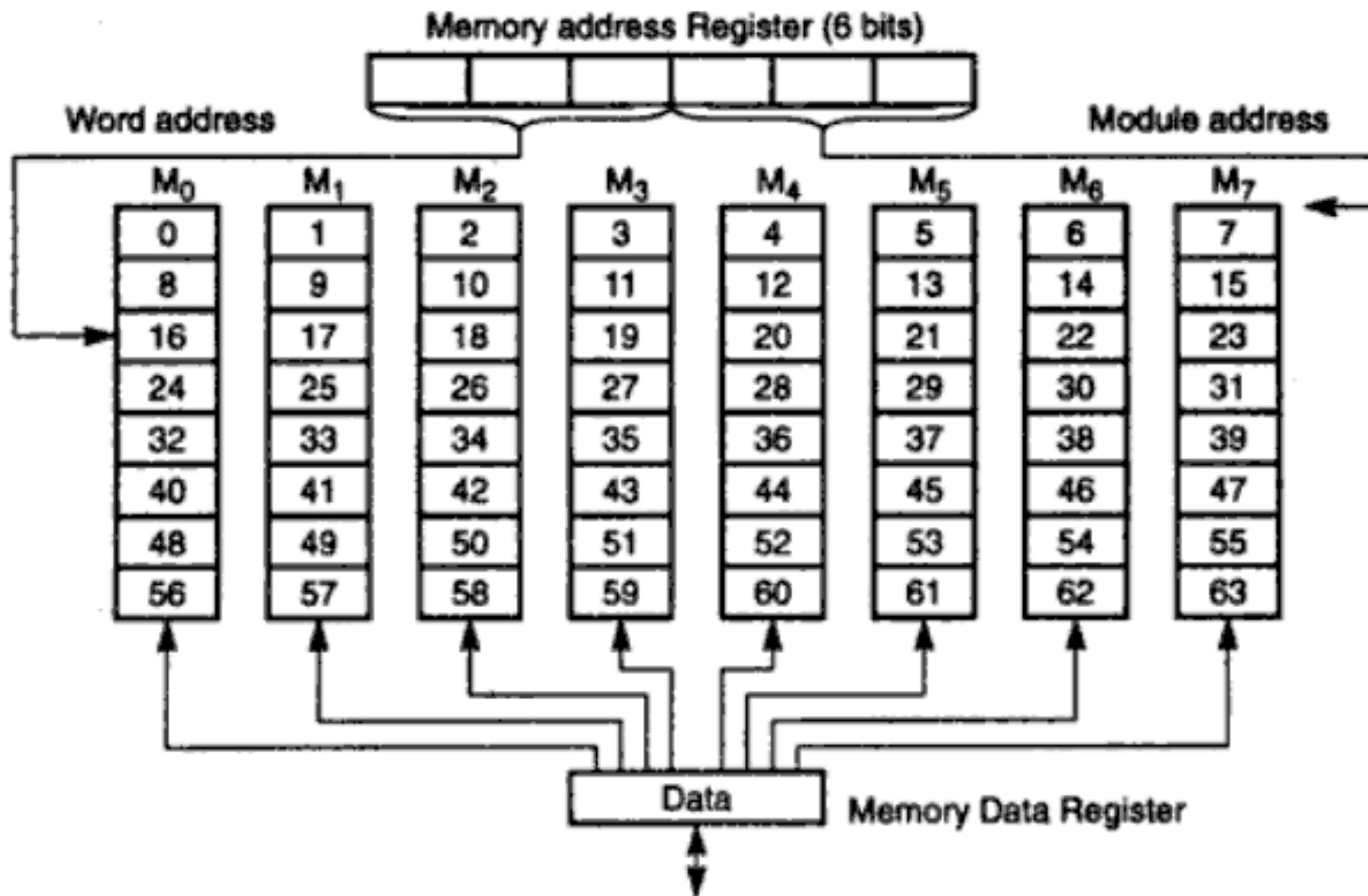
Hit Ratios The cache hit ratio is affected by the cache size and by the block size in different ways. These effects are illustrated in Figs. 5.14b and 5.14c, respectively. Generally, the hit ratio increases with respect to increasing cache size (Fig. 5.14b).

When the cache size approaches infinity, a 100% hit ratio should be expected. However, this will never happen because the cache size is always bounded by a limited budget. The initial cache loading and changes in locality also prevent such an ideal

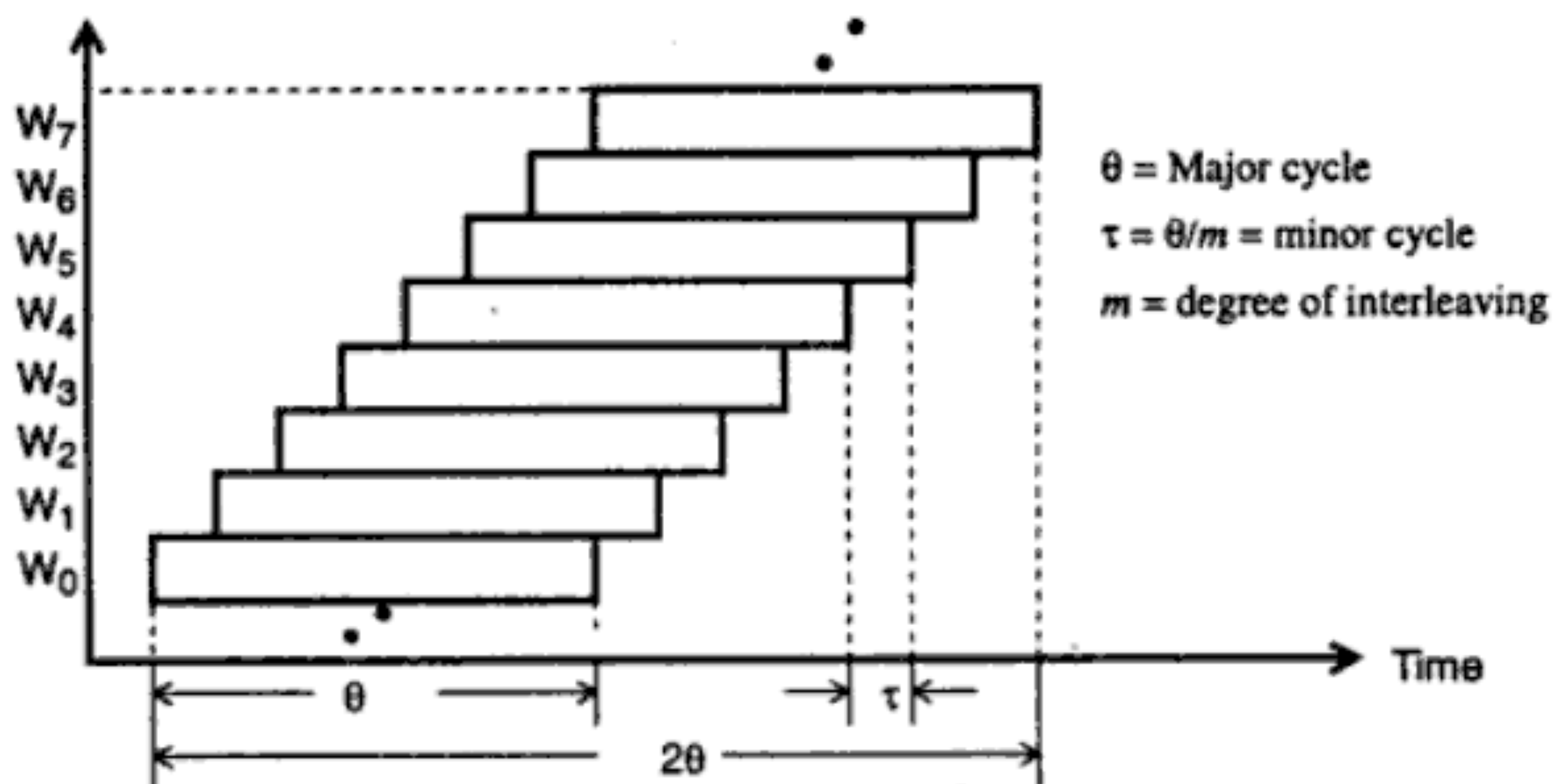
Copyrighted material

successive memory modules separated in every minor cycle τ.

Note that the pipelined access of the block of eight contiguous words is sandwiched between other pipelined block accesses before and after the present block. Even though the total block access time is 2θ, the *effective access time* of each word is reduced to τ as the memory is contiguously accessed in a pipelined fashion.



(a) Eight-way low-order interleaving (absolute address shown in each memory word)



(b) Pipelined access of eight consecutive words in a C-access memory

Figure 5.16 Multiway interleaved memory organization and the C-access timing chart.

Copyrighted material

5.4 Sequential and Weak Consistency Models

This section studies shared-memory behavior in relation to program execution order and memory-access order. The sequential consistency and weak consistency memory models are characterized and their potential for improving performance is assessed. In Chapter 9, we will introduce the processor consistency and release consistency models for building scalable multiprocessor systems.

5.4.1 Atomicity and Event Ordering

The problem of memory inconsistency arises when the memory-access order differs from the program execution order. As illustrated in Fig. 5.19a, a uniprocessor system maps an SISD sequence into a similar execution sequence. Thus memory accesses (for instructions and data) are consistent with the program execution order. This property has been called *sequential consistency* (Lamport, 1979).

In a shared-memory multiprocessor, there are multiple instruction sequences in different processors as shown in Fig. 5.19b. Different ways of interleaving the MIMD instruction sequences into a global memory-access sequence lead to different shared memory behaviors.

How these two sequences are made consistent distinguishes the memory behavior in strong and weak models. The quality of a memory model is indicated by hardware/software efficiency, simplicity, usefulness, and bandwidth performance.

Memory Consistency Issues The behavior of a shared-memory system as observed by processors is called a *memory model*. Specification of the memory model answers three fundamental questions: (1) What behavior should a programmer/compiler expect from a shared-memory multiprocessor? (2) How can a definition of the expected behavior guarantee coverage of all contingencies? (3) How must processors and the memory system behave to ensure consistent adherence to the expected behavior of the multiprocessor?

In general, choosing a memory model involves making a compromise between a strong model minimally restricting software and a weak model offering efficient implementation. The use of *partial order* in specifying memory events gives a formal description of special memory behavior.

Primitive memory operations for multiprocessors include *load* (*read*), *store* (*write*), and one or more synchronization operations such as *swap* (atomic *load-store*) or *conditional store*. For simplicity, we consider one representative synchronization operation *swap*, besides the *load* and *store* operations.

Event Orderings On a multiprocessor, concurrent instruction streams (or threads) executing on different processors are *processes*. Each process executes a code segment. The order in which shared memory operations are performed by one process may be used by other processes. *Memory events* correspond to shared-memory accesses. Consistency models specify the order by which the events from one process should be observed by other processes in the machine.

Copyrighted material

accessed in real time over all *loads* and *stores* with respect to all processor pairs and location pairs.

(3) If two operations appear in a particular program order, then they appear in the same memory order.

(4) The *swap* operation is atomic with respect to other *stores*. No other *store* can intervene between the *load* and *store* parts of a *swap*.

(5) All *stores* and *swaps* must eventually terminate.

Lamport's definition sets the basic spirit of sequential consistency. The memory-access constraints imposed by Dubois et al. are refined from Lamport's definition with respect to atomicity. The conditions on sequential consistency specified by Sindhu et al. are further refined with respect to partial ordering relations. Implementation requirements of these constraints are discussed below.

Implementation Considerations Figure 5.20 shows that the shared memory consists of a single port that is able to service exactly one operation at a time, and a switch that connects this memory to one of the processors for the duration of each memory operation. The order in which the switch is thrown from one processor to another determines the global order of memory-access operations.

The sequential consistency model implies total ordering of *stores/loads* at the instruction level. This should be transparent to all processors. In other words, sequential consistency must hold for any processor in the system.

A conservative multiprocessor designer may prefer the sequential consistency model, in which consistency is enforced by hardware on-the-fly. Memory accesses are atomic and strongly ordered, and confusion can be avoided by having all processors/caches wait sufficiently long for unexpected events.

Strong ordering of all shared-memory accesses in the sequential consistency model preserves the program order in all processors. A sequentially consistent multiprocessor cannot determine whether the system is a multitasking uniprocessor or a multiprocessor. Interprocessor communication can be implemented with simple *loads/stores*, such as Dekker's algorithm for synchronized entry into a critical section by multiple processors. All memory accesses must be globally performed in program order.

A processor cannot issue another access until the most recently shared writable memory access by a processor has been globally performed. This may require the propagation of all shared-memory accesses to all processors, which is rather time-consuming and costly.

Most multiprocessors have implemented the sequential consistency model because of its simplicity. However, the model may lead to rather poor memory performance due to the imposed strong ordering of memory events. This is especially true when the system becomes very large. Thus sequential consistency reduces the scalability of a multiprocessor system.

5.4.3 Weak Consistency Models

The multiprocessor memory model may range anywhere from strong (or sequential) consistency to various degrees of weak consistency. In this section, we describe the *weak*

Copyrighted material

Each processor is rated with 10 MIPS if a 100% cache hit ratio is assumed. On the average, each instruction needs 0.2 memory access. The read access and write access are assumed equally probable.

For a crude approximation, consider only the penalty caused by shared-memory access and ignore all other overheads. The cache is targeted to maintain a hit ratio of 0.95. A cache access on a read-hit takes 20 ns; that on a write-hit takes 60 ns with a write-back scheme, and with a write-through scheme it needs 400 ns.

When a cache block is to be replaced, the probability that it is dirty is estimated as 0.1. An average block transfer time between the cache and shared memory via the bus is 400 ns.

(a) Derive the effective memory-access times per instruction for the write-through and write-back caches separately.
(b) Calculate the effective MIPS rate for each processor. Determine an upper bound on the effective MIPS rate of the 16-processor system. Discuss why the upper bound cannot be achieved by considering the memory penalty alone.

Problem 5.5 Explain the following terms associated with cache and memory architectures.

(a) Low-order memory interleaving.
(b) Physical address cache versus virtual address cache.
(c) Atomic versus nonatomic memory accesses.
(d) Memory bandwidth and fault tolerance.

Problem 5.6 Explain the following terms associated with cache design:

(a) Write-through versus write-back caches.
(b) Cacheable versus noncacheable data.
(c) Private caches versus shared caches.
(d) Cache flushing policies.
(e) Factors affecting cache hit ratios.

Problem 5.7 Consider the simultaneous execution of the three programs on the three processors shown in Fig. 5.19c. Answer the following questions with reasoning or supported by computer simulation results:

(a) List the 90 execution interleaving orders of the six instructions $\{a, b, c, d, e, f\}$ which will preserve the individual program orders. The corresponding output patterns (6-tuples) should be listed accordingly.
(b) Can all 6-tuple combinations be generated out of the 720 non-program-order interleavings? Justify the answer with reasoning and examples.
(c) We have assumed atomic memory access in this example. Explain why the output 011001 is not possible in an atomic memory multiprocessor system if individual program orders are preserved.

Copyrighted material

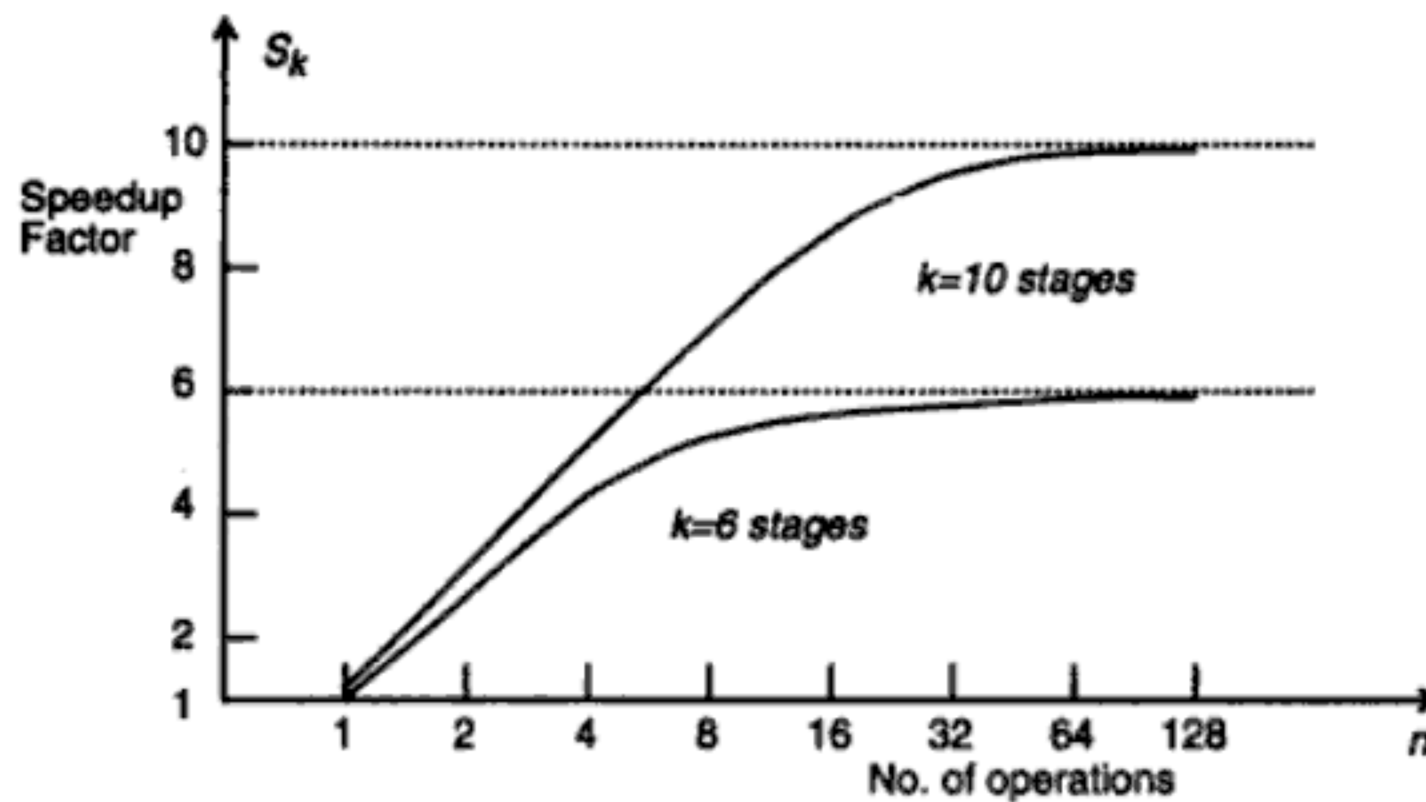
Problem 5.19 Explain the following terms associated with memory management:

(a) The role of a memory manager in an OS kernel.
(b) Preemptive versus nonpreemptive memory allocation policies.
(c) Swapping memory system and examples.
(d) Demand paging memory system and examples.
(e) Hybrid memory system and examples.

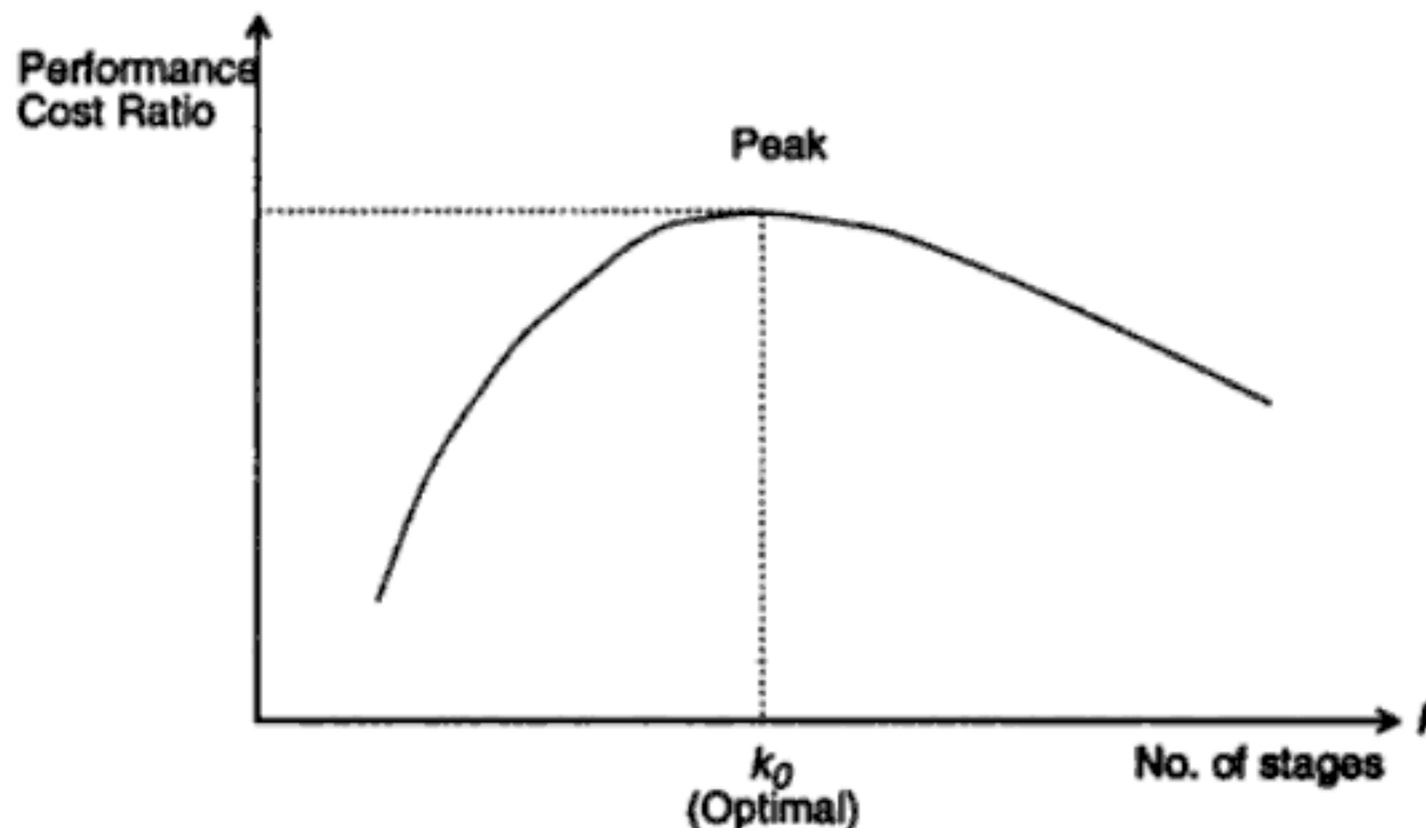
Problem 5.20 Compare the memory-access constraints in the following memory consistency models:

(a) Determine the similarities and subtle differences among the conditions on sequential consistency imposed by Lamport (1979), by Dubois et al. (1986), and by Sindhu et al. (1992), respectively.
(b) Repeat question (a) between the DSB model and the TSO model for weak consistency memory systems.
(c) A PSO (partial store order) model for weak consistency has been refined from the TSO model. Study the PSO specification in the paper by Sindhu et al. (1992) and compare the relative merits between the TSO and the PSO memory models.

Copyrighted material



(a) Speedup factor as a function of the number of operations (Eq. 6.5)



(b) Optimal number of pipeline stages (Eqs. 6.6 and 6.7)

Figure 6.2 Speedup factors and the optimal number of pipeline stages for a linear pipeline unit.

Let t be the total time required for a nonpipelined sequential program of a given function. To execute the same program on a k-stage pipeline with an equal flow-through delay t, one needs a clock period of $p = t/k + d$, where d is the latch delay. Thus, the pipeline has a maximum throughput of $f = 1/p = 1/(t/k+d)$. The total pipeline cost is roughly estimated by $c + kh$, where c covers the cost of all logic stages and h represents the cost of each latch. A pipeline *performance/cost ratio* (PCR) has been defined by Larson (1973):

$$PCR = \frac{f}{c+kh} = \frac{1}{(t/k+d)(c+kh)} \tag{6.6}$$

Figure 6.2b plots the PCR as a function of k. The peak of the PCR curve corre-

Copyrighted material

by the number of latencies along the cycle. The latency cycle (1,8) thus has an average latency of $(1+8)/2 = 4.5$. A *constant cycle* is a latency cycle which contains only one latency value. Cycles (3) and (6) in Figs. 6.5b and 6.5c are both constant cycles. The average latency of a constant cycle is simply the latency itself. In the next section, we describe how to obtain these latency cycles systematically.

6.2.2 Collision-Free Scheduling

When scheduling events in a pipeline, the main objective is to obtain the shortest average latency between initiations without causing collisions. In what follows, we present a systematic method for achieving such collision-free scheduling.

We study below *collision vectors*, *state diagrams*, *single cycles*, *greedy cycles*, and *minimal average latency* (MAL). These pipeline design theory was originally developed by Davidson (1971) and his students.

Collision Vectors By examining the reservation table, one can distinguish the set of permissible latencies from the set of forbidden latencies. For a reservation table with n columns, the *maximum forbidden latency* $m \leq n-1$. The permissible latency p should be as small as possible. The choice is made in the range $1 \leq p \leq m-1$.

A permissible latency of $p = 1$ corresponds to the ideal case. In theory, a latency of 1 can always be achieved in a static pipeline which follows a linear (diagonal or streamlined) reservation table as shown in Fig. 6.1c.

The combined set of permissible and forbidden latencies can be easily displayed by a collision vector, which is an m-bit binary vector $C = (C_m C_{m-1} \cdots C_2 C_1)$. The value of $C_i = 1$ if latency i causes a collision and $C_i = 0$ if latency i is permissible. Note that it is always true that $C_m = 1$, corresponding to the maximum forbidden latency.

For the two reservation tables in Fig. 6.3, the collision vector $C_X = (1011010)$ is obtained for function X, and $C_Y = (1010)$ for function Y. From C_X, we can immediately tell that latencies $7, 5, 4$, and 2 are forbidden and latencies $6, 3$, and 1 are permissible. Similarly, 4 and 2 are forbidden latencies and 3 and 1 are permissible latencies for function Y.

State Diagrams From the above collision vector, one can construct a *state diagram* specifying the permissible state transitions among successive initiations. The collision vector, like C_X above, corresponds to the *initial state* of the pipeline at time 1 and thus is called an *initial collision vector*. Let p be a permissible latency within the range $1 \leq p \leq m-1$.

The *next state* of the pipeline at time $t+p$ is obtained with the assistance of an m-bit right shift register as in Fig. 6.6a. The initial collision vector C is initially loaded into the register. The register is then shifted to the right. Each 1-bit shift corresponds to an increase in the latency by 1. When a 0 bit emerges from the right end after p shifts, it means p is a permissible latency. Likewise, a 1 bit being shifted out means a collision, and thus the corresponding latency should be forbidden.

Logical 0 enters from the left end of the shift register. The next state after p shifts is thus obtained by bitwise-ORing the initial collision vector with the shifted register

Copyrighted material

In total, the operation X_1 has been delayed one cycle from time 4 to time 5 and the operation X_2 has been delayed two cycles from time 5 to time 7. All remaining operations (marked as X in Fig. 6.8b) are unchanged. This new table leads to a new collision vector (100010) and a modified state diagram in Fig. 6.8c.

This diagram displays a greedy cycle $(1,3)$, resulting in a reduced MAL $= (1+3)/2 = 2$. The delay insertion thus improves the pipeline performance, yielding a lower bound for the MAL.

■

Pipeline Throughput This is essentially the initiation rate or the average number of task initiations per clock cycle. If N tasks are initiated within n pipeline cycles, then the *initiation rate* or *pipeline throughput* is measured as N/n. This rate is determined primarily by the inverse of the MAL adapted. Therefore, the scheduling strategy does affect the pipeline performance.

In general, the shorter the adapted MAL, the higher the throughput that can be expected. The highest achievable throughput is one task initiation per cycle, when the MAL equals 1 since $1 \leq \text{MAL} \leq$ the shortest latency of any greedy cycle. Unless the MAL is reduced to 1, the pipeline throughput becomes a fraction.

Pipeline Efficiency Another important measure is *pipeline efficiency.* The percentage of time that each pipeline stage is used over a sufficiently long series of task initiations is the *stage utilization.* The accumulated rate of all stage utilizations determines the pipeline efficiency.

Let us reexamine latency cycle (3) in Fig. 6.5b. Within each latency cycle of three clock cycles, there are two pipeline stages, S_1 and S_3, which are completely and continuously utilized after time 6. The pipeline stage S_2 is used for two cycles and is idle for one cycle.

Therefore, the entire pipeline can be considered $8/9 = 88.8\%$ efficient for latency cycle (3). On the other hand, the pipeline is only $14/27 = 51.8\%$ efficient for latency cycle (1, 8) and $8/16 = 50\%$ efficient for latency cycle (6), as illustrated in Figs. 6.5a and 6.5c, respectively. Note that none of the three stages is fully utilized with respect to two initiation cycles.

The pipeline throughput and pipeline efficiency are related to each other. Higher throughput results from a shorter latency cycle. Higher efficiency implies less idle time for pipeline stages. The above example demonstrates that higher throughput also accompanies higher efficiency. Other examples may show a contrary conclusion. The relationship between the two measures is a function of the reservation table and of the initiation cycle adopted.

At least one stage of the pipeline should be fully (100%) utilized at the steady state in any acceptable initiation cycle; otherwise, the pipeline capability has not been fully explored. In such cases, the initiation cycle may not be optimal and another initiation cycle should be examined for improvement.

Copyrighted material

it contains instructions sequentially ahead of the current instruction. This saves the instruction fetch time from memory. Second, it recognizes when the target of a branch falls within the loop boundary. In this case, unnecessary memory accesses can be avoided if the target instruction is already in the loop buffer. The CDC 6600 and Cray 1 have used loop buffers.

Multiple Functional Units Sometimes a certain pipeline stage becomes the bottleneck. This stage corresponds to the row with the maximum number of checkmarks in the reservation table. This bottleneck problem can be alleviated by using multiple copies of the same stage simultaneously. This leads to the use of multiple execution units in a pipelined processor design (Fig. 6.12).

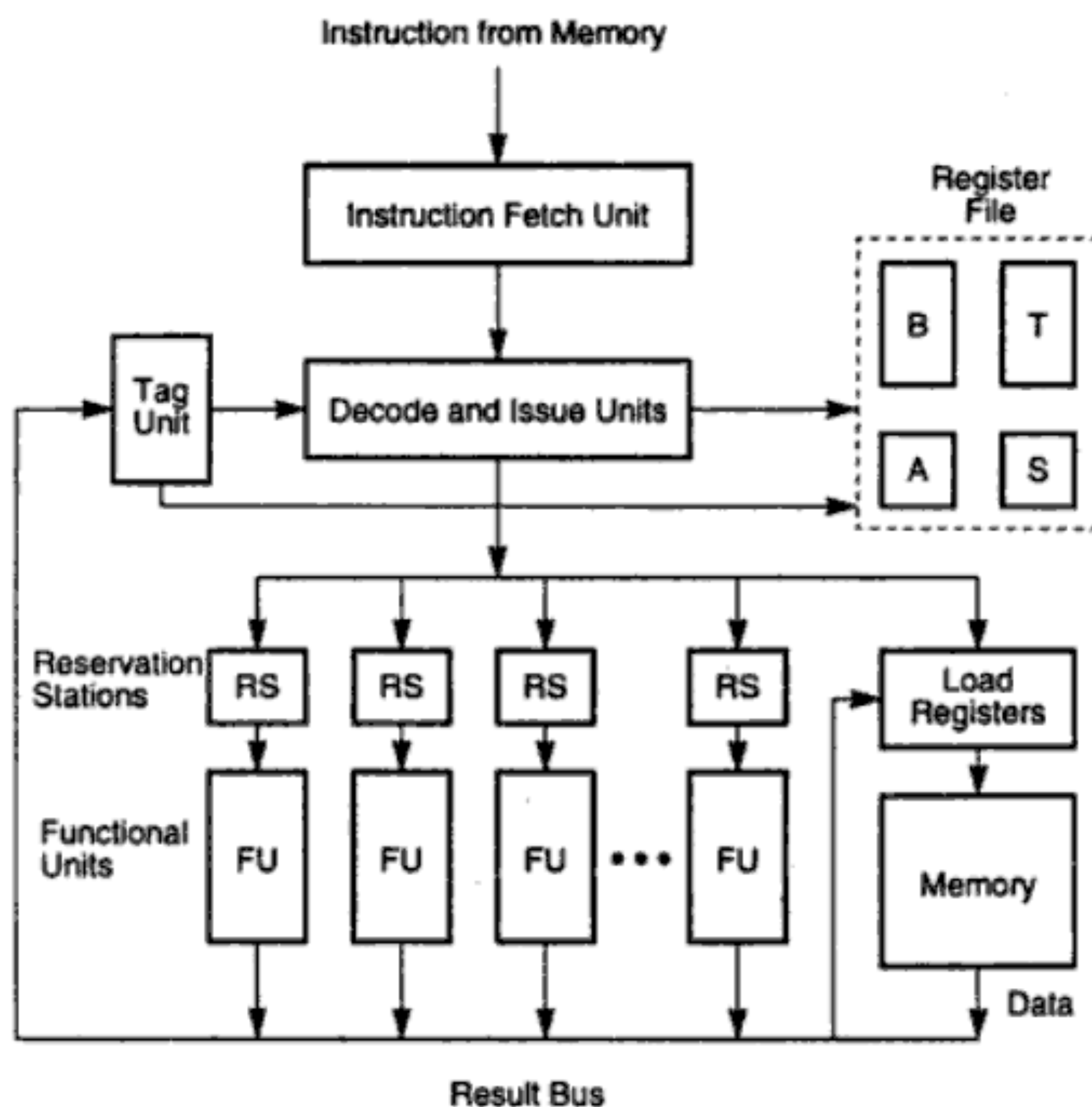


Figure 6.12 A pipelined processor with multiple functional units and distributed reservation stations supported by tagging. (Courtesy of G. Sohi; reprinted with permission from *IEEE Transactions on Computers*, March 1990)

Sohi (1990) used a model architecture for a pipelined scalar processor containing multiple functional units (Fig. 6.12). In order to resolve data or resources dependences among the successive instructions entering the pipeline, the *reservation stations* (RS) are used with each functional unit. Operands can wait in the RS until its data dependences have been resolved. Each RS is uniquely identified by a *tag*, which is monitored by a *tag unit*.

Copyrighted material

Most of today's pipelined computers use some form of static scheduling by the compiler. The compiler-based software interlocking is cheaper to implement and flexible to apply. However, in high-performance computers, we need special hardware support to achieve dynamic instruction scheduling as explained below.

Tomasulo's Algorithm This hardware dependence-resolution scheme was first implemented with multiple floating-point units of the IBM 360/91 processor. The hardware platform was abstracted in Fig. 6.12. For the Model 91 processor, three RSs are used in a floating-point adder and two pairs in a floating-point multiplier. The scheme resolves resource conflicts as well as data dependences using register tagging to allocate or deallocate the source and destination registers.

An issued instruction whose operands are not available is forwarded to an RS associated with the functional unit it will use. It waits until its data dependences have been resolved and its operands become available. The dependence is resolved by monitoring the result bus (called *common data bus* in Model 91). When all operands for an instruction are available, it is dispatched to the functional unit for execution. All working registers are tagged. If a source register is busy when an instruction reaches the issue stage, the tag for the source register is forwarded to an RS. When the register becomes available, the tag can signal the availability.

Example 6.6 Tomasulo's algorithm for dynamic instruction scheduling

Tomasulo's algorithm was applied to work with processors having a few floating-point registers. In the case of Model 91, only four registers were available. Figure 6.16a shows a minimum-register machine code for computing $X = Y + Z$ and $A = B \times C$. The pipeline timing with Tomasulo's algorithm appears in Fig. 6.16b. Here, the total execution time is 13 cycles, counting from cycle 4 to cycle 15 by ignoring the pipeline startup and draining times.

Memory is treated as a special functional unit. When an instruction has completed execution, the result (along with its tag) appears on the result bus. The registers as well as the RSs monitor the result bus and update their contents (and ready/busy bits) when a matching tag is found. Details of the algorithm can be found in the original paper by Tomasulo (1967).

■

CDC Scoreboarding The CDC 6600 was an early high-performance computer that used dynamic instruction scheduling hardware. Figure 6.17a shows a CDC 6600-like processor, in which multiple functional units appeared as multiple execution pipelines. Parallel units allow instructions to complete out of the original program order. The processor had instruction buffers for each execution unit. Instructions are issued to available functional units regardless of whether register input data were available.

The instruction's control information would then wait in a buffer for its data to be produced by other instructions. To control the correct routing of data between execution units and registers, the CDC 6600 used a centralized control units known as

Copyrighted material

to use the entire history of the branch to predict the future choice. This is infeasible to implement. Therefore, most dynamic prediction is determined with limited recent history, as illustrated in Fig. 6.19.

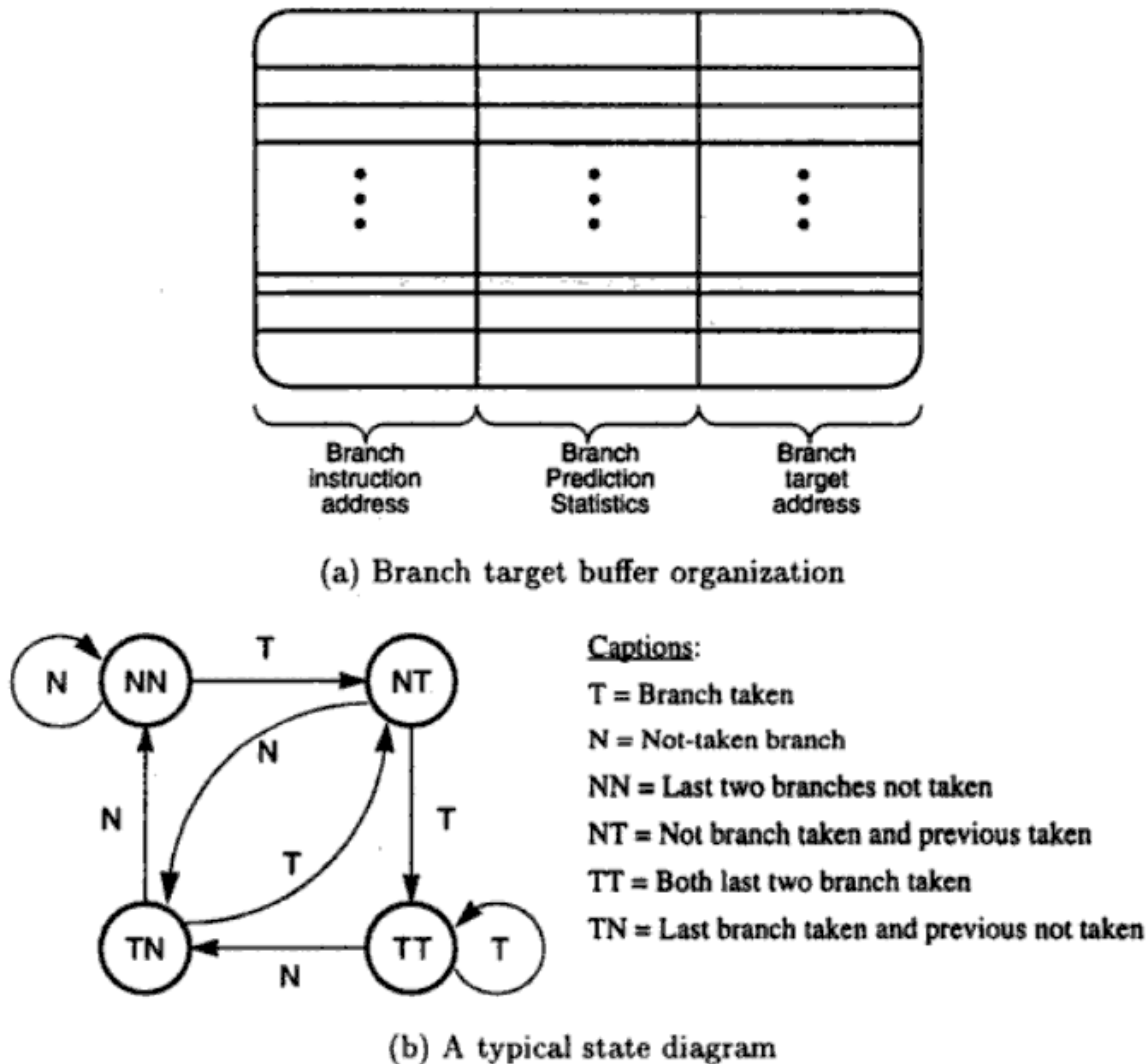


Figure 6.19 Branch history buffer and a state transition diagram used in dynamic branch prediction. (Courtesy of Lee and Smith, *IEEE Computer*, 1984)

Cragon (1992) has classified dynamic branch strategies into three major classes: One class predicts the branch direction based upon information found at the decode stage. The second class uses a cache to store target addresses at the stage the effective address of the branch target is computed. The third scheme uses a cache to store target instructions at the fetch stage. All dynamic predictions are adjusted dynamically as a program is executed.

Dynamic prediction demands additional hardware to keep track of the past behavior of the branch instructions at run time. The amount of history recorded should be small. Otherwise, the prediction logic becomes too costly to implement.

Lee and Smith (1984) have shown the use of a *branch target buffer* (BTB) to

Copyrighted material

$$X = (-1)^s \times 2^{e-1023} \times (1.m) \tag{6.16}$$

Special rules are given in the standard to handle overflow or underflow conditions. Interested readers may check the published IEEE standards for details.

■

Floating-Point Operations The four primitive arithmetic operations are defined below for a pair of floating-point numbers represented by $X = (m_x, e_x)$ and $Y = (m_y, e_y)$. For clarity, we assume $e_x \leq e_y$ and base $r = 2$.

$$X + Y = (m_x \times 2^{e_x - e_y} + m_y) \times x^{e_y} \tag{6.17}$$

$$X - Y = (m_x \times 2^{e_x - e_y} - m_y) \times x^{e_y} \tag{6.18}$$

$$X \times Y = (m_x \times m_y) \times 2^{e_x + e_y} \tag{6.19}$$

$$X \div Y = (m_x \div m_y) \times 2^{e_x - e_y} \tag{6.20}$$

The above equations clearly identify the number of arithmetic operations involved in each floating-point function. These operations can be divided into two halves: One half is for exponent operations such as comparing their relative magnitudes or adding/subtracting them; the other half is for mantissa operations, including four types of fixed-point operations.

Floating-point units are ideal for pipelined implementation. The two halves of the operations demand almost twice as much hardware as that required in a fixed-point unit. Arithmetic shifting operations are needed for equalizing the two exponents before their mantissas can be added or subtracted.

Shifting a binary fraction m to the right k places corresponds to the weighting $m \times 2^{-k}$, and shifting k places to the left corresponds to $m \times 2^k$. In addition, normalization of a floating-point number also requires left shifts to be performed.

Elementary Functions Elementary functions include trigonometric, exponential, logarithmic, and other transcendental functions. Truncated polynomials or power series can be used to evaluate the elementary functions, such as $\sin x$, $\ln x$, e^x, $\cosh x$, $\tan^{-1} y, \sqrt{x}, x^3$, etc.

Some CORDIC (coordinate rotation digital computer) algorithms have been developed to calculate trigonometric functions and to convert between binary and mixed radix number systems. Interested readers may refer to the book by Hwang (1979) for details of computer arithmetic functions and their hardware implementation.

It should be noted that computer arithmetic can be implemented by hardwired random logic circuitry as well as by table lookup using ROMs or RAMs in memory. Frequently used constants and special function values can be easily generated by table lookup. Hashing can provide fast access to these tables.

6.4.2 Static Arithmetic Pipelines

Most of today's arithmetic pipelines are designed to perform fixed functions. These *arithmetic/logic units* (ALUs) perform fixed-point and floating-point operations sepa-

Copyrighted material

This arithmetic pipeline has three stages. The mantissa section and exponent section are essentially two separate pipelines. The mantissa section can perform floating-point add or multiply operations, either single-precision (32 bits) or double-precision (64 bits).

In the mantissa section, stage 1 receives input operands and returns with computation results; 64-bit registers are used in this stage. Note that all three stages are connected to two 64-bit data buses. Stage 2 contains the array multiplier (64×8) which must be repeatedly used to carry out a long multiplication of the two mantissas.

The 67-bit adder performs the addition/subtraction of two mantissas, the barrel shifter is used for normalization. Stage 3 contains registers for holding results before they are loaded into the register file in stage 1 for subsequent use by other instructions.

On the exponent side, a 16-bit bus is used between stages. Stage 1 has an exponent adder for comparing the relative magnitude of two exponents. The result of stage 1 is used to equalize the exponents before mantissa addition can be performed. Therefore, a shift count (from the output of the exponent adder) is sent to the barrel shifter for mantissa alignment.

After normalization of the final result (getting rid of leading zeros), the exponent needs to be readjusted in stage 3 using another adder. The final value of the resulting exponent is fed from the register in stage 3 to the register file in stage 1, ready for subsequent usage.

■

Convergence Division Division can be carried out by repeated multiplications. Mantissa division is carried out by a *convergence method*. This convergence division obtains the quotient $Q = M/D$ of two normalized fractions $0.5 \leq M < D < 1$ in two's complement notation by performing two sequences of chain multiplications as follows:

$$Q = \frac{M \times R_1 \times R_2 \times \cdots \times R_k}{D \times R_1 \times R_2 \times \cdots \times R_k} \tag{6.22}$$

where the successive multipliers

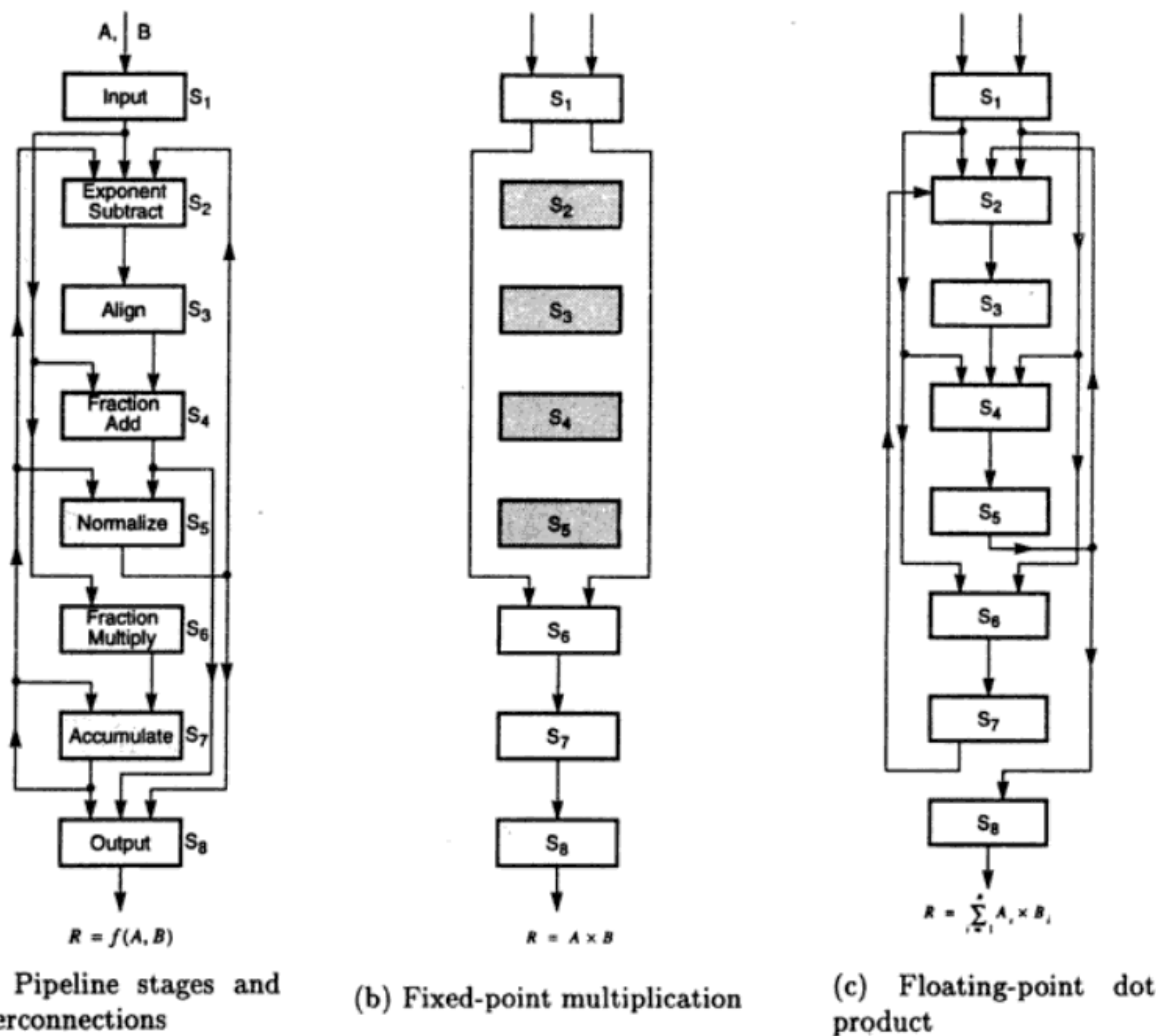
$$R_i = 1 + \delta^{2^{i-1}} = 2 - D^{(i)} \qquad \text{for } i = 1, 2, \ldots, k \quad \text{and} \quad D = 1 - \delta$$

The purpose is to choose R_i such that the denominator $D^{(k)} = D \times R_1 \times R_2 \times \cdots \times R_k \to 1$ for a sufficient number of k iterations, and then the resulting numerator $M \times R_1 \times R_2 \times \cdots \times R_k \to Q$.

Note that the multiplier R_i can be obtained by finding the two's complement of the previous chain product $D^{(i)} = D \times R_1 \times \cdots \times R_{i-1} = 1 - \delta^{2^{i-1}}$ because $2 - D^{(i)} = R_i$. The reason why $D^{(k)} \to 1$ for large k is that

$$\begin{aligned} D^{(i)} &= (1-\delta)(1+\delta)(1+\delta^2)(1+\delta^4)\cdots(1+\delta^{2^{i-1}}) \\ &= (1-\delta^2)(1+\delta^2)(1+\delta^4)\cdots(1+\delta^{2^{i-1}}) \\ &= (1-\delta^{2^i}) \qquad \text{for } i = 1, 2, \cdots, k \end{aligned} \tag{6.23}$$

Copyrighted material



(a) Pipeline stages and interconnections

(b) Fixed-point multiplication

(c) Floating-point dot product

Figure 6.27 The multiplication arithmetic pipeline of the TI Advanced Scientific Computer and the interstage connections of two representative functions. (Shaded stages are unutilized).

implementation technology.

Pipeline Design Parameters Some parameters used in designing the scalar base machine and superscalar machines are summarized in Table 6.1 for four types of pipeline processors to be studied below. All pipelines discussed are assumed to have k stages.

The machine *pipeline cycle* for the scalar base machine is assumed to be 1 time unit, called the *base cycle*. We defined the instruction *issue rate*, *issue latency*, and *simple operation latency* in Section 4.1.1. The *instruction-level parallelism* (ILP) is the maximum number of instructions that can be simultaneously executed in the pipeline.

For the base machine, all of these parameters have a value of 1. All machine types are designed relative to the base machine. The ILP is needed to fully utilize a given pipeline machine.

Copyrighted material

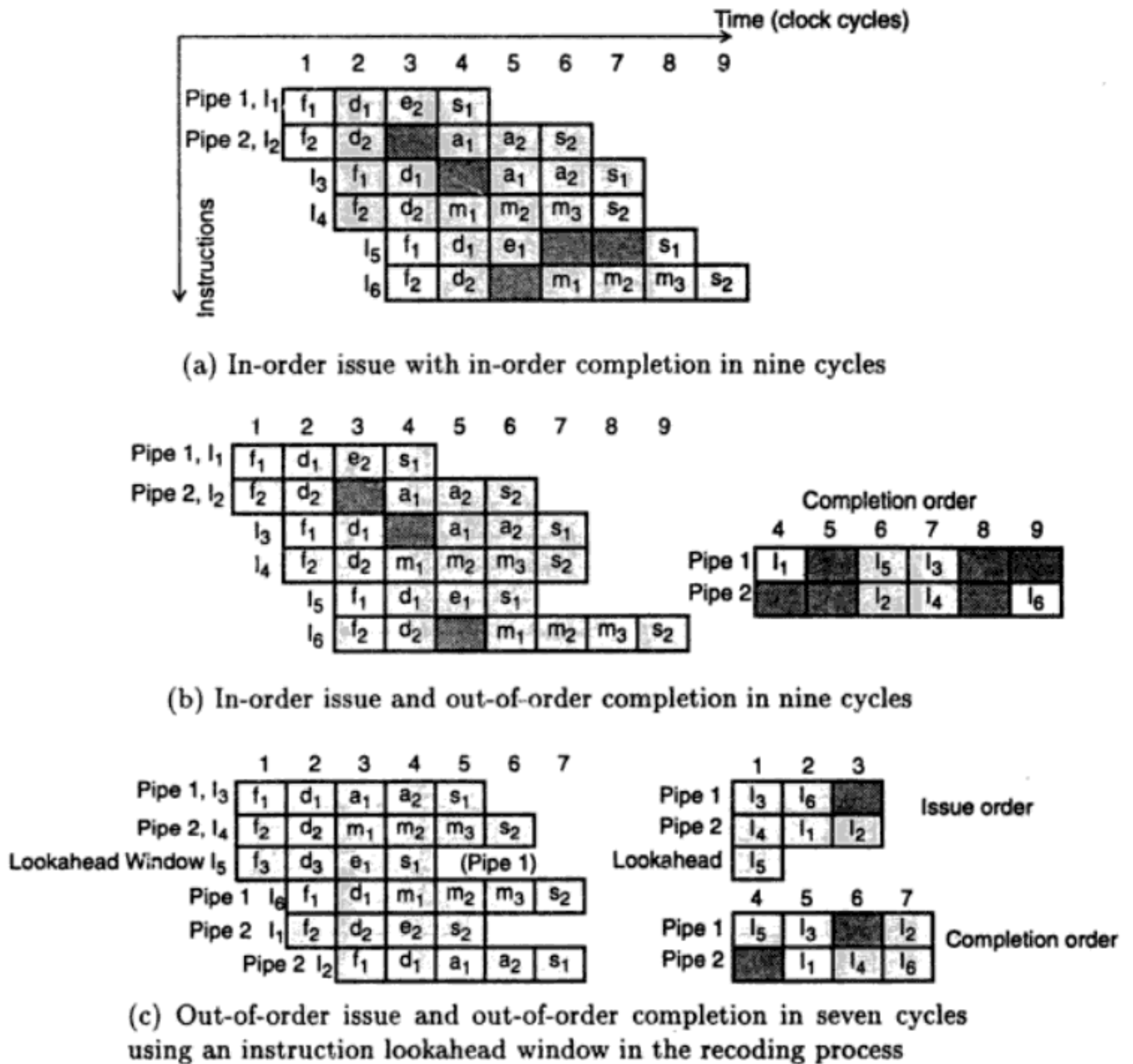


Figure 6.30 Instruction issue and completion policies for a superscalar processor with and without instruction lookahead support. (Timing charts correspond to parallel execution of the program in Fig. 6.28)

instructions I2, I4, and I6 in three consecutive cycles. Due to I1 → I2, I2 has to wait one cycle to use the data loaded in by I1.

I3 is delayed one cycle for the same adder used by I2. I6 has to wait for the result of I5 before it can enter the multiplier stages. In order to maintain in-order completion, I5 is forced to wait for two cycles to come out of pipeline 1. In total, nine cycles are needed and five idle cycles (shaded boxes) are observed.

In Fig. 6.30b, out-of-order completion is allowed even if in-order issue is practiced. The only difference between this out-of-order schedule and the in-order schedule is that I5 is allowed to complete ahead of I3 and I4, which are totally independent of I5. The total execution time does not improve. However, the pipeline utilization rate does.

Only three idle cycles were observed. Note that in Figs. 6.29a and 6.29b, we did

Copyrighted material

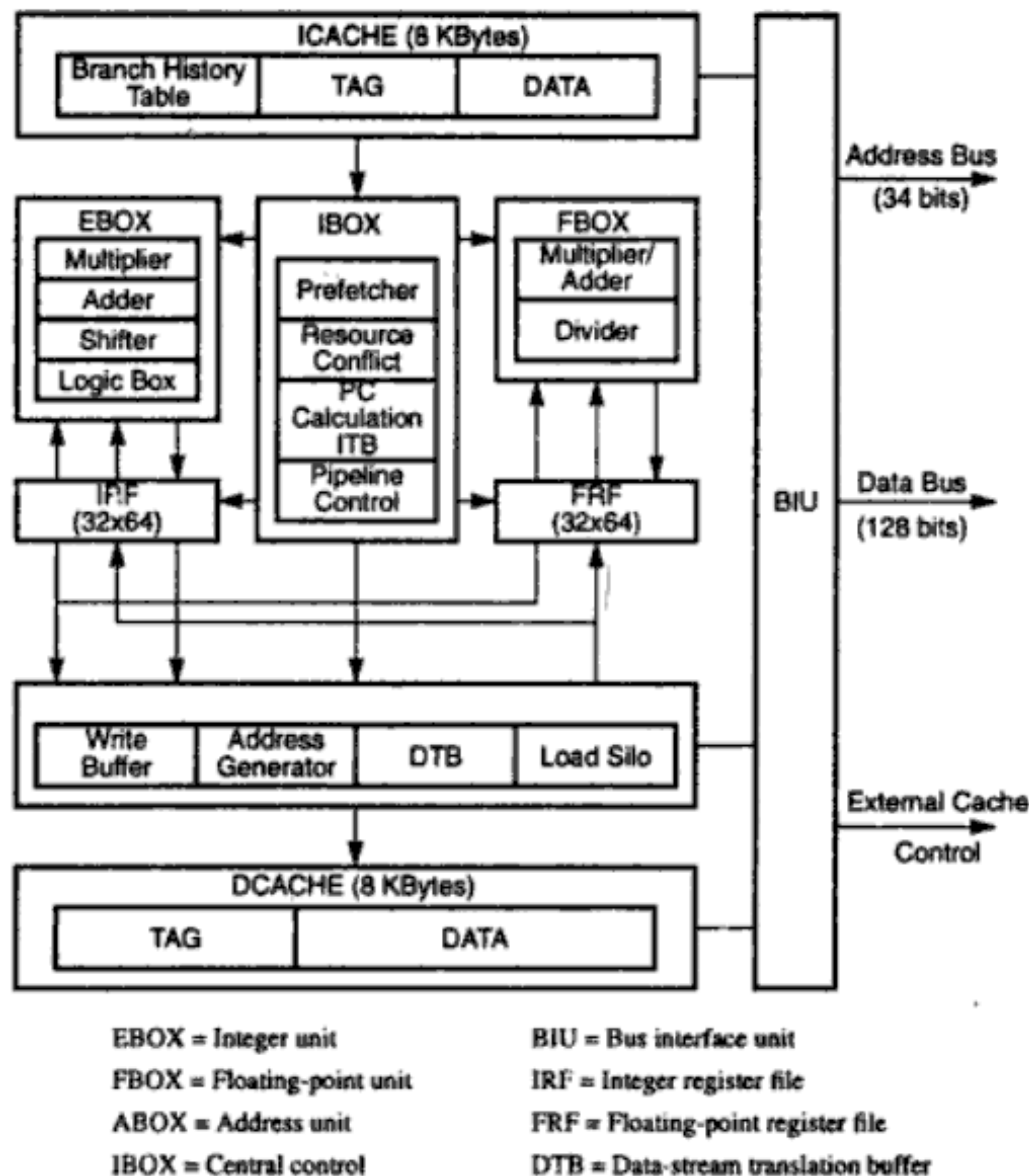


Figure 6.32 The architecture of the DEC 21064-A microprocessor. (Courtesy of Digital Equipment Corporation, 1992)

Unlike others, the Alpha architecture has thirty-two 64-bit integer registers and thirty-two 64-bit floating-point registers. The integer pipeline has 7 stages, and the floating-point pipeline has 10 stages in the initial Alpha design. All Alpha instructions are 32 bits. The instructions interact with each other only by one instruction writing into a register or memory and another one reading from the same place.

The first Alpha implementation issues two instructions per cycle. It is expected to increase the number of issues in later implementations. Pipeline timing hazards, load delay slots, and branch delay slots are all minimized by hardware support. The Alpha is designed to support fast multiprocessor interlocking and interrupts.

A privileged library of software has been developed to run full VMS and to run OSF/1 using different versions of the software library that mirror many of the VAX/OS and MIPS/OS features, respectively. This library makes Alpha an attractive architecture for multiple operating systems. The processor is designed to have a 300-MIPS peak and a 150-Mflops peak at 150 MHz.

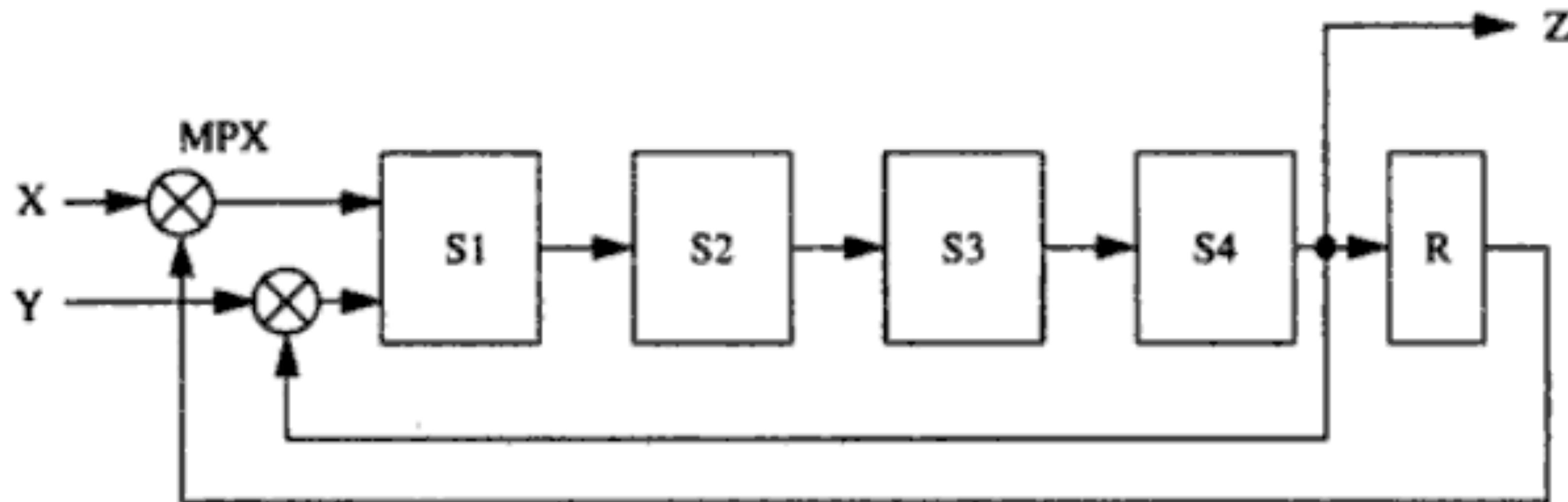
■

Copyrighted material

purpose is to yield a new reservation table leading to an optimal latency equal to the lower bound.

(a) Show the modified reservation table with five rows and seven columns.
(b) Draw the new state transition diagram for obtaining the optimal cycle.
(c) List all the simple cycles and greedy cycles from the state diagram.
(d) Prove that the new MAL equals the lower bound.
(e) What is the optimal throughput of this pipeline? Indicate the percentage of throughput improvement compared with that obtained in part (d) of Problem 6.6.

Problem 6.8 Consider an adder pipeline with four stages as shown below. The pipeline consists of input lines X and Y and output line Z. The pipeline has a register R at its output where the temporary result can be stored and fed back to S1 at a later point in time. The inputs X and Y are multiplexed with the outputs R and Z.



(a) Assume the elements of the vector A are fed into the pipeline through input X, one element per cycle. What is the minimum number of clock cycles required to compute the sum of an N-element vector A: $s = \sum_{I=1}^{N} A(I)$? In the absence of an operand, a value of 0 is input into the pipeline by default. Neglect the setup time for the pipeline.
(b) Let τ be the clock period of the pipelined adder. Consider an equivalent non-pipelined adder with a flow-through delay of 4τ. Find the actual speedup $S_4(64)$ and the efficiency $\eta_4(64)$ of using the above pipeline adder for $N = 64$.
(c) Find the maximum speedup $S_4(\infty)$ and the efficiency $\eta_4(\infty)$ when N tends to infinity.
(d) Find $N_{1/2}$, the minimum vector length required to achieve half of the maximum speedup.

Problem 6.9 Consider the following pipeline reservation table.

	1	2	3	4
S1	X			X
S2		X		
S3			X	

Copyrighted material

Part III

Parallel and Scalable Architectures

Chapter 7
Multiprocessors and Multicomputers

Chapter 8
Multivector and SIMD Computers

Chapter 9
Scalable, Multithreaded, and Dataflow Architectures

Summary

Part III consists of three chapters dealing with parallel, vector, and scalable architectures for building high-performance computers. The multiprocessor system interconnects studied include crossbar switches, multistage networks, hierarchical buses, and multidimensional ring, mesh, and torus architectures. Three generations of multicomputer developments are reviewed. Then we consider message-passing mechanisms.

Vector supercomputers appear either as pipelined multiprocessors or as SIMD data-parallel computers. We study the architectures of the Cray Y-MP, C-90, Cray/MPP, NEC SX, Fujitsu VP-2000, VPP500, VAX 9000, Hitachi S-820, Stardent 3000, CM-2, MasPar MP-1, and CM5 for concurrent scalar/vector processing.

Chapter 9 introduces scalable architectures for massively parallel processing applications. These include both von Neumann, fine-grain, multithreaded, and dataflow architectures. Various latency-hiding techniques are described, including the principles of multithreading. Case studies include the Intel Paragon, Stanford Dash, MIT Alewife, J-Machine and *T, Tera computer, KSR-1, Wisconsin Multicube, USC/OMP, ETL EM4, etc.

Copyrighted material

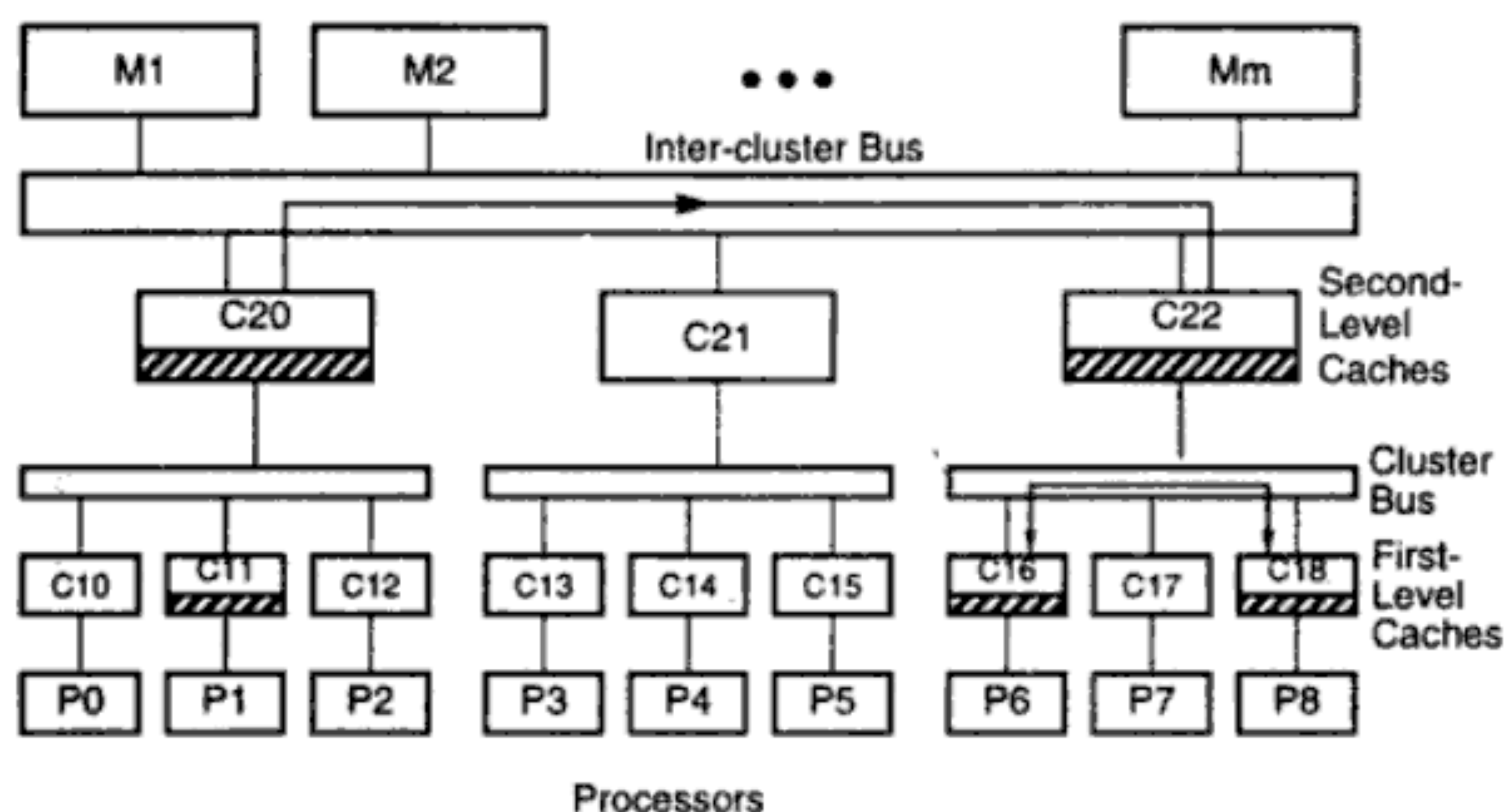


Figure 7.3 A hierarchical cache/bus architecture for designing a scalable multiprocessor. (Courtesy of Wilson; reprinted from *Proc. of Annual Int. Symp. on Computer Architecture*, 1987)

The upper-level caches form another level of shared memory between each cluster and the main memory modules connected to the intercluster bus. Most memory requests should be satisfied at the lower-level caches. Intercluster cache coherence is controlled among the second-level caches and the resulting effects are passed to the lower level.

Example 7.1 Encore's Ultramax multiprocessor architecture

The Ultramax has a two-level hierarchical-bus architecture as depicted in Fig. 7.4. The Ultramax architecture is very similar to that characterized by Wilson, except that the global Nanobus is used only for intercluster communications.

The shared memories are distributed to all clusters instead of being connected to the intercluster bus. The cluster caches form the second-level caches and perform the same filtering and cache coherence control for remote accesses as in Wilson's scheme. After an access request reaches the top bus, it is routed down to the cluster memory that matches it with the reference address.

■

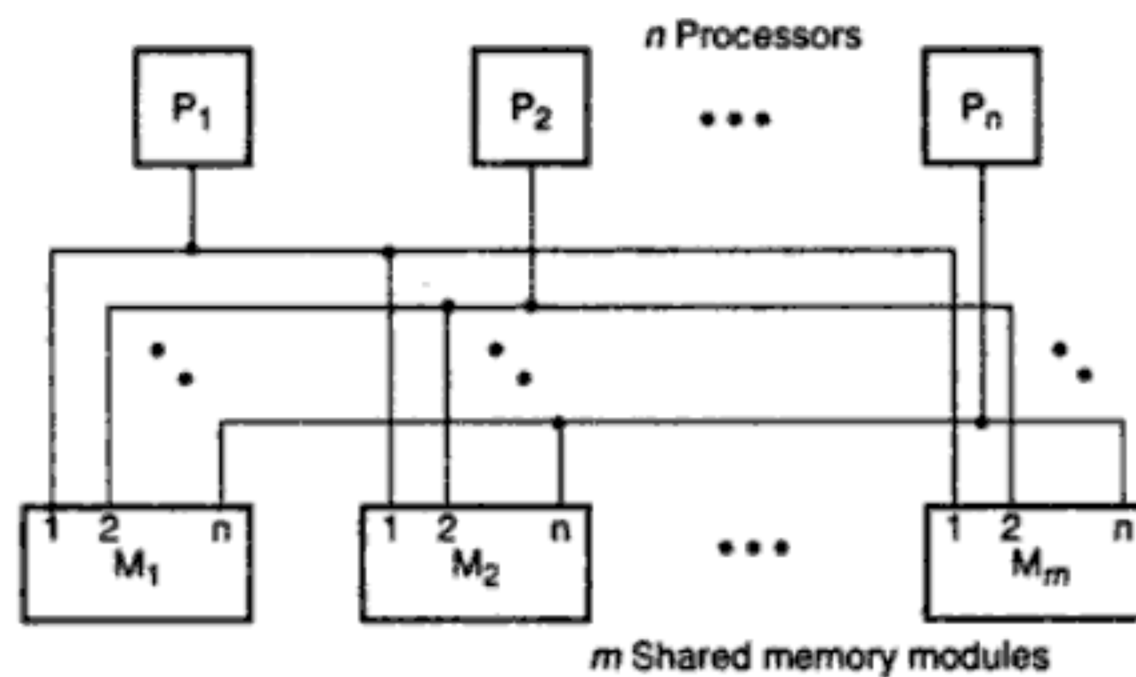
The idea of using *bridges* between multiprocessor clusters is to allow transactions initiated on a local bus to be completed on a remote bus. As exemplified in Fig. 7.5, multiple Futurebuses+ are used to build a very large system consisting of three multiprocessor clusters.

Bridges are used to interface these clusters. The main functions of a bridge include communication protocol conversion, interrupt handling in split transactions, and serving as cache and memory agents. Besides bridging between multiprocessor clusters built around Futurebus+, the standard also provides bridge specifications for the VME bus (IEEE P1014.2) and Multibus II (IEEE P1296.2). These will allow Futurebus+ to be compatible with other bus architectures.

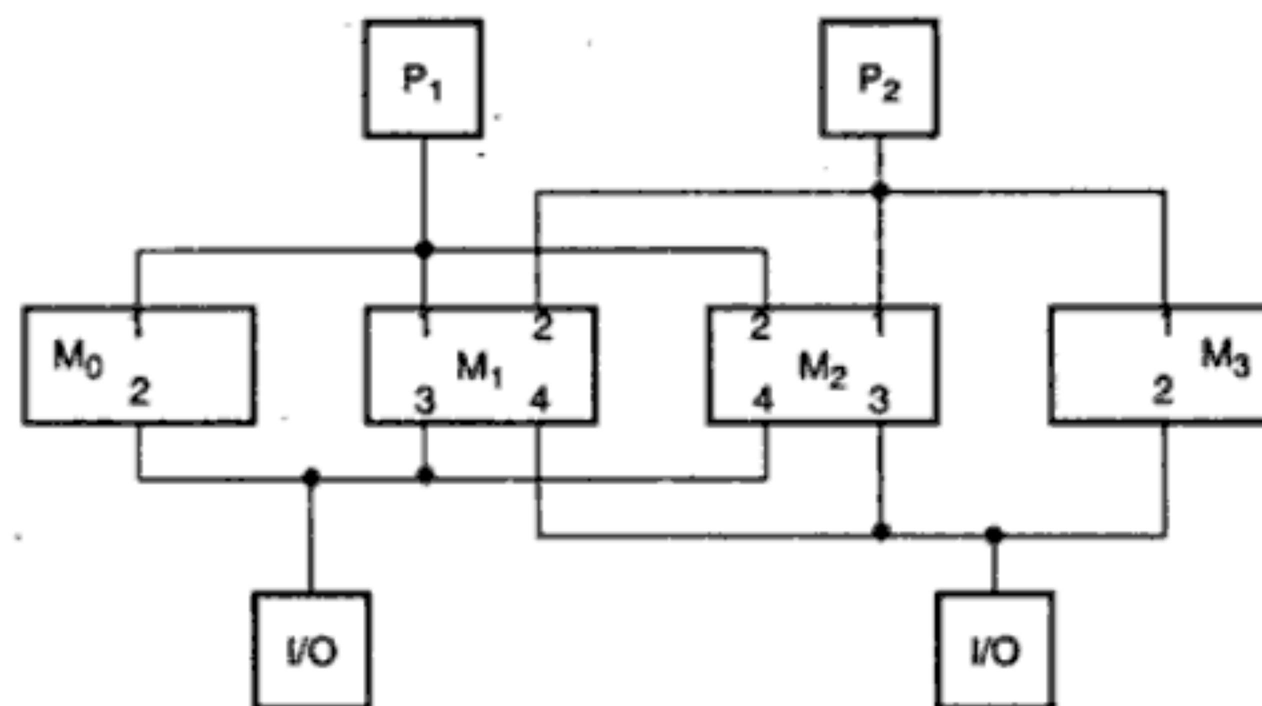
Copyrighted material

Multiport Memory Because building a crossbar network into a large system is cost-prohibitive, many mainframe multiprocessors use a multiport memory organization. The idea is to move all crosspoint arbitration and switching functions associated with each memory module into the memory controller.

Thus the memory module becomes more expensive due to the added access ports and associated logic as demonstrated in Fig. 7.7a. The circles in the diagram represent n switches tied to n input ports of a memory module. Only one of n processor requests will be honored at a time.



(a) n-port memory modules used



(b) Memory ports prioritized or privileged in each module by numbers

Figure 7.7 Multiport memory organizations for multiprocessor systems. (Courtesy of P. H. Enslow, *ACM Computing Surveys*, March 1977)

The multiport memory organization is a compromise solution between a low-cost, low-performance bus system and a high-cost, high-bandwidth crossbar system. The contention bus is time-shared by all processors and device modules attached. The multiport memory must resolve conflicts among processors.

This memory structure becomes expensive when m and n become very large. A

Copyrighted material

In total, sixteen 8×8 crossbar switches are used in Fig. 7.10a and $16 \times 8 + 8 \times 8 = 192$ are used in Fig. 7.10b. Larger Butterfly networks can be modularly constructed using more stages. Note that no broadcast connections are allowed in a Butterfly network, making these networks a restricted subclass of Omega networks.

The Hot-Spot Problem When the network traffic is nonuniform, a *hot spot* may appear corresponding to a certain memory module being excessively accessed by many processors at the same time. For example, a semaphore variable being used as a synchronization barrier may become a hot spot since it is shared by many processors.

Hot spots may degrade the network performance significantly. In the NYU Ultracomputer and the IBM RP3 multiprocessor, a combining mechanism has been added to the Omega network. The purpose was to combine multiple requests heading for the same destination at switch points where conflicts are taking place.

An atomic read-modify-write primitive Fetch&Add(x, e), has been developed to perform parallel memory updates using the combining network.

Fectch&Add This atomic memory operation is effective in implementing an N-way synchronization with a complexity independent of N. In a Fetch&Add(x, e) operation, x is an integer variable in shared memory and e is an integer increment. When a single processor executes this operation, the semantics is

$$\begin{aligned} &\text{Fetch\&Add}(x, e) \\ &\qquad \{\ temp \leftarrow x; \\ &\qquad\quad x \leftarrow temp + e; \\ &\qquad \text{return } temp\} \end{aligned} \tag{7.1}$$

When N processes attempt to Fetch&Add(x, e) the same memory word simultaneously, the memory is updated only once following a *serialization principle*. The sum of the N increments, $e_1 + e_2 + \cdots + e_N$, is produced in any arbitrary serialization of the N requests.

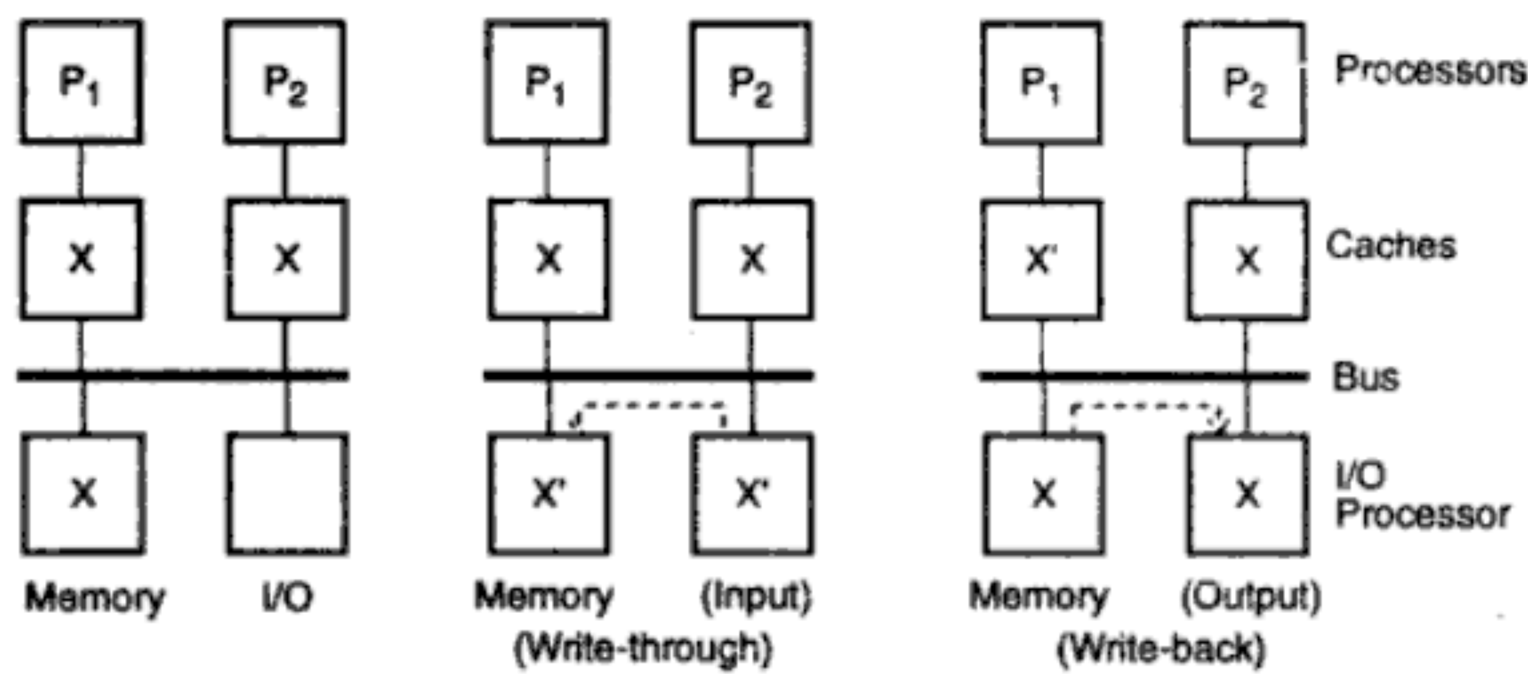
This sum is added to the memory word x, resulting in a new value $x + e_1 + e_2 + \cdots + e_N$. The values returned to the N requests are all unique, depending on the serialization order followed. The net result is similar to a sequential execution of N Fetch&Adds but is performed in one indivisible operation. Two simultaneous requests are combined in a switch as illustrated in Fig. 7.11.

One of the following operations will be performed if processor P_1 executes $\text{Ans}_1 \leftarrow$ Fetch&Add(x, e_1) and P_2 executes $\text{Ans}_2 \leftarrow$ Fetch&Add(x, e_2) simultaneously on the shared variable x. If the request from P_1 is executed ahead of that from P_2, the following values are returned:

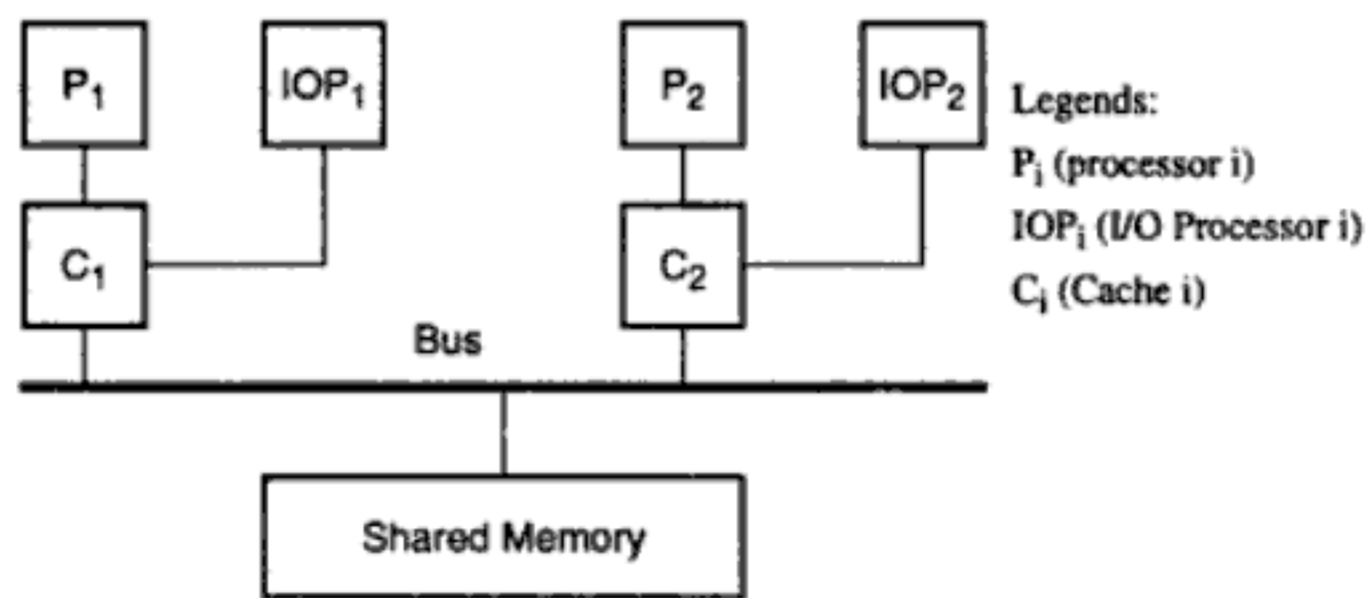
$$\begin{aligned} \text{Ans}_1 &\leftarrow x \\ \text{Ans}_2 &\leftarrow x + e_1 \end{aligned} \tag{7.2}$$

If the execution order is reversed, the following values are returned:

Copyrighted material



(a) I/O operations bypassing the cache



(b) A possible solution

Figure 7.13 Cache inconsistency after an I/O operation and a possible solution. (Adapted from Dubois, Scheurich, and Briggs, 1988)

transactions. If a bus transaction threatens the consistent state of a locally cached object, the cache controller can take appropriate actions to invalidate the local copy. Protocols using this mechanism to ensure coherence are called *snoopy protocols* because each cache snoops on the transactions of other caches.

On the other hand, scalable multiprocessor systems interconnect processors using short point-to-point wires in direct or multistage networks. Unlike the situation in buses, the bandwidth of these networks increases as more processors are added to the system. However, such networks do not have a convenient snooping mechanism and do not provide an efficient broadcast capability. In such systems, the cache coherence problem can be solved using some variant of directory schemes.

In general, a cache coherence protocol consists of the set of possible states in the local caches, the state in the shared memory, and the state transitions caused by the messages transported through the interconnection network to keep memory coherent. In what follows, we first describe the snoopy protocols and then the directory-based protocols. These protocols rely on software, hardware, or a combination of both for implementation. Cache coherence can also be enforced in the TLB or assisted by the

Copyrighted material

Cache Events and Actions The memory-access and invalidation commands trigger the following events and actions:

- *Read-miss*: When a processor wants to read a block that is not in the cache, a *read-miss* occurs. A *bus-read* operation will be initiated. If no *dirty* copy exists, then main memory has a consistent copy and supplies a copy to the requesting cache. If a *dirty* copy does exist in a remote cache, that cache will inhibit the main memory and send a copy to the requesting cache. In all cases, the cache copy will enter the *valid* state after a read-miss.
- *Write-hit*: If the copy is in the *dirty* or *reserved* state, the *write* can be carried out locally and the new state is *dirty*. If the new state is *valid*, a *write-invalidate* command is broadcast to all caches, invalidating their copies. The shared memory is *written through*, and the resulting state is *reserved* after this first *write*.
- *Write-miss*: When a processor fails to write in a local cache, the copy must come either from the main memory or from a remote cache with a dirty block. This is accomplished by sending a *read-invalidate* command which will invalidate all cache copies. The local copy is thus updated and ends up in a *dirty* state.
- *Read-hit*: Read-hits can always be performed in a local cache without causing a state transition or using the snoopy bus for invalidation.
- *Block Replacement*: If a copy is *dirty*, it has to be written back to main memory by block replacement. If the copy is *clean* (i.e., in either the *valid*, *reserved*, or *invalid* state), no replacement will take place.

Goodman's write-once protocol demands special bus lines to inhibit the main memory when the memory copy is invalid, and a *bus-read* operation is needed after a *read-miss*. Most standard buses cannot support this inhibition operation.

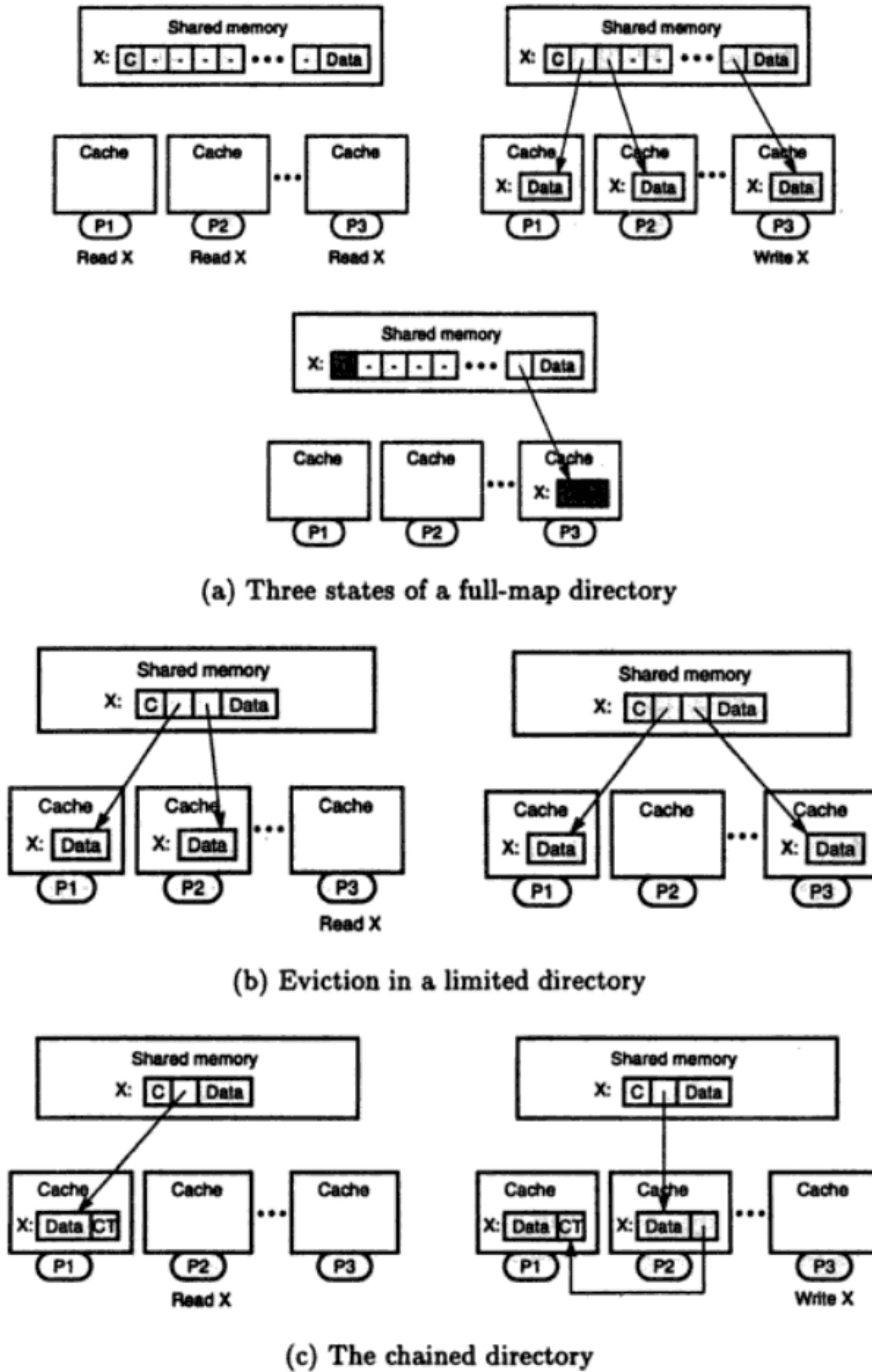
The IEEE Futurebus+ proposed to include this special bus provision. Using a write-through policy after the first *write* and using a write-back policy in all additional *writes* will eliminate unnecessary invalidations.

Snoopy cache protocols are popular in bus-based multiprocessors because of their simplicity of implementation. The write-invalidate policies have been implemented on the Sequent Symmetry multiprocessor and on the Alliant FX multiprocessor.

Besides the DEC Firefly multiprocessor, the Xerox Palo Alto Research Center has implemented another write-update protocol for its Dragon multiprocessor workstation. The Dragon protocol avoids updating memory until replacement, in order to improve the efficiency of intercache transfers.

The Futurebus+ Protocol The Futurebus+ parallel protocols support both connected and split transactions. Generally speaking, connected transactions are cheaper and easier to implement on a single bus segment. Split transactions are more complex and expensive but provide greater concurrency in building large hierarchical-bus multiprocessors. This section explains the parallel arbitration mechanism and cache coherence developed for Futurebus+-based multiprocessors. The Futurebus+ is well suited to shared-memory multiprocessors. The types of bus transactions are tuned to drive the state of various cache modules to conform with almost any cache coherence protocol. A unified cache model, called MESI, has been developed to maintain cache consistency in

Copyrighted material



(a) Three states of a full-map directory

(b) Eviction in a limited directory

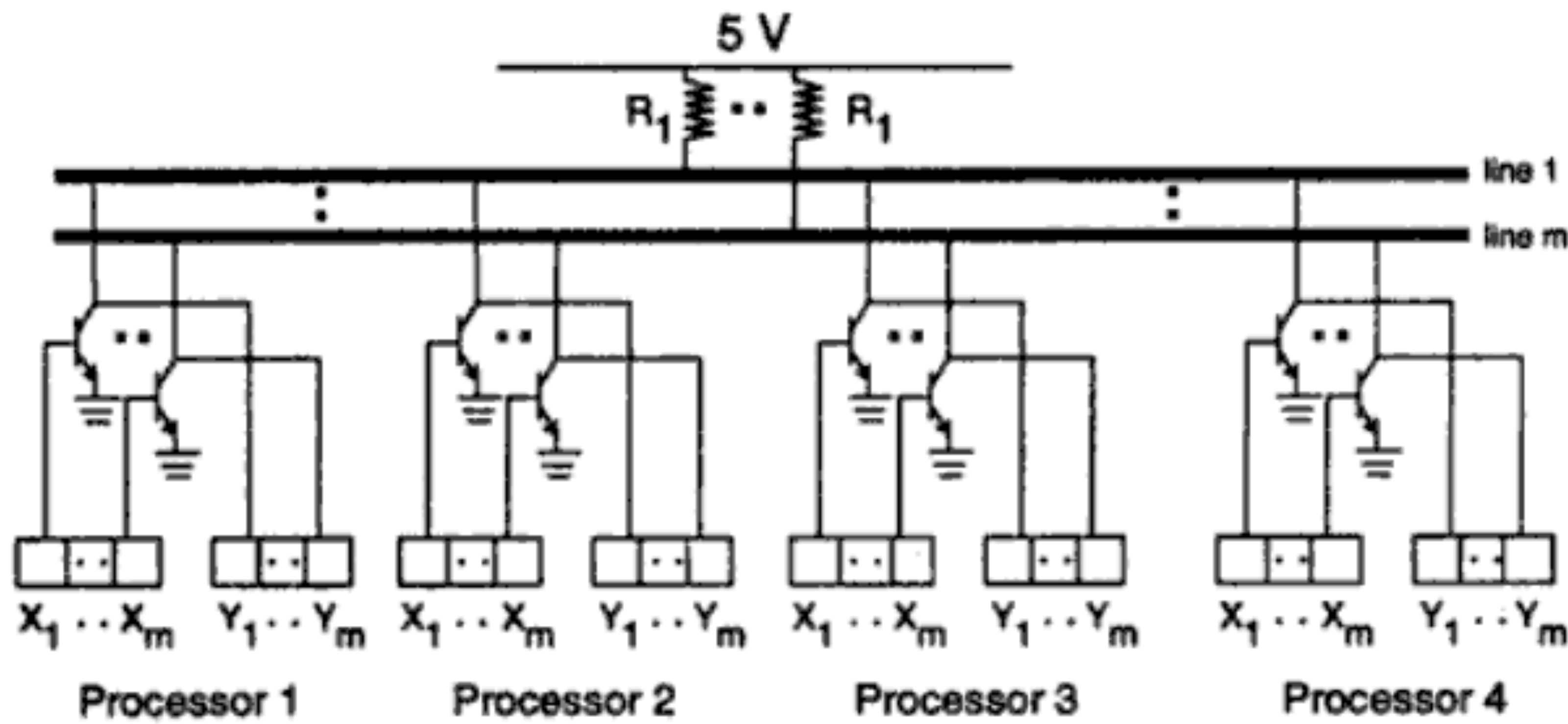
(c) The chained directory

Figure 7.19 Three types of cache directory protocols. (Courtesy of Chaiken et al., *IEEE Computer*, June 1990)

Copyrighted material

barrier counter, the synchronization point has been reached. No processor can execute beyond the barrier until the synchronization process is complete.

Wired Barrier Synchronization A wired-NOR logic is shown in Fig. 7.20 for implementing a barrier mechanism for fast synchronization. Each processor uses a dedicated control vector $X = (X_1, X_2, ..., X_m)$ and accesses a common monitor vector $Y = (Y_1, Y_2, ..., Y_m)$ in shared memory, where m corresponds to the barrier lines used.



(a) Barrier lines and interface logic

Step 1: Forking (use of one barrier line)

Line 1	Processor 1	Processor 2	Processor 3	Processor 4
X	1	1	1	1
Y	0	0	0	0

Step 2: Process 1 and Process 3 reach the synchronization point

	Process 1	Process 2	Process 3	Process 4
X	0	1	0	1
Y	0	0	0	0

Step 3: All processes reach the synchronization point

	Process 1	Process 2	Process 3	Process 4
X	0	0	0	0
Y	1	1	1	1

(b) Synchronization steps

Figure 7.20 The synchronization of four independent processes on four processors using one wired-NOR barrier line. (Adapted from Hwang and Shang, *Proc. Int. Conf. Parallel Processing*, 1991)

The number of barrier lines needed for synchronization depends on the multipro-

Copyrighted material

Table 7.1 Three Generations of Multicomputer Development

Generation	First	Second	Third
Years	1983-87	1988-92	1993-97
Typical node			
MIPS	1	10	100
Mflops scalar	0.1	2	40
Mflops vector	10	40	200
Memory (Mbytes)	0.5	4	32
Typical system			
N (nodes)	64	256	1024
MIPS	64	2560	100K
Mflops scalar	6.4	512	40K
Mflops vector	640	10K	200K
Memory (Mbytes)	32	1K	32K
Communication latency (100-byte message)			
Neighbor (microseconds)	2000	5	0.5
Nonlocal (microseconds)	6000	5	0.5

(Modified from Athas and Seitz, "Multicomputers: Message-Passing Concurrent Computers", *IEEE Computer*, August 1988).

from medium- to fine-grain multicomputers using a globally shared virtual memory.

The Second Generation These are the multicomputers which are still in use at present. A major improvement of the second generation includes the use of better processors, such as the i386 in the iPSC/2 and the i860 in the iPSC/860 and in the Delta. The nCUBE/2 implements 64 custom-designed VLSI processors on a single PC board. The memory per node has also increased to 10 times that of the first generation.

Most importantly, hardware-supported routing, such as *wormhole routing*, reduces the communication latency significantly from 6000 μs to less than 5 μs. In fact, the latency for remote and local communications has become the same, independent of the number of hops between any two nodes.

The architecture of a typical second-generation multicomputer is shown in Fig. 7.23. This corresponds to a 16-node mesh-connected architecture. Mesh routing chips (MRCs) are used to establish the four-neighbor mesh network. All the mesh communication channels and MRCs are built on a backplane.

Each node is implemented on a PC board plugged into the backplane at the proper MRC position. All I/O devices, graphics, and the host are connected to the periphery (boundary) of the mesh. The Intel Delta system has such a mesh architecture.

Another representative system is the nCUBE/2 which implements a hypercube with up to 8192 nodes with a total of 512 Gbytes of distributed memory. Note that some parameters in Table 7.1 have been updated from the conservative estimates made by Atlas and Seitz in 1988.

Copyrighted material

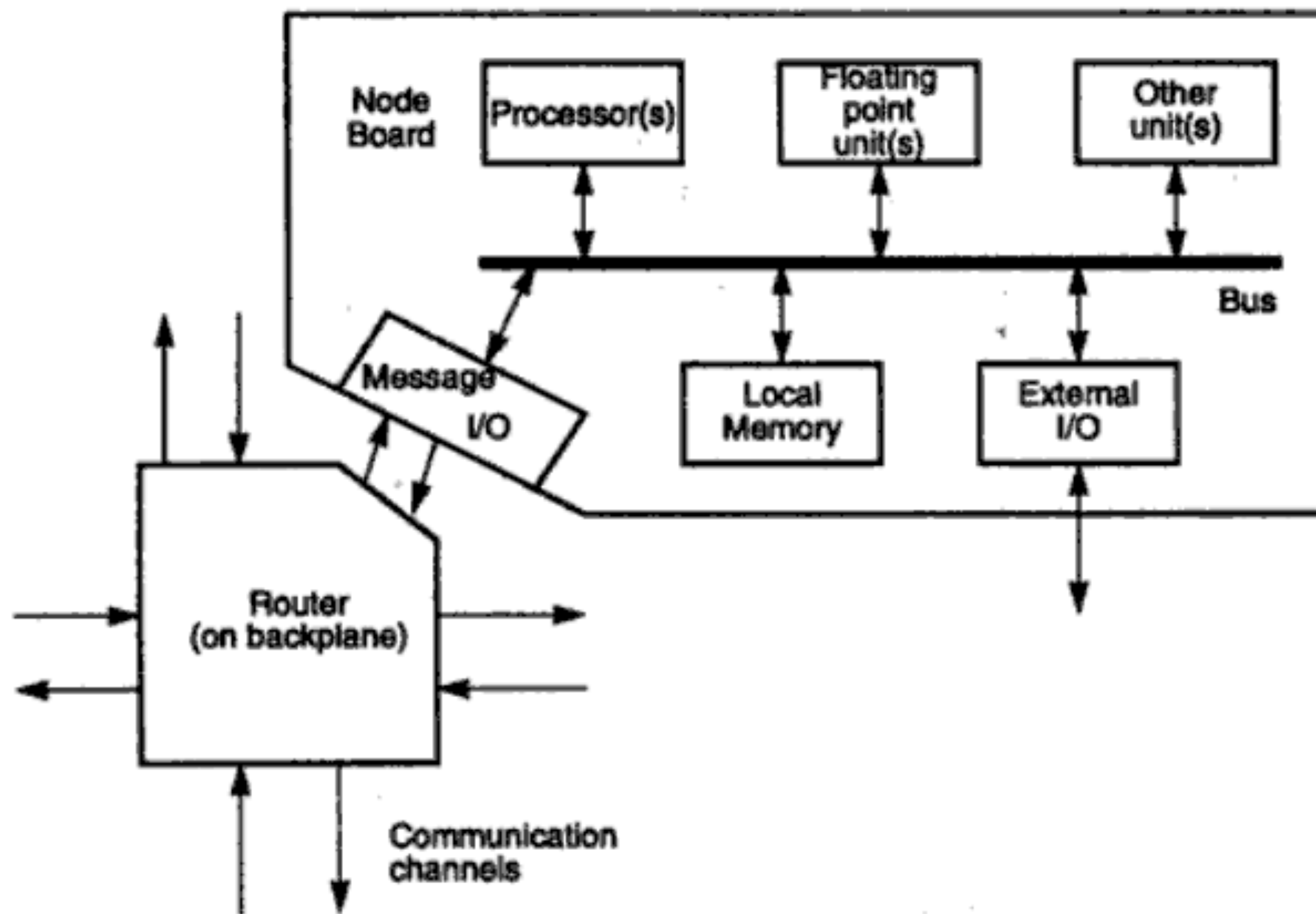


Figure 7.25 Node architecture of the Paragon multicomputer.

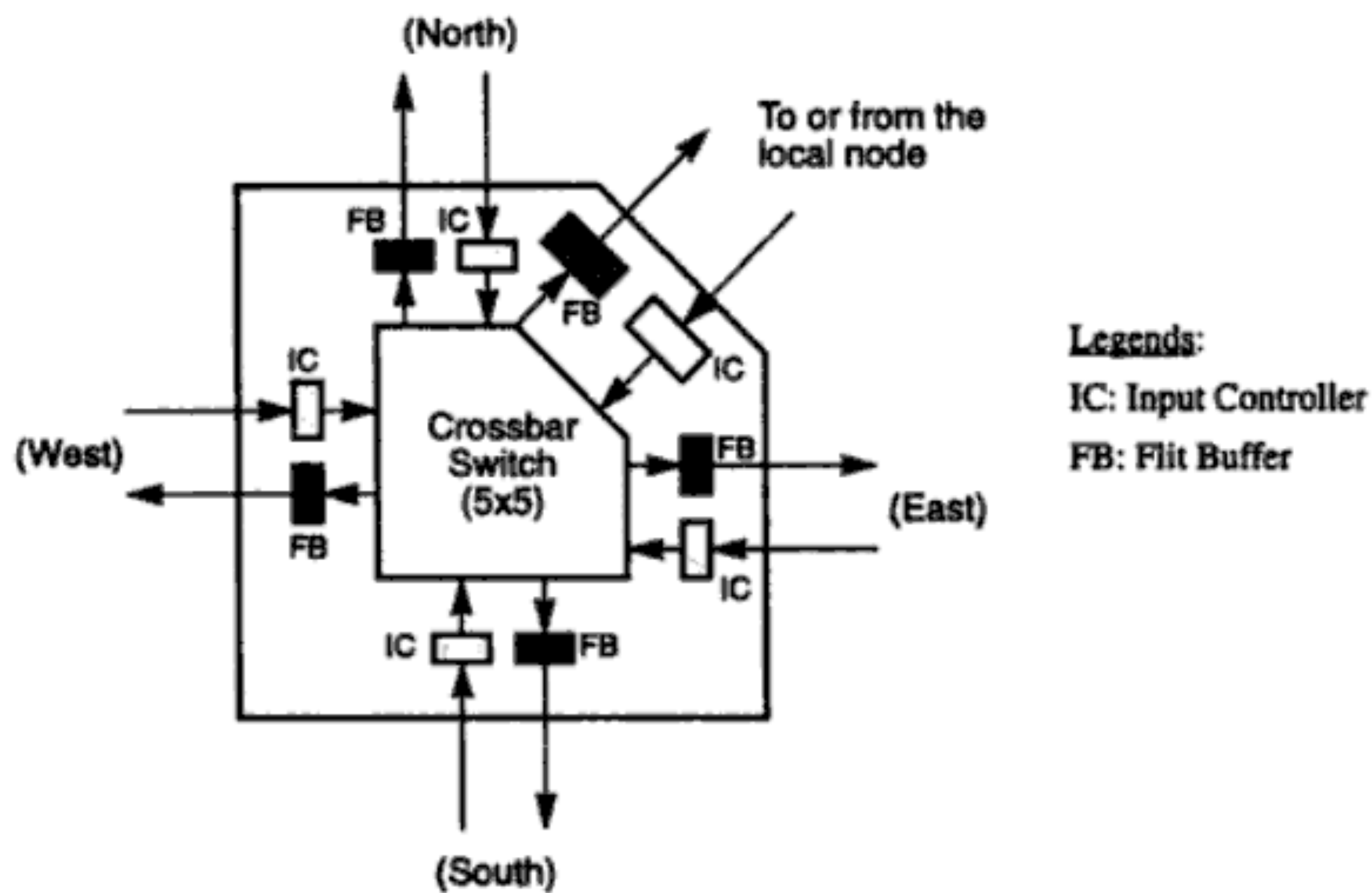
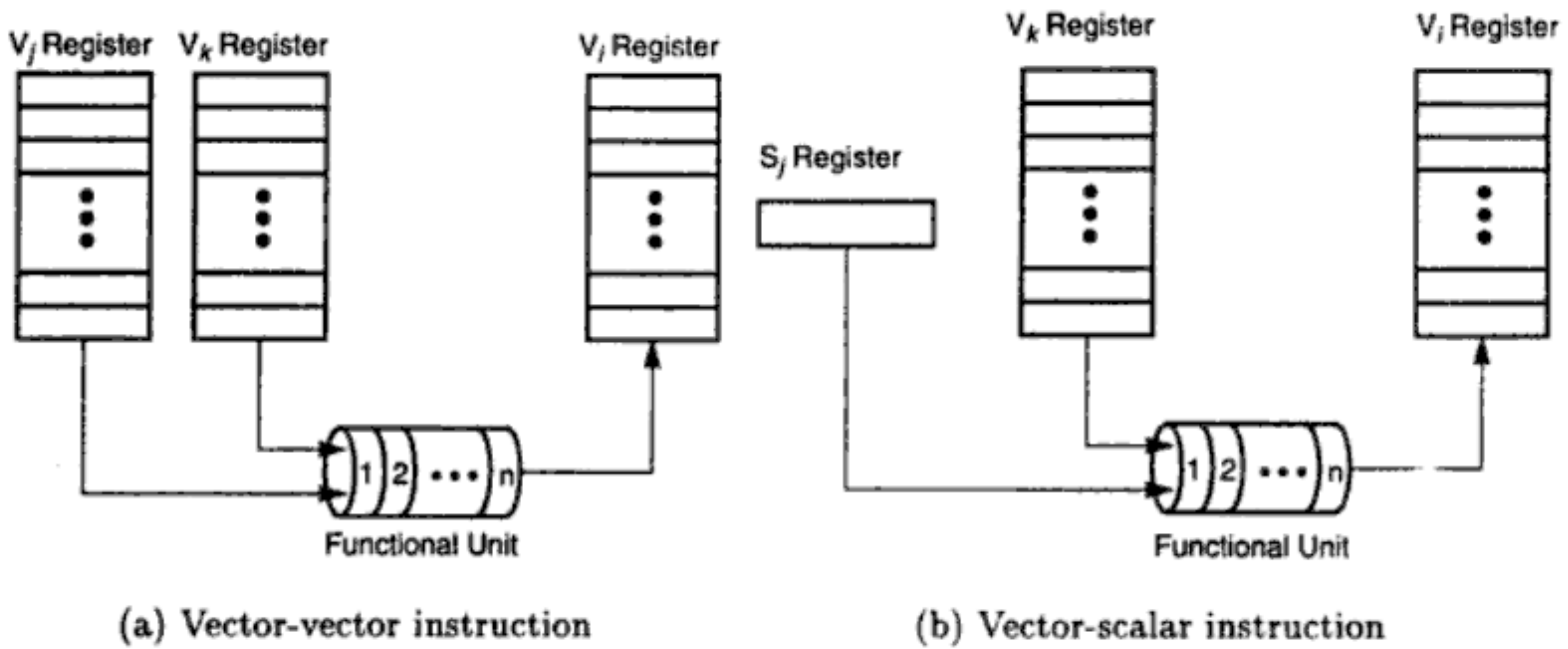
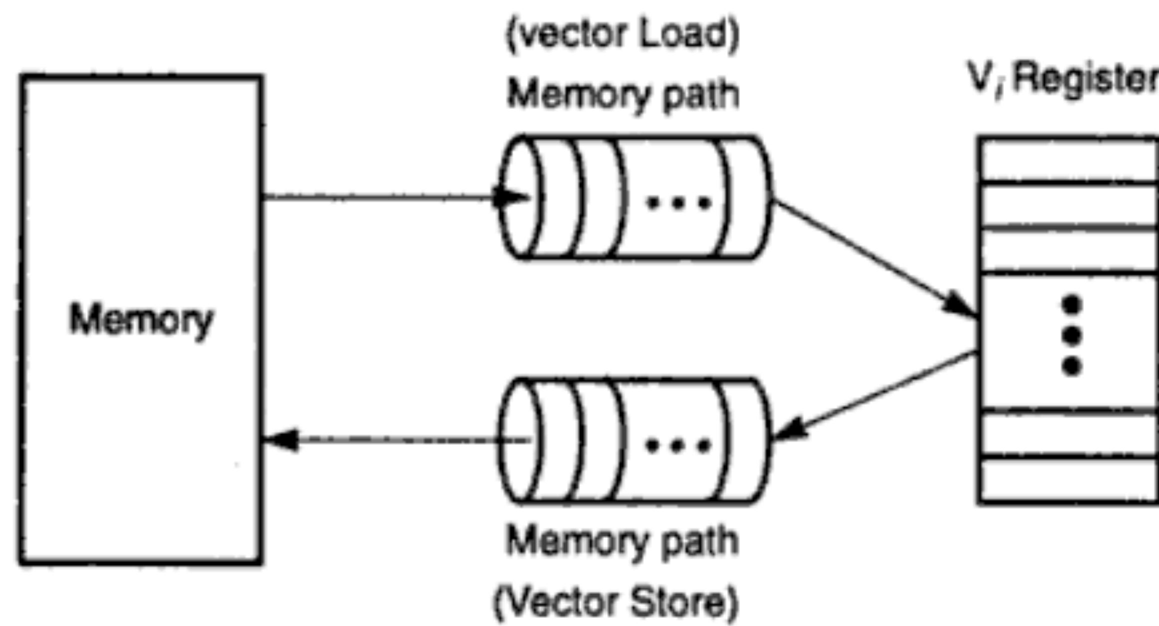


Figure 7.26 The structure of a mesh-connected router with four pairs of I/O channels connected to neighboring routers.

Copyrighted material



(a) Vector-vector instruction (b) Vector-scalar instruction



(c) Vector-memory instructions

Figure 8.1 Vector instruction types in Cray-like computers.

(M) as defined below:

$$f_4 : M \to V \qquad \textit{Vector load} \tag{8.4}$$

$$f_5 : V \to M \qquad \textit{Vector store} \tag{8.5}$$

(4) *Vector reduction instructions* — These correspond to the following mappings:

$$f_6 : V_i \to s_j \tag{8.6}$$

$$f_7 : V_i \times V_j \to s_k \tag{8.7}$$

Examples of f_6 include finding the *maximum*, *minimum*, *sum*, and *mean value* of all elements in a vector. A good example of f_7 is the *dot product*, which performs $s = \sum_{i=1}^{n} a_i \times b_i$ from two vectors $A = (a_i)$ and $B = (b_i)$.

(5) *Gather and scatter instructions* — These instructions use two vector registers to gather or to scatter vector elements randomly throughout the memory, corresponding to the following mappings:

$$f_8 : M \to V_1 \times V_0 \qquad \textit{Gather} \tag{8.8}$$

Copyrighted material

to eight processors in a single system using a 6-ns clock rate and 256 Mbytes of shared memory.

The Cray Y-MP C-90 was introduced in 1990 to offer an integrated system with 16 processors using a 4.2-ns clock. We will study models Y-MP 816 and C-90 in detail in the next section.

Another product line is the Cray 2S introduced in 1985. The system allows up to four processors with 2 Gbytes of shared memory and a 4.1-ns superpipelined clock. A major contribution of the Cray 2 was to switch from the batch processing COS to multiuser UNIX System V on a supercomputer. This led to the UNICOS operating system, derived from the UNIX/V and Berkeley 4.3 BSD, currently in use in most Cray supercomputers.

The Cyber/ETA Series Control Data Corporation (CDC) introduced its first supercomputer, the STAR-100, in 1973. Cyber 205 was the successor produced in 1982. The Cyber 205 runs at a 20-ns clock rate, using up to four vector pipelines in a uniprocessor configuration.

Different from the register-to-register architecture used in Cray and other supercomputers, the Cyber 205 and its successor, the ETA 10, have memory-to-memory architecture with longer vector instructions containing memory addresses.

The largest ETA 10 consists of 8 CPUs sharing memory and 18 I/O processors. The peak performance of the ETA 10 was targeted for 10 Gflops. Both the Cyber and the ETA Series are no longer in production but are still in use at several supercomputer centers.

Japanese Supercomputers NEC produced the SX-X Series with a claimed peak performance of 22 Gflops in 1991. Fujitsu produced the VP-2000 Series with a 5-Gflops peak performance at the same time. These two machines use 2.9- and 3.2-ns clocks, respectively.

Shared communication registers and reconfigurable vector registers are special features in these machines. Hitachi offers the 820 Series providing a 3-Gflops peak performance. Japanese supercomputers are strong in high-speed hardware and interactive vectorizing compilers.

The NEC SX-X 44 NEC claims that this machine is the fastest vector supercomputer (22 Gflops peak) ever built up to 1992. The architecture is shown in Fig. 8.5. One of the major contributions to this performance is the use of a 2.9-ns clock cycle based on VLSI and high-density packaging.

There are four arithmetic processors communicating through either the shared registers or via the shared memory of 2 Gbytes. There are four sets of vector pipelines per processor, each set consisting of two add/shift and two multiply/logical pipelines. Therefore, 64-way parallelism is observed with four processors, similar to that in the C-90.

Besides the vector unit, a high-speed scalar unit employs RISC architecture with 128 scalar registers. Instruction reordering is supported to exploit higher parallelism. The main memory is 1024-way interleaved. The extended memory of up to 16 Gbytes

Copyrighted material

8.2.2 Cray Y-MP, C-90, and MPP

We study below the architectures of the Cray Research Y-MP, C-90, and recently announced MPP. Besides architectural features, we examine the op rating systems, languages/compilers, and target performance of these machines.

Table 8.3 Architectural Characteristics of Three Current Supercomputers

Machine Characteristics	Cray Y-MP C90/16256	NEC SX-X Series	Fujitsu VP-2000 Series
Number of processors	16 CPUs	4 arithmetic processors	1 for VP2600/10, 2 for VP2400/40
Machine cycle time	4.2 ns	2.9 ns	3.2 ns
Max. memory	256M words (2 Gbytes).	2 Gbytes, 1024-way interleaving.	1 or 3 Gbytes of SRAM.
Optional SSD memory	512M, 1024M, or 2048M words (16 Gbytes).	16 Gbytes with 2.75 Gbytes/s transfer rate.	32 Gbytes of extended memory.
Processor architecture: vector pipelines, functional and scalar units	Two vector pipes and two functional units per CPU, delivering 64 vector results per clock period.	Four sets of vector pipelines per processor, each set consists of two adder/shift and two multiply/logical pipelines. A separate scalar pipeline.	Two load/store pipes and 5 functional pipes per vector unit, 1 or 2 vector units, 2 scalar units can be attached to each vector unit.
Operating system	UNICOS derived from UNIX/V and BSD.	Super-UX based on UNIX System V and 4.3 BSD.	UXP/M and MSP/EX enhanced for vector processing.
Front-ends	IBM, CDC, DEC, Univac, Apollo, Honeywell.	Built-in control processor and 4 I/O processors.	IBM-compatible hosts.
Vectorizing languages / compilers	Fortran 77, C, CF77 5.0, Cray C release 3.0	Fortran 77/SX, Vectorizer/XS, Analyzer/SX.	Fortran 77 EX/VP, C/VP compiler with interactive vectorizer.
Peak performance and I/O bandwidth	16 Gflops, 13.6 Gbytes/s.	22 Gflops, 1 Gbyte/s per I/O processor.	5 Gflops, 2 Gbyte/s with 256 channels.

The Cray Y-MP 816 A schematic block diagram of the Y-MP 8 is shown in Fig. 8.9. The system can be configured to have one, two, four, and eight processors. The eight CPUs of the Y-MP share the central memory, the I/O section, the interprocessor communication section, and the real-time clock.

Copyrighted material

```
      Do 10 I = 1, N
         Load        R1, X(I)
         Load        R2, Y(I)
         Multiply    R1, S                    (8.13)
         Add         R2, R1
         Store       Y(I), R2
   10 Continue
```

where X(I) and Y(I), I = 1, 2, ..., N, are two source vectors originally residing in the memory. After the computation, the resulting vector is stored back to the memory. S is an immediate constant supplied to the multiply instruction.

After vectorization, the above scalar SAXPY code is converted to a sequence of five vector instructions:

$$\begin{array}{rcll} \mathrm{M(x:x+N-1)} & \rightarrow & \mathrm{V1} & \textit{Vector load} \\ \mathrm{M(y:y+N-1)} & \rightarrow & \mathrm{V2} & \textit{Vector load} \\ \mathrm{S \times V1} & \rightarrow & \mathrm{V1} & \textit{Vector multiply} \\ \mathrm{V2+V1} & \rightarrow & \mathrm{V2} & \textit{Vector add} \\ \mathrm{V2} & \rightarrow & \mathrm{M(y:y+N-1)} & \textit{Vector store} \end{array} \tag{8.14}$$

The same vector notation used in Eq. 4.1 is applied here, where x and y are the starting memory addresses of the X and Y vectors, respectively; V1 and V2 are two N-element vector registers in the vector processor.

The vector code in Eq. 8.14 can be expressed as a CVF as follows, using Fortran 90 notation:

$$\mathrm{Y(1:N) = S \times X(1:N) + Y(1:N)} \tag{8.15}$$

For simplicity, we write the above expression for a CVF as follows:

$$\mathrm{Y(I) = S \times X(I) + Y(I)} \tag{8.16}$$

where the index I implies that all vector operations involve N elements.

■

Compound Vector Functions Table 8.6 lists a number of example CVFs involving one-dimensional vectors indexed by I. The same concept can be generalized to multidimensional vectors with multiple indexes. For simplicity, we discuss only CVFs defined over one-dimensional vectors. Typical operations appearing in these CVFs include *load*, *store*, *multiply*, *divide*, *logical*, and *shifting* vector operations. We use "slash" to represent the *divide* operations. All vector operations are defined on a component-wise basis unless otherwise specified.

The purpose of studying CVFs is to explore opportunities for concurrent processing of linked vector operations. The numbers of available vector registers and functional pipelines impose some limitations on how many CVFs can be executed simultaneously.

Copyrighted material

8.4.1 Implementation Models

Two SIMD computer models are described below based on the memory distribution and addressing scheme used. Most SIMD computers use a single control unit and distributed memories, except for a few that use associative memories.

The instruction set of an SIMD computer is decoded by the array control unit. The *processing elements* (PEs) in the SIMD array are passive ALUs executing instructions broadcast from the control unit. All PEs must operate in lockstep, synchronized by the same array controller.

Distributed-Memory Model Spatial parallelism is exploited among the PEs in an SIMD computer. A distributed-memory SIMD computer consists of an array of PEs which are controlled by the same array control unit, as shown in Fig. 8.22a. Program and data are loaded into the control memory through the host computer.

An instruction is sent to the control unit for decoding. If it is a scalar or program control operation, it will be directly executed by a scalar processor attached to the control unit. If the decoded instruction is a vector operation, it will be broadcast to all the PEs for parallel execution.

Partitioned data sets are distributed to all the local memories attached to the PEs through a vector data bus. The PEs are interconnected by a data-routing network which performs inter-PE data communications such as shifting, permutation, and other routing operations. The data-routing network is under program control through the control unit. The PEs are synchronized in hardware by the control unit.

In other words, the same instruction is executed by all the PEs in the same cycle. However, masking logic is provided to enable or disable any PE from participation in a given instruction cycle. The Illiac IV was such an SIMD machine consisting of 64 PEs with local memories interconnected by an 8×8 mesh with wraparound connections (Fig. 2.18b).

Almost all SIMD machines built today are based on the distributed-memory model. Various SIMD machines differ mainly in the data-routing network chosen for inter-PE communications. The four-neighbor mesh architecture has been the most popular choice in the past. Besides Illiac IV, the Goodyear MPP and AMT DAP610 were also implemented with the two-dimensional mesh. Variations from the mesh are the hypercube embedded in a mesh implemented in the CM-2, and the X-Net plus a multistage crossbar router implemented in the MasPar MP-1.

Shared-Memory Model In Fig. 8.22b, we show a variation of the SIMD computer using shared memory among the PEs. An alignment network is used as the inter-PE memory communication network. Again this network is controlled by the control unit.

The Burroughs Scientific Processor (BSP) has adopted this architecture, with $n = 16$ PEs updating $m = 17$ shared-memory modules through a 16×17 alignment network. It should be noted that the value m is often chosen to be relatively prime with respect to n, so that parallel memory access can be achieved through skewing without conflicts.

The alignment network must be properly set to avoid access conflicts. Most SIMD computers are built with distributed memories. Some SIMD computers use bit-slice PEs,

Copyrighted material

implemented by the 1.3 Gbytes/s global router network.

The disk array provides up to 17.3 Gbytes of formatted capacity with a 9-Mbytes/s sustained disk I/O rate. The parallel disk array is a necessity to support data-parallel computation and provide file system transparency and multilevel fault tolerance.

8.5 The Connection Machine CM-5

The grand challenge applicationswill drive the development of present and future MPP systems to achieve the 3T performance goals. The Connection Machine Model CM-5 is the latest effort of Thinking Machines Corporation toward this end. We describe below the innovations surrounding the CM-5 architectural development, its building blocks, and the application paradigms.

8.5.1 A Synchronized MIMD Machine

The CM-2 and its predecessors were criticized for having a rigid SIMD architecture, limiting general-purpose applications. The CM-5 designers liberated themselves by choosing a universal architecture, which combines the advantages of both SIMD and MIMD machines.

Traditionally, supercomputer programmers were forced to choose between MIMD and SIMD computers. An MIMD machine is good at independent branching but bad at synchronization and communication. On the contrary, an SIMD machine is good at synchronization and communication but poor at branching. The CM-5 was designed with a synchronized MIMD structure to support both styles of parallel computation.

The Building Blocks The CM-5 architecture is shown in Fig. 8.27. The machine contains from 32 to 16,384 *processing nodes*, each of which can have a 32-MHz SPARC processor, 32-Mbytes of memory, and a 128-Mflops vector processing unit capable of performing 64-bit floating-point and integer operations.

Instead of using a single sequencer (as in the CM-2), the system uses a number of *control processors*, which are Sun Microsystems workstation computers. The number of control processors, varying with different configurations, ranges from one to several tens. Each control processor is configured with memory and disk based on the needs.

Input and output are provided via high-bandwidth *I/O interfaces* to graphics devices, mass secondary storage such as a data vault, and high-performance networks. Additional low-speed I/O is provided by Ethernet connections to the control processors. The largest configuration is expected to occupy a space of 30 m × 30 m, and is shooting for a peak performance over 1 Tflops.

The Network Functions The building blocks are interconnected by three networks: a *data network*, a *control network*, and a *diagnostic network*. The data network provides high-performance, point-to-point data communications between the processing nodes. The control network provides cooperative operations, including broadcast, synchronization, and scans, as well as system management functions.

The diagnostic network allows "back-door" access to all system hardware to test

Copyrighted material

(f) Sparse matrix and masking instruction.

Problem 8.3 Explain the following memory organizations for vector accesses:

(a) S-access memory organization.
(b) C-access memory organization.
(c) C/S-access memory organization.

Problem 8.4 Distinguish among the following vector processing machines in terms of architecture, performance range, and cost-effectiveness:

(a) Full-scale vector supercomputers.
(b) High-end mainframes or near-supercomputers.
(c) Minisupercomputers or supercomputing workstations.

Problem 8.5 Explain the following terms associated with compound vector processing:

(a) Compound vector functions.
(b) Vector loops and pipeline chaining.
(c) Systolic program graphs.
(d) Pipeline network or pipenets.

Problem 8.6 Answer the following questions related to the architecture and operations of the Connection Machine CM-2:

(a) Describe the processing node architecture, including the processor, memory, floating-point unit, and network interface.
(b) Describe the hypercube router and the NEWS grid and explain their uses.
(c) Explain the scanning and spread mechanisms and their applications on the CM-2.
(d) Explain the concepts of broadcasting, global combining, and virtual processors in the use of the CM-2.

Problem 8.7 Answer the following questions about the MasPar MP-1:

(a) Explain the X-Net mesh interconnect (the PE array) built into the MP-1.
(b) Explain how the multistage crossbar router works for global communication between all PEs.
(c) Explain the computing granularity on PEs and how fast I/O is performed on the MP-1.

Problem 8.8 Answer the following questions about the Connection Machine CM-5:

Copyrighted material

as units of coherence. This tends to increase false-sharing activity.

Table 9.1 Representative Shared-Virtual-Memory Systems (Excerpts from Nitzberg and Lo, *IEEE Comput.*, August 1991)

System and Developer	Implementation and Structure	Coherence Semantics and Protocols	Special mechanics for Performance and Synchronization
Stanford Dash (Lenoski, Laudon, Gharachorloo, Gupta, and Hennessy, 1988–).	Mesh-connected network of Silicon Graphics 4D/340 workstations with added hardware for coherent caches and prefetching.	Release memory consistency with write-invalidate protocol.	Relaxed coherence, prefetching, and use queued locks for synchronization.
Yale Linda (Carriero and Gelernter, 1982–).	Software-implemented system based on the concepts of tuple space with access functions to achieve coherence via virtual memory management.	Coherence varies with environment; hashing is used in associative search; no mutable data.	Linda can be implemented for many languages and machines using C-Linda or Fortran-Linda interfaces.
CMU Plus (Bisiani and Ravishankar, 1988–).	A hardware implementation using MC 88000, Caltech mesh, and Plus kernel.	Uses processor consistency, nondemand write-update coherence, delayed operations.	Pages for sharing, words for coherence, complex synchronization instructions.
Princeton Shiva (Li and Schaefer, 1988).	Software-based system for Intel iPSC/2 with a Shiva/native operating system.	Sequential consistency, write-invalidate protocol, 4-Kbyte page swapping.	Uses data structure compaction, messages for semaphores and signal.wait, distributed memory as backing store.

Scalability issues of SVM architectures include determining the sizes of data structures for maintaining memory coherence and how to take advantage of the fast data transmission among distributed memories in order to implement large SVM address spaces. Data structure compaction and page swapping can simplify the design of a large SVM address space without using disks as backing stores. A number of alternative choices are given in Li (1992).

Copyrighted material

Compiler Support To support data-parallel programming, the array language expressions and their optimizing compilers must be embedded in familiar standards such as Fortran 77, Fortran 90, and C. The idea is to unify the program execution model, facilitate precise control of massively parallel hardware, and enable incremental migration to data-parallel execution.

Compiler-optimized control of SIMD machine hardware allows the programmer to drive the PE array transparently. The compiler must separate the program into scalar and parallel components and integrate with the UNIX environment.

The compiler technology must allow array extensions to optimize data placement, minimize data movement, and virtualize the dimensions of the PE array. The compiler generates data-parallel machine code to perform operations on arrays.

Array sectioning allows a programmer to reference a section or a region of a multidimensional array. Array sections are designated by specifying a start index, a bound, and a stride. *Vector-valued subscripts* are often used to construct arrays from arbitrary permutations of another array. These expressions are vectors that map the desired elements into the target array. They facilitate the implementation of *gather* and *scatter* operations on a vector of indices.

SIMD programs can be recompiled for MIMD architecture. The idea is to develop a source-to-source precompiler to convert, for example, from Connection Machine C* programs to C programs running on an nCUBE message-passing multicomputer in SPMD mode.

In fact, SPMD programs are a special class of SIMD programs which emphasize medium-grain parallelism and synchronization at the subprogram level rather than at the instruction level. In this sense, the data-parallel programming model applies to both synchronous SIMD and loosely coupled MIMD computers. Program conversion between different machine architectures is very much needed to broaden software portability.

10.1.4 Object-Oriented Model

If one considers special language features and their implications, additional models for parallel programming can be introduced. An object-oriented programming model is characterized below.

In this model, *objects* are dynamically created and manipulated. Processing is performed by sending and receiving messages among objects. Concurrent programming models are built up from low-level objects such as processes, queues, and semaphores into high-level objects like monitors and program modules.

Concurrent OOP The popularity of *object-oriented programming* (OOP) is attributed to three application demands: First, there is increased use of interacting processes by individual users, such as the use of multiple X windows. Second, workstation networks have become a cost-effective mechanism for resource sharing and distributed problem solving. Third, multiprocessor technology has advanced to the point of providing supercomputing power at a fraction of the traditional cost.

As a matter of fact, program abstraction leads to program modularity and software reusability as is often found in OOP. Other areas that have encouraged the growth of

Copyrighted material

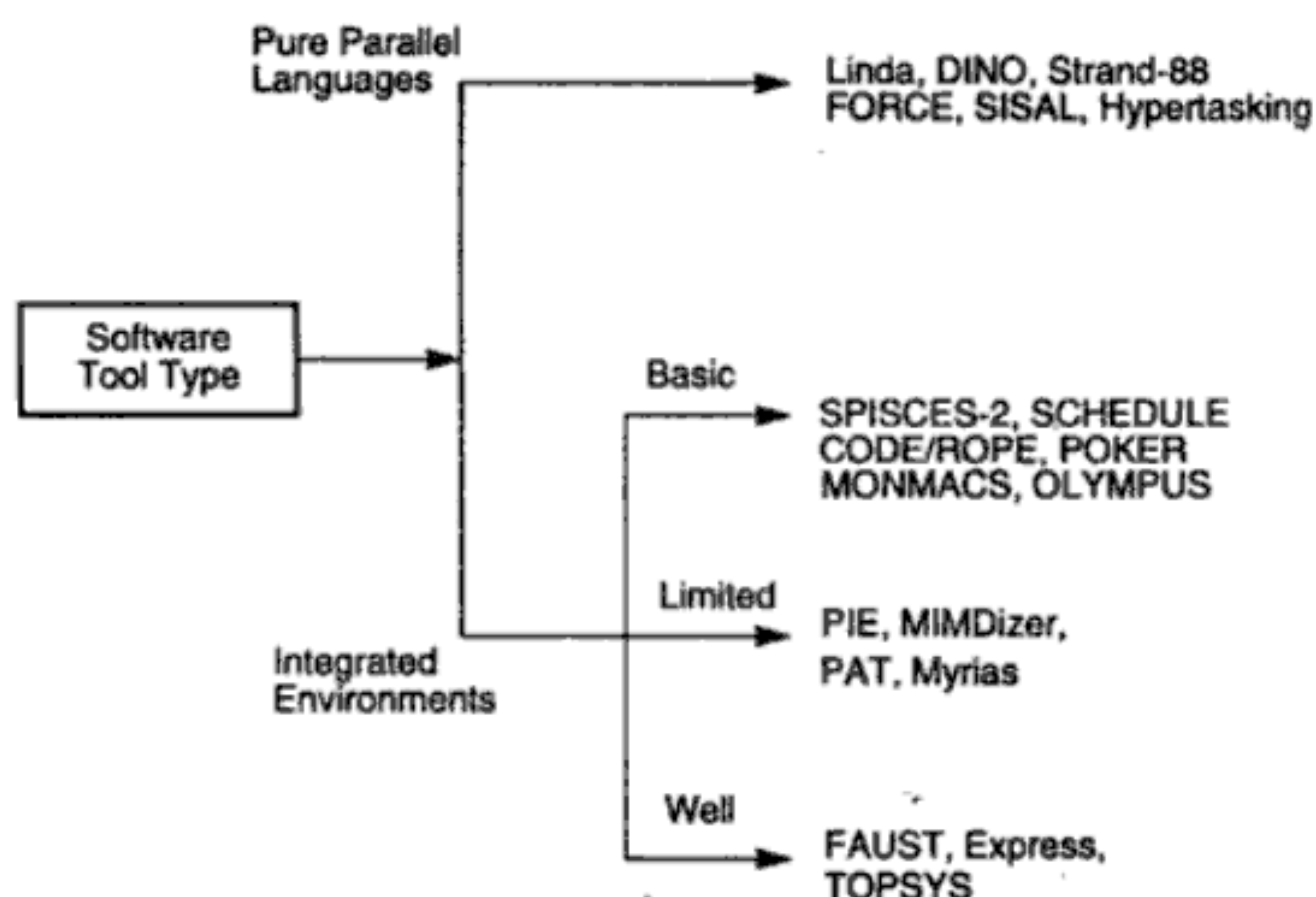


Figure 11.1 Software tool types for parallel programming. (Courtesy of Chang and Smith, 1990)

Limited integration provides tools for parallel debugging, performance monitoring, or program visualization beyond the capability of the basic environments listed. Well-developed environments provide intensive tools for debugging programs, interaction of textual/graphical representations of a parallel program, visualization support for performance monitoring, program visualization, parallel I/O, parallel graphics, etc.

The classification of a particular tool will likely change with time. A pure language might become an integrated environment for parallel programming someday when a complete set of tools is added to it. For example, C-Linda and Fortran-Linda were developed to help C and Fortran programmers write parallel programs using the tuple spaces in Linda.

Environment Features In designing a parallel programming language, one often faces a dilemma involving *compatibility*, *expressiveness*, *ease of use*, *efficiency*, and *portability*. Parallel languages are developed either by introducing new languages such as Linda and Occam or by extending existing sequential languages such as Fortran 90, C*, and Concurrent Pascal.

A new parallel programming language has the advantage of using high-level parallel concepts or constructs for parallelism instead of using imperative (algorithmic) languages which might have a sequential nature or exhibit the von Neumann bottleneck and therefore are inherently sequential. New languages are usually difficult to market for acceptance in a short time period because they are often incompatible with existing sequential languages.

Most parallel computer designers choose the language extension approach to solving the compatibility problem. High-level parallel constructs were added to Fortran, C, Pascal, and Lisp to make them suitable for use on parallel computers. Special optimiz-

Copyrighted material

11.3 Shared-Variable Program Structures

We describe below the use of spin locks, suspend locks, binary and counting semaphores, and monitors for shared-variable programming. These mechanisms can be used to implement various synchronization methods among concurrent processes. Shared-variable constructs are also used in OS kernel development for protected access of certain kernel areas.

11.3.1 Locks for Protected Access

Lock and unlock mechanisms are described below using shared variables among multiple processes. *Binary locks* are used globally among multiple processes. *Spin locks* are based on a time-slot concept. *Dekker's locks* are based on using distributed requests jointly with a spin lock. Special multiprocessor instructions are needed to implement these locking mechanisms.

Spin Locks The entrance and exit of a CS can be controlled by a binary *spin lock* mechanism in which the gate is protected by a single binary variable x, which is shared by all processes attempting to enter the CS.

Example 11.2 Definition of a binary spin lock

The gate variable x is initially set to 0, corresponding to the *open* status. Each process P_i is allowed to test the value of x until it becomes 0. Then it can enter the CS. The gate must be *closed* by setting $x = 1$ after entering.

```
Shared var x: (0,1)          /The spin lock/
    x := 0                   /The CS is open initially/
Process Pi for all i
    Repeat
      :                      /Spinning with busy wait/
    Until x = 0
    x := 1                   /Close gate after entry/
      :                      /The critical section/
    x := 0                   /Open gate after done and exit/
```

After the CS is completed, the gate is reopened. A busy-wait protocol is used in spin locks. Precautions must be exercised to prevent simultaneous entries into the CS by multiple processes.

■

Example 11.3 Definition of a generalized spin lock with n possible values

One way to guarantee mutually exclusive entry is to use a *generalized spin lock* with n processes as defined below. The gate variable x is allowed to assume n

Copyrighted material

Performance Tuning One technique for tuning the performance is to use double-buffer messages in a manager-worker decomposition. The idea is to send a worker two pieces of work the first time. As soon as the worker is finished working on the first piece, another piece is readily available.

Once the communication/computation issues have been well balanced, the results from the first piece of work are returned to the manager who can send over another piece of work before the worker finishes the second piece. There is always one job waiting in the worker's queue, and thus workers are kept busy all the time.

A potential problem with the manager-worker approach is that the manager may become the bottleneck. According to Intel iPSC experience, up to 50 workers managed by a single manager do not create a serious bottleneck problem as long as a good communication/computation ratio can be maintained.

Clearly, for a multicomputer consisting of thousands of processors in the future, the manager bottleneck problem will become more serious. One can always consider a hierarchy of managers.

Another possible solution is to use floating managers or multitasking processors, which can execute both a worker process or a manager process on the same processor. These options must be carefully experimented with before they can be adopted in real applications.

11.5.3 Heterogeneous Processing

In this section, we learn how to combine object-oriented programming with message-passing techniques for distributed computing applications. We first characterize objects in relation to parallelism. Then we illustrate the *object decomposition* concept using an air traffic control simulation example.

Finally, we present the concept of *layered parallelism* using a seismic monitoring example which involves all the decomposition techniques we have learned to solve very large-scale problems.

Objects and Parallelism The object-oriented approach to parallel programming offers a formal basis for decomposing the data structures and threads of control in user programs. In what follows, we define objects and reveal the relationship between objects and parallel processing.

The idea of objects comes from *data abstraction*, in order to hide low-level details from programmers for large-scale problems, such as in archive data/knowledge processing. An object encompasses a set of logically related data and a set of procedures which operate on the object's data, as illustrated by the example in Fig. 11.11a.

The example shows that temporary storage in the form of a stack can be treated as an object consisting of a last-in-first-out queue of data which can be pushed down or popped up in its management. It should be noted that an object type (class) is conceptually different from instances of the object type.

Those instances, called objects, are the ones used in program execution. Only the object's procedures have access to the object's data. A programmer can be freed from knowing the detailed implementation of the objects.

Copyrighted material

Index

Accetta, M., 687, 695, 698, 712
Acosta, R. D., 322
Active Memory Technology (AMT)
DAP600 Series, 33, 83, 368, 555
DAP610, 32, 44, 121, 447, 448
Actor model, 557
Actus, 661
Ada, 13, 646, 657
Adaptive routing, 375, 382, 384, 387, 392, 394, 401
Address arithmetic unit (AAU), 507, 512
Address translation cache (ATC), 167, 169
Address-only transfer, 221
Addressing mode, 162–164, 166, 167, 208, 209
Advanced Micro Device (AMD) 29000, 170, 171, 315
Adve, S. V., 256, 257
Agarwal, A., 23, 44, 141, 142, 149, 500, 539
Agha, G., 557, 558, 612
Aho, A., 149
Ahuja, S., 661
Aiken, H., 4
Alewife, 23, 44, 475, 494, 501, 538, 539
Algebraic optimization, 581, 583
Algol, 5
Algorithm, 7, 33, 34
CRCW, 37
deterministic, 34
EREW, 37
nondeterministic, 34
nonpolynomial, 34
NP-complete, 315
parallel, 7, 8, 32, 34, 35, 38, 49
polynomial-complexity, 34
Allan, S. J., 612
ALLCACHE, 522, 523
Allen, J. R., 567, 612
Allen, M., 322
Alliant Computer Systems Corporation
FX, 355
FX Fortran compiler, 560, 567, 619
FX/2800, 347
FX/2800-200, 115
FX/80, 23, 589
Almasi, G. S., 46
ALU dependence, 54
Alverson, R., 44, 539
Amdahl Corporation
470/V6, 234
470/V7, 234
470V/8, 280
Amdahl's law, 105, 111, 112, 124, 129–134, 136, 149, 151
Amdahl, G., 131, 149
Ametek 2010, 371, 394
Ametek S/14, 368
Analytical modeling, 236
Anaratone, M., 84, 96
Anderson, D. W., 322
Andrews, G. R., 612
Antidependence, 52, 289, 312, 315
APAL, 33
Apollo workstation, 476
Arbiter, 215
Arbitration competition, 218
Archibald, J., 257
Architecture-independent language, 9
Architecture-neutral distribution format (ANDF), 702
Arden, B. W., 96
Ardent Computer Corporation, 432
Arithmetic and logic unit (ALU), 161, 167, 174, 177, 281, 299, 318, 507
Arithmetic mean, 108
Array processing language, 555
Array processor, 120, 555
Artificial intelligence (AI), 7, 13, 46, 157,

Copyrighted material

C-90, 422–424
CFT, 60
CFT compiler, 567
Cray 1, 29, 44, 316, 410, 411, 438, 471
Cray 1S, 410, 411
Cray 2, 412, 417, 421, 620
Cray 2S, 412
MPP, 5, 6, 44, 423–424
T3D, 423, 424
X-MP, 5, 6, 411, 438, 439, 469, 471, 591, 620, 630, 632, 633
Y-MP, 29, 30, 44, 95, 115, 143, 145, 162, 347, 406, 411, 412, 417, 419, 421, 422, 424, 440, 469, 472, 505, 617, 621, 628, 630, 633, 661
Cray, S. 410
Critical section (CS), 548, 549, 629, 634–637, 640, 641, 661, 662
Crossbar switch network, 76, 78, 101, 89, 94–96, 336, 339, 340, 347, 380, 395, 421, 429, 432, 455, 486, 534
Crossing dependence, 577
Crosspoint switch, 338, 339, 395
Cube-connected cycle (CCC), 26, 85–86, 89, 96, 100
Culler, D. E., 494, 534, 539
Cybenko, G., 143
Cycle scheduling, 594
Cycle time, 14, 15
Cycles per instruction (CPI), 14–17, 47, 48, 115, 150, 152, 157–160, 163, 164, 167, 170, 177, 180, 182, 209
Cypress
CY7C157, 171
CY7C601, 170–172
CY7C602, 170–172
CY7C604, 170, 171

DADO, 83
Daisy chain, 218, 219
Dally, W. J., 19, 43, 87, 88, 96, 103, 394, 507, 511, 514, 539
Dash, 23, 42, 49, 475, 476, 478, 480–482, 489, 494, 501, 516–521, 538, 539
Data dependence, 51–53, 57, 268, 280–282, 285, 288, 290, 291, 297, 305, 306, 310, 312, 313, 322, 564, 568, 570, 579, 588, 592, 626
Data Diffusion Machine (DDM), 23
Data parallelism 120
Data pollution point, 357
Data structure, 7, 8, 14, 72, 76, 187, 629, 643, 644, 648, 649, 652, 654, 656
dynamic, 649
shared, 625, 657
static, 648, 649
Data token, 71, 72
Data transfer bus (DTB), 214, 215, 221
Data-driven, 70, 71, 74–76
Data-parallel programming model, 554–556, 612, 623
Data-routing function, 76–78
Database management, 7, 13, 26, 117, 118, 120
Dataflow architecture, 534, 539
Dataflow computer, 70–72, 75, 475, 531–534, 536, 540, 559, 617
dynamic, 45, 533, 540
static, 45, 533, 540
Dataflow graph, 72, 74, 531–534, 559, 560
Dataflow language, 559, 561
Dataflow multithreading, 491
Davis, E. W., 455, 469
Davison, E. S., 274, 277, 322
Deadlock, 222, 375, 379–382, 388, 391, 392, 548, 626, 638–640, 663, 664
buffer, 375, 380, 399
channel, 375, 380, 399
static prevention, 640
Deadlock avoidance, 26, 375, 387, 401, 549, 640
dynamic, 640
Deadlock recovery, 640
Deadlock-free allocation, 640
Debit credit benchmark, 118
Degree of chaining, 439, 441, 442
Degree of parallelism (DOP), 291
Degree of superscalar processor, 159, 178, 180, 210
DeGroot, D., 46, 612
Dekel, E., 149
Dekker's protocol, 635–637, 662
Delay insertion, 279
Delay slot, 292, 295–297, 312, 319
Delayed branch, 292, 295–297, 312
Delta network, 337, 393, 398
Demand-driven, 70, 71, 74–76

Copyrighted material

Instruction execution rate, 106, 111, 115, 150, 151
Instruction fetch, 280, 281, 283, 310, 311, 325
Instruction format, 162, 163, 174, 209
Instruction issue, 57, 265, 311, 312, 315–317, 321–323
Instruction issue rate, 159, 208, 309, 317, 321
Instruction issue latency, 159, 160, 208, 309
Instruction lookahead, 6
Instruction prefetching, 265, 280, 281, 283, 284
Instruction reordering, 412, 583, 585
Instruction set, 579, 598
Instruction window, 181
Instruction write-back, 280, 284, 325
Instruction-level parallelism, 120
Instruction-set architecture (ISA), 15, 162, 209, 507, 622
Integer unit (IU), 54, 172, 179, 180, 227
Integrated circuit, 4
  large-scale integration (LSI), 4
  medium-scale integration (MSI), 4, 5
  small-scale integration (SSI), 4, 5
  very large-scale integration (VLSI), 4, 6, 157, 163, 412, 506, 507, 525
Intel Corporation
  8086/8088, 162, 167, 203
  8008, 167
  8080, 167
  8085, 167
  8087, 162
  80186, 167
  80286, 162, 167, 203, 368
  80287, 162
  80387, 162
  i386, 24, 162, 167, 203
  i486, 24, 158, 162, 165, 167, 168, 174, 203, 208–210, 225
  i586, 164, 167, 177, 210
  i860, 26, 27, 158, 170, 171, 174, 175, 208, 210, 227, 234, 347, 370, 373, 621, 622
  i860XP, 372
  i960, 57, 297
  iPSC, 84, 620, 645–647, 656, 661, 685, 701
  iPSC-VX compiler, 567
  iPSC/1, 25, 84, 368, 369, 372, 505, 506, 684, 685
  iPSC/2, 84, 370, 372, 478, 479, 684, 685
  iPSC/860, 370, 372, 373, 617, 620, 621, 661, 685
  iWarp, 84, 96, 394
  NX/2, 667, 685
  Paragon, 5, 6, 25, 26, 43, 83, 85, 121, 143, 367, 369, 372, 373, 394, 501, 686
  Paragon XP/S, 621, 661
  Touchstone Delta, 114, 115, 121, 143, 370, 621, 686
Inter-PE communication, 71, 77
Interactive compiler optimization, 427
Interactive scheduling, 693
Interconnection network, 75–95, 476, 482, 518, 519, 525, 527
  direct, 75
  dynamic, 75, 76, 89–95
  indirect, 75
  performance. 80
  static, 75–76, 80–88
Interdomain socket, 695
Internal data forwarding, 265, 280, 282, 283, 286, 287, 291, 292, 305
Interprocess communication (IPC), 547–549, 668, 683, 686–688, 694–697, 702, 703, 711, 712
Interprocessor communication latency, 21, 33, 124
Interprocessor-memory network (IPMN), 331
Interrupt, 215, 218, 221, 228, 335, 364, 373
  interprocessor, 364
  priority, 221
Interrupt handler, 215, 221
Interrupt mechanism, 221
Interrupt message, 648
Interrupter, 215, 221
Interstage connection (ISC), 91
Invalidation-based cache coherence protocol, 518, 520
Isoefficiency, 126
Isoefficiency function, 149
Iteration space, 568, 575, 577, 601, 603, 605, 607–608, 610
IVY, 476, 478

Copyrighted material

19, 24, 41, 76–78, 89, 91–93, 96, 101, 331, 336, 358, 393, 418, 447, 454
Multitasked clustering, 701
Multitasking, 254, 426, 545, 549, 562, 591, 621, 628–633, 656, 668–671, 678, 683, 709
Multithreaded architecture, 75, 148, 475, 491, 498, 500, 516, 531, 537–539, 550
Multithreaded kernel, 667, 672, 677, 678, 680–682, 712, 714
Multithreaded MPP, 491, 536
Multithreaded multitasking, 678, 686–688
Multithreaded processor, 49, 494, 596
Multithreaded server, 709
Multithreading, 3, 44, 475, 491–500, 514, 525, 530, 534, 538, 550, 562, 671, 678, 680, 682, 690, 701, 709, 713
Multiway shuffle, 91
Multiple-context processor, 491, 494, 496, 539
Mutual exclusion, 198, 548, 549, 551, 552, 555

N-queens problem, 654
Nanobus, 222, 334, 335, 393
Nassimi, D., 149
National Semiconductor
  NS32532, 167
NCUBE Corporation
  3200, 614
  6400, 394
  nCUBE/10, 368
  nCUBE/2, 26, 43, 115, 120, 370
Near-supercomputer, 24, 429
NEC, 143, 318
  SX series, 30, 410, 412, 441, 469, 472
  SX-3, 115
Network control strategy, 332
  centralized, 332
  distributed, 332
Network diameter, 77, 80, 82–86, 89
Network flow control strategy, 26
Network interface unit, 538
Network latency, 80, 87, 89
Network partitioning, 391
Network Queueing System (NQS), 621
Network server, 696
Network throughput, 87–88
Network UNIX host, 683
Network-connected system, 348
New Japanese national computer project (NIPT), 536
NeXT Computer, 686, 688, 691, 694, 701, 713
Ni, L. M., 13, 46, 105, 107, 129, 134, 137, 149, 394, 401
Nickolls, J. R., 469
Nicolau, A., 149, 613
Nikhil, R. S., 45, 492–494, 532, 534, 536, 537, 539, 540
Nitzberg, B., 208, 478, 479, 489, 539
No-remote-memory-access (NORMA) model, 24, 48, 690, 714
Node degree, 77, 82–86, 89
Node duplication, 67
Node name, 64
Node splitting, 588
Node-addressed model, 683, 714
Noncacheable data, 363
Nonuniform-memory-access (NUMA) model, 19, 22–24, 42, 46, 48, 331, 368, 476, 690, 701, 714
NP-class problem, 34
NP-complete problem, 34, 569
NP-hard problem, 67
NuBus, 222, 393
Numerical computing, 7
Nussbaum, D, 141, 142, 149

Object decomposition, 644, 656, 657, 664
Object-oriented model, 683, 686, 688, 695, 712, 714
Object-oriented programming (OOP), 514, 556, 557, 643, 656, 657
Occam, 646, 647, 661
Oldehoeft, R., 612
Omega network, 86, 91, 96, 101–103, 142, 143, 153, 336, 337, 341–343, 345, 347, 393, 398, 399, 486, 534
Open Software Foundation OSF/1, 545, 621, 622, 667, 686, 701–703, 708, 712
Operand fetch, 280, 281, 308, 325
Operating system
  Cray operating system (COS), 411, 412
  UNICOS, 412, 419
  ULTRIX, 430

Copyrighted material

Seitz, C. L., 19, 39, 43, 46, 368–370, 394, 514, 515, 539, 661, 713
Self-service protocol, 626
Semaphore, 345, 364, 545, 551, 556, 561, 628, 637, 663
  binary, 549, 637–640, 661–663
  counting, 549, 551, 634, 638, 640
Semaphored kernel, 682
Sender-synchronized protocol, 364
Sequent Computer Systems
  Balance 21000, 701
  Symmetry, 355
  Symmetry S81, 24, 90, 95
  Symmetry series, 118, 431
Sequential bottleneck, 112, 130, 131
Sequential buffer, 283
Sequential computer, 9, 17
Sequential consistency (SC), 248, 251–256, 258, 479, 487–489, 523, 551
Sequential environment, 17
Sequin, C., 208
Serial complexity, 34
Serial fraction, 112
Serialization principle, 345
Serially reusable code, 629
Server synchronization, 627, 628
Session control, 705
Session leader, 705
Set-associative mapping, *see* Cache
Sevcik, K., 96
Shadow object, 698, 700
Shang, S., 359, 365, 366, 394
Shapiro, E., 612
Shar, L. E., 277, 322, 323
Shared distributed memory, 122, 136, 149
Shared memory, 6, 18–24, 27, 35, 37, 213, 228, 238, 248, 253, 258, 262, 331, 335, 341, 345, 348, 349, 351, 353–355, 365, 368, 395, 399
Shared variable, 11, 13, 19, 345, 348, 398, 548, 549, 551, 559, 561
Shared virtual memory (SVM), 6, 148, 370, 372, 476, 671, 686, 711
Shared-memory multiprocessor, *see* Multiprocessor
Shared-variable communication, 548
Shared-variable programming model, 547–551
Sharing list, 483–486
Sheperdson, J. C., 35, 46
Shih, Y. L., 394
Shimada, T., 45
Shiva, 478, 479
Side-effect dependence, 626
Siegel, H. J., 31, 46, 96
Siewiorek, D. P., 469
Silicon Graphics, Inc.
  4-D, 226, 479, 517
Simple cycle, 276, 324, 325, 327
Simple operation latency, 159, 165, 166, 178, 208, 309, 317, 321
Sindhu, P. S., 248–256
Single floating master, 674
Single index variable (SIV), 571, 574–576, 578
Single instruction stream over multiple data streams (SIMD), 11, 27, 30–33, 37, 43, 48, 59, 63, 77, 80, 91, 120–122, 143, 183, 186, 368, 399, 403, 446–457, 505, 539, 547, 554–556, 561, 614, 617, 625
Single instruction stream over single data stream (SISD), 11, 49
Single master kernel, 672
Single program over multiple data streams (SPMD), 62, 63, 120, 368, 399, 504, 524, 539, 541, 554, 556, 562, 614, 649
Single-assignment language, 559
Single-error correction/double-error detection (SECDED), 421
Single-stage network, 336, 338
SISAL, 559, 612
Slave processor, 162
Sleep-wait protocol, *see* Wait protocol
Slotted token ring, 500
Small computer systems interface (SCSI), 334, 393
Smith, A. J., 238, 256
Smith, B., 713
Smith, B. J., 44, 516, 525, 539
Smith, B. T., 661
Smith, J. E., 281, 289–291, 322, 415, 469
Snir, M, 36, 46
Snoopy bus, 351–355
Snoopy cache-coherence protocol, 351, 355–358, 396, 482, 519
Soft atom, 625, 627

Copyrighted material

Answers to Selected Problems

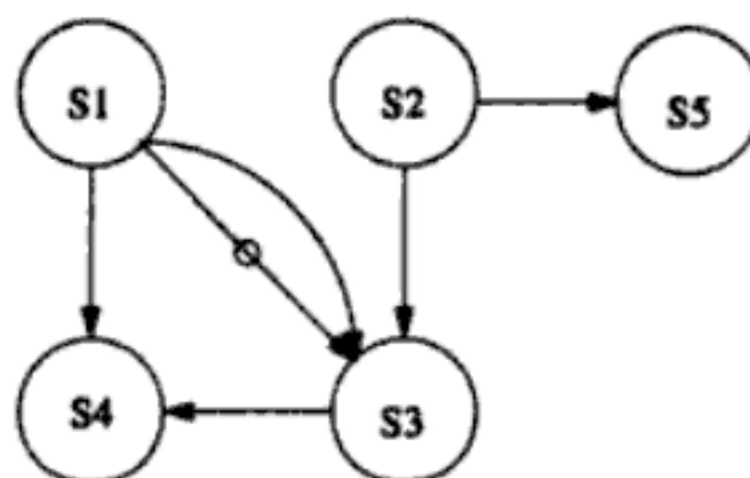
Provided below are brief or partial answers to a few selected exercise problems. These answers are meant for readers to verify the correctness of their answers. Derivations or detailed computational steps in obtaining these answers are left for readers.

Problem 1.1 Average CPI=1.55 cycles per instruction. Effective processor performance = 25.8 MIPS. Execution time = 3.87 ns.

Problem 1.4 (a) Average CPI = 2.24. (b) MIPS rate = 17.86

Problem 1.8 (a) Sequential execution time = 1664 CPU cycles. (b) SIMD execution time = 26 machine cycles. (c) Speedup factor = 64.

Problem 2.5
(a)



(b) S_4 and S_5 need to use the same Store Unit in accessing the memory. Therefore they are potentially storage-dependent.

Copyrighted material

Kai Hwang has introduced the issues in designing and using high performance parallel computers at a time when a plethora of scalable computers utilizing commodity microprocessors offer higher peak performance than traditional vector supercomputers.... The book presents a balanced treatment of the theory, technology architecture, and software of advanced computer systems. The emphasis on parallelism, scalability, and programmability makes this book rather unique and educational. I highly recommend Dr. Hwang's timely book. I believe it will benefit many readers and be a fine reference.

C. Gordon Bell

This book offers state-of-the-art principles and techniques for designing and programming parallel, vector, and scalable computer systems. Written by a leading expert in the field, the authoritative text covers:

- **Theory of parallelism** — Parallel computer models, program and network properties, performance laws, and scalability analysis.

- **Advanced computer technology** — RISC, CISC, Superscalar, VLIW, and superpipelined processors, cache coherence, memory hierarchy, advanced pipelining, and system interconnects.

- **Parallel and scalable architectures** — Multiprocessors, multicomputers, multivector and SIMD computers, and scalable, multithreaded, and dataflow architectures.

- **Software for parallel programming** — Parallel models, languages, compilers, message passing, program development, synchronization, parallel UNIX extensions, and heterogeneous programming.

- **Illustrative examples and problems** — Over 100 examples with solutions, 300 illustrations, and 200 homework problems involving designs, proofs, and analysis. Answers to selected problems are given. Solutions Manual available to instructors.

- **Case studies of real systems** — Industrial computers from Cray, Intel, TMC, Fujitsu, NEC, Hitachi, IBM, DEC, MasPar, nCUBE, BBN, KSR, Tera, and Stardent, and experimental systems from Stanford, MIT, Caltech, Illinois, Wisconsin, USC, and ETL in Japan.

*The **McGraw-Hill** Companies*

Tata McGraw-Hill
Publishing Company Limited
7 West Patel Nagar, New Delhi 110 008

Visit our website at : www.tatamcgrawhill.com

ISBN-13: 978-0-07-053070-6
ISBN-10: 0-07-053070-X

Copyrighted material