FUNDAMENTALS of PARALLEL COMPUTER ARCHITECTURE

Multichip and Multicore Systems

Fundamentals of Parallel Computer Architecture

Multichip and Multicore Systems

Yan Solihin

North Carolina State University

Solihin Books

Contents

Preface vii

1 Perspectives 1

1.1 Evolution of Parallel Computers 4
1.2 Definition of Parallel Computers 9
1.3 Why Parallel Computers? 10
1.4 Flynn's Taxonomy of Parallel Computers 11
1.4.1 Classes of MIMD Parallel Computers 13
1.5 Examples of High-End Parallel Computers 14

I Shared Memory Parallel Programming 17

2 Parallel Programming Models 19

2.1 Overview 19
2.2 Comparing Shared Memory and Message Passing Models 22
2.3 A Simple Example 24

3 Shared Memory Parallel Programming 29

3.1 Steps in Parallel Programming 30
3.2 Identifying Code and Loop-Level Parallelism 32
3.2.1 Iteration-space Traversal Graph and Loop-Carried Dependence Graph 34
3.2.2 Finding Parallel Tasks Across Iterations 35
3.2.3 DOACROSS Parallelism 40
3.2.4 Parallelism Across Statements in a Loop 43
3.2.5 DOPIPE Parallelism Across Statements in a Loop 45

3.3 Identifying Algorithm-Level Parallelism 48
3.4 Determining the Scope of Variables 51
3.4.1 Privatization 53
3.4.2 Reduction Variables and Operation 55
3.4.3 Summary of Criteria 56
3.5 Synchronization 57
3.6 Mapping Tasks to Processors and Memory 59
3.7 A Brief Introduction to OpenMP 60
3.8 Example: Matrix Multiplication Parallelization 64
3.9 Exercise Problems 68

4 Issues in Shared Memory Programming 75
4.1 Common Correctness Pitfalls 76
4.1.1 Result Preservation 76
4.1.2 Incorrect or Missing Synchronization 78
4.1.3 Incorrect Variable Scope 80
4.2 Parallelizing Compiler Limitations 81
4.2.1 A Case Study on the MIPSpro Compiler 81
4.3 Performance Considerations 87
4.3.1 Amdahl's Law 87
4.3.2 Parallel Thread Granularity 90
4.3.3 Synchronization Granularity 91
4.3.4 Inherent and Artifactual Communication 92
4.3.5 Scheduling and Load Balancing 96
4.3.6 Memory Hierarchy Considerations 99
4.4 Exercise Problems 106

5 Parallel Programming for Linked Data Structures 113
5.1 Parallelization Challenges in LDS 114
5.1.1 Loop-Level Parallelization is Insufficient 115
5.1.2 Approaches to Parallelization of LDS 117
5.1.3 Challenges in LDS Parallelization 117
5.2 Parallelization Techniques for Linked Lists 126
5.2.1 Parallelization Among Readers 126
5.2.2 Global Lock Approach 127
5.2.3 Fine-Grain Lock Approach 131
5.3 Exercise Problems 136

II Designing Parallel Computer Architectures 139

6 Introduction to Memory Hierarchy Organization 141
6.1 Motivation for Memory Hierarchy 142
6.2 Basic Architectures of a Cache 144
6.2.1 Placement Policy 144
6.2.2 Replacement Policy 149
6.2.3 Write Policy 151
6.2.4 Inclusion Policy on Multi-Level Caches 153
6.2.5 Split Cache Organization 155
6.2.6 Translation Lookaside Buffer and Cache Addressing . . . 155
6.2.7 Non-Blocking Cache 158
6.3 Cache Performance 159
6.3.1 The Power Law of Cache Misses 162
6.3.2 Stack Distance Profile 164
6.3.3 Cache Performance Metrics 165
6.4 Prefetching 168
6.4.1 Stride and Sequential Prefetching 170
6.4.2 Prefetching in Multiprocessor Systems 172
6.5 Exercise Problems 174

7 Introduction to Shared Memory Multiprocessors 179
7.1 The Cache Coherence Problem 181
7.2 Memory Consistency Problem 184
7.3 Synchronization Problem 186
7.4 Exercise Problems 192

8 Bus-Based Coherent Multiprocessors 195
8.1 Overview 196
8.1.1 Basic Support for Bus-Based Multiprocessors 199
8.2 Cache Coherence in Bus-Based Multiprocessors 203
8.2.1 Coherence Protocol for Write-Through Caches 203
8.2.2 MSI Protocol with Write Back Caches 206
8.2.3 MESI Protocol with Write Back Caches 215
8.2.4 MOESI Protocol with Write Back Caches 222
8.2.5 Update-Based Protocol with Write Back Caches 229
8.3 Impact of Cache Design on Cache Coherence Performance 235

8.4 Performance and Other Practical Issues 236
8.4.1 Prefetching and Coherence Misses 236
8.4.2 Multi-Level Caches 237
8.4.3 Snoop Filtering 238
8.5 Exercise Problems 240

9 Hardware Support for Synchronization 247
9.1 Lock Implementations 248
9.1.1 Evaluating the Performance of Lock Implementations . . . 248
9.1.2 The Need for Atomic Instructions 249
9.1.3 Test and Set Lock 252
9.1.4 Test and Test and Set Lock 255
9.1.5 Load Linked and Store Conditional Lock 257
9.1.6 Ticket Lock 262
9.1.7 Array-Based Queuing Lock 265
9.1.8 Qualitative Comparison of Lock Implementations 267
9.2 Barrier Implementations 269
9.2.1 Sense-Reversal Centralized Barrier 270
9.2.2 Combining Tree Barrier 274
9.2.3 Hardware Barrier Implementation 274
9.3 Exercise Problems 277

10 Memory Consistency Models 283
10.1 Programmers' Intuition 284
10.2 Architecture Mechanisms for Ensuring Sequential Consistency . . 290
10.2.1 Basic SC Implementation on a Bus-Based Multiprocessor 290
10.2.2 Techniques to Improve SC Performance 293
10.3 Relaxed Consistency Models 296
10.3.1 Safety Net 297
10.3.2 Processor Consistency 297
10.3.3 Weak Ordering 299
10.3.4 Release Consistency 302
10.3.5 Lazy Release Consistency 307
10.4 Synchronization in Different Memory Consistency Models 309
10.5 Exercise Problems 314

11 Distributed Shared Memory Multiprocessors **319**
11.1 Approaches to Large Scale Multiprocessors 320
11.2 Building a Directory-Based Coherence Protocol 322
11.2.1 Directory Format 327
11.3 Basic DSM Cache Coherence Protocol 332
11.4 Implementation Correctness and Performance 338
11.4.1 Handling Races due to Out-of-Sync Directory State 338
11.4.2 Handling Races due to Non-Instantaneous Processing of a Request 342
11.4.3 Write Propagation and Serialization 352
11.4.4 Synchronization Support 354
11.4.5 Memory Consistency Models 356
11.5 Exercise Problems 357

12 Interconnection Network Architecture **361**
12.1 Link and Channel 363
12.2 Network Topology 365
12.3 Routing Policies and Algorithms 371
12.4 Router Architecture 378
12.5 Exercise Problems 381

III Designing Multicore Architectures **385**

13 Multicore Architecture: Overview and Challenges **387**
13.1 Measuring Aggregate Multicore Performance 389
13.2 Unique Challenges in Multicore Architectures 394
13.2.1 Cache Organization 395
13.2.2 Performance Volatility 397
13.2.3 Scalability Problems 399
13.3 Exercise Problems 401

14 Multicore Memory Hierarchy Organization **403**
14.1 Physical Cache Organization 405
14.1.1 Physically Shared Cache 405
14.1.2 Physically Private Caches 409
14.1.3 Hybrid Private+Shared Caches 409
14.2 Logical Cache Organization 413

14.2.1 Mapping Function 419
14.2.2 Improving Distance Locality of Shared Cache 421
14.2.3 Capacity Sharing in the Private Cache Configuration . . . 422
14.3 Keeping Cache Tiles Coherent 426
14.3.1 Broadcast/Snooping Coherence Protocol on Multicore . . 427
14.3.2 Directory-Based Coherence Protocol 434
14.4 Exercise Problems 438

15 Performance Volatility **443**
15.1 Empirical Observation 444
15.2 Factors Causing Performance Volatility 456
15.2.1 L2 Cache Space Contention 456
15.3 Dealing with Performance Volatility 458
15.3.1 Cache Partitioning 458
15.3.2 Off-chip Bandwidth Partitioning 461
15.3.3 Interaction Between Cache and Bandwidth Partitioning . . 465
15.3.4 System Performance versus Performance Volatility 468
15.4 Exercise Problems 469

16 Bandwidth Wall Bottleneck **471**
16.1 Assumptions, Scope, and Limitations of Study 473
16.2 Modeling CMP Memory Traffic 474
16.3 Bandwidth Wall Impact on CMP Scaling 476
16.3.1 Bandwidth Wall Impact on CMP Scaling 476
16.3.2 CMP Scaling and Die Area Allocation 479
16.4 Analyzing Memory Traffic Reduction Techniques 480
16.4.1 Indirect Techniques 482
16.4.2 Direct Techniques 489
16.4.3 Dual Techniques 491
16.4.4 Discussion and Recommendations 495
16.5 Concluding Remarks 500
16.6 Exercise Problems 502

List of Abbreviations **513**

Preface

The world of parallel computers is undergoing a significant change. Parallel computers started as high end supercomputing systems mainly used for scientific computation. Recently, the trend towards multicore design has enabled the implementation of a parallel computer on a single chip. Parallel computers are now ubiquitous, present in all mainstream architectures for servers, desktops, and embedded systems. Single-chip parallel computers now co-exist with larger parallel computer architectures in servers and supercomputers. The applications that run on parallel computers are increasingly becoming more diversed, including business applications (data mining, transaction processing, financial modeling, etc.), scientific applications (physical and biological simulation, weather modeling, etc.), as well as entertainment applications (game playing, computer graphics, animation, etc.). These trends coincide with the slowing of performance growth of a single processor. In the past, programmers could rely on the processor becoming faster to enable more complex software to be built. Those days are gone. Now, programmers must exploit parallelism in order to obtain better performance for their code. All these changes are exciting, but at the same time they bring serious challenges in how to design parallel architectures that give programmers what they need.

The purpose of this book is to bring together concepts, technologies, and implementations of parallel computer architectures in a package that is suitable for use as an introductory graduate text. The following philosophies were used when writing this book.

First, the book focuses on introducing basic concepts in programming and architectures of shared memory multiprocessors to starting graduate students. The approach is to start with an intuition leading to a concept before introducing the concept. As a consequence, while to some the writing may appear a bit verbose, beginning students may appreciate the intuition and clarity.

In addition, the book is purposely designed to be relatively short for several reasons. We want to avoid a situation in which students are overwhelmed with the length of each topic that they are discouraged from reading the book in its entirety. Each chapter in the book is designed to be short enough for students to read in a relatively short period of time. We hope that students will be encouraged to read the entire book, and even if they skip parts of the books, we hope that they will skip entire chapters rather than parts of a chapter. To reach the length goal, it is necessary to leave out some contents. For example, advanced concepts that are used mainly for research-level investigations are left out. Details that are unnecessary for understanding the basic concepts are also left out. As students become more proficient with parallel architectures, they are encouraged to learn more advanced concepts from research papers, white papers, and other publications.

Organization of this Book

The book is organized into three parts. The first part deals with programming issues in shared memory multiprocessors, such as the programming model, techniques to parallelize regular and irregular applications. The second part deals with the architectures for shared memory multiprocessors: introduction to memory hierarchy, basic problems, cache coherence, memory consistency, synchronization, and various classes of shared memory multiprocessors. The third part deals specifically with multicore architectures: multicore-specific challenges, cache organization choices in a multicore architecture, performance volatility, and scalability in the presence of bandwidth wall.

Acknowledgment

The author thanks the students from the Fall 2006 course CSC/ECE 506 for motivating the author to write this textbook. The author is greatly indebted to the following reviewers who have helped improved the organization and writing of the book: Anil Krishna, Laxminarayan Chavani, Mohit Ghambir, Abhik Sarkar, Xiaowei Jiang, Fang Liu, Siddhartha Chhabra, Brian Rogers, Devesh Tiwari, and Ed Gehringer. The author thanks the students of CSC/ECE 506 in Spring 2008 for giving comments for early drafts of this book. The authors would like to acknowledge the contribution of experimental results from studies by Fang Liu, Ahmad Samih, Brian Rogers, Xiaowei Jiang, Sharad Bade, and Asaf Ebgi.

Chapter 1

Perspectives

Contents

1.1 Evolution of Parallel Computers **4**
1.2 Definition of Parallel Computers **9**
1.3 Why Parallel Computers? **10**
1.4 Flynn's Taxonomy of Parallel Computers **11**
1.4.1 Classes of MIMD Parallel Computers 13
1.5 Examples of High-End Parallel Computers **14**

Throughout the history of computer systems, parallel computers have been an important class of computers. A parallel computer combines a large number of processing elements (CPUs) into a single system, allowing a large computation to be carried out in orders of magnitude shorter execution time. Scientists and engineers have relied on parallel computers to solve important scientific questions by running simulations on them. What is at stake is the pace of scientific discovery and engineering innovations. Over time, parallel computers have found a broader audience. For example, corporations rely on mining data from a large collection of databases, which is a very computation intensive process. Businesses rely on processing transactions on powerful parallel computers, while internet search engine providers use parallel computers to rank web pages and evaluate their relevance based on a search criteria. Gamers demand games that show more realistic simulation of physical phenomena and realistic rendering of three-dimensional images. In short, there have been needs for parallel computers, and there will be even more needs for them in the future.

An interesting development recently has been the evolution of parallel computer architectures from the architecture of big, powerful, and expensive computer systems to the mainstream architecture of servers, desktops, and embedded systems. Before 2001, parallel computers were mainly used in the server and supercomputer markets. Client machines (desktops, laptops, and mobile devices) were single-processor systems. For various reasons that will be discussed later, since 2001 they have been evolving into an architecture in which multiple processor cores are implemented in a single chip. Such an architecture is popularly known as the *multicore* architecture. With the adoption of multicore architectures, most of computer systems today are parallel computers.

One of the enabling trends for the move to multicore architectures has been the increasing miniaturization of transistors, through which more and more transistors can be packed in a single chip. The pace for this transistor integration has been staggering. For decades, it has faithfully followed a prediction made by Intel co-founder Gordon Moore in 1965 that the number of transistors that can be manufactured inexpensively in a single Integrated Circuit (IC) doubles every two years [1]. This staggering growth has resulted at first in fitting of more and more components of a single processor on a chip, followed by adding more features to a single processor, and more recently has resulted in replicating processor cores in one chip.

For a historical look at how fast transistor integration changes processor architectures, Table 1.1 shows some of the Intel processors developed from 1971 to 2006. From 1971 to 2006, the number of transistors that fit on a chip has grown from 2,300 to 167 million, a $72,608\times$ increase. The clock frequency of the processor has also grown from 750KHz to 2.4 GHz, a $3,200\times$ increase. The width of the processor has increased from 8 bits to 64 bits. What a spectacular growth! At first, transistor integration was used to move components of a single processor that did not fit on chip into the chip. For example, Intel 486 integrated the floating point unit (FPU) on chip. Previously, floating point computation was performed by a co-processor. Then, as more and more transistors could be integrated on a chip, the processor was augmented with features that exploited parallelism at the instruction level, such as out-of-order execution (in the Intel Pentium Pro), and dynamic branch prediction (in the Intel Pentium MMX). Since the speed of main memory was not keeping up with the growth of speed of the processor, it

[1] A more recent version states that the doubling is every 18 months rather than every two years, but the original prediction by Moore was every two years.

was necessary to introduce memory hierarchy, where a smaller memory was integrated on chip for faster data access. It started with the integration of first level (L1) data and instruction caches in Intel Pentium, and a second level (L2) cache in Intel Pentium III. Gradually, however, there was a diminishing return on the performance obtained from a single core, causing the design to shift into adding parallelism between threads of execution, which started from *simultaneous multithreading* (SMT) in Intel Pentium III, and dual processor cores in Intel Xeon. If transistor integration continues as expected in the next several years, it is likely that the processor chip will integrate more processor cores, larger caches and/or deeper cache hierarchy, and other components such as the memory controller and a graphic processor.

■ *Did you know?*

From 1971 to 2000 the number of transistors on a chip has increased by a factor of 18,260. The clock frequency has increased by a factor of 18,519. This represents an annual growth rate of 40%. Gordon Moore once joked, "If the automobile industry advanced as rapidly as the semiconductor industry, a Rolls Royce would get a half million miles per gallon and it would be cheaper to throw it away than to park it."

Of course, any trend that is exponential will likely end at some point. There are signs that it is becoming more difficult to keep Moore's law going. Restrictions in power consumption and circuit reliability will eventually slow down transistor integration. However, predicting when Moore's law will no longer hold is very difficult since researchers keep improving the process technology. However, one important observation that can be made right now is performance scaling has started slowing down even as transistor integration speed has not. For example, clock frequency, long associated with performance scaling and Moore's law, has significantly slowed. One major reason for this is power efficiency. While power consumption scales linearly as frequency is increased, deepening the processor pipeline to provide the frequency increase incurs diminishing return on performance. Hence, power efficiency decreases.

In the future, even when scaling down transistor size continues at a rapid pace, transistors are becoming so small that even a small deviation in the manufacturing process can result in unreliable transistors. Unreliable transistors can be remedied by introducing redundancy at various levels (e.g. Error Correcting Code/ECC at the caches), but redundancy comes at the expense of performance. An increasing portion of the transistors will be used to ensure reliable operation of the system, resulting in slowing the performance growth.

Note that the integration of two cores on a chip marked a new milestone in processor design. Parallel computers are now integrated on a single chip. IBM has 2-core Power processors, Intel has dual- and quad-core Xeon processors, AMD with its dual- and quad-core Opteron processors, and Sun with 8-core Niagara processors. Such a milestone was the turning point for parallel computer architecture. Previously in order to have more than one processor in a system, users had to buy an expensive system that has a motherboard supporting more than one processor chips. Now, a single processor chip has multiple processor cores. Virtually all new laptops, desktops, servers are now parallel computers.

Table 1.1: Evolution of Intel Processors. Other chipmakers also enjoy a similar transistor integration progress.

From	Procs	Specifications	New Features
1971	4004	740 KHz, 2300 transistors, 10μm, 640B addressable memory, 4KB program memory	
1978	8086	16-bit, 5–10MHz, 29,000 transistors at 3μm, 1MB addressable memory	
1982	80286	8–12.5MHz,	Virtual memory and protection mode
1985	386	32-bit, 16-33MHz, 275K transistors, 4GB addressable memory	Pipelining
1989	486	25-100MHz, 1.2M transistors,	FPU integration
1993	Pentium	60–200MHz,	on chip L1 caches, SMP support
1995	Pentium Pro	16KB L1 caches, 5.5M transistors,	out-of-order execution
1997	Pentium MMX	233-450MHz, 32KB L1 cache, 4.5M transistors	dynamic branch prediction, MMX instruction sets
1999	Pentium III	450-1400MHz, 256KB L2 cache on chip, 28M transistors	SSE instruction sets
2000	Pentium IV	1.4-3GHz, 55M transistors	hyper-pipelining, SMT
2006	Xeon	64-bit, 2GHz, 167M transistors, 4MB L2 cache on chip	Dual core, virtualization support

1.1 Evolution of Parallel Computers

When looking at the history of parallel computers, we must keep in mind how Moore's law has changed processor architectures. Parallel architectures were initially a natural idea because there were not enough transistors on a chip to implement a complete microprocessor. Hence, it was natural to have multiple chips that communicated with each other, either when those chips implemented differ-

ent components of a processor or when they implemented components of different processors. At that time, all levels of parallelism were considered in parallel computer architectures: instruction level parallelism, data parallelism, and task parallelism. Over time, it became clear which types of parallelism were more appropriate for implementation across processors or within a single processor. For example, instruction level parallelism is now implemented in the architecture of a single processor because it requires register-level communication among instructions, which can only be supported with a low latency on a single processor.

As transistor integration continued, an entire microprocessor fit into a single chip (e.g., Intel 4004 in 1971). Since then, the performance of a single microprocessor increased rapidly, pretty much at Moore's law speed. Such a rapid performance growth was both fueled by transistor miniaturization which drove up the clock frequency of the processor, as well as two classes of architecture techniques: *instruction-level parallelism* (ILP) and *cache memory*. Gains from these architecture techniques were so significant that uniprocessor systems could catch up the performance of parallel computers in a few years, while costing a tiny fraction of parallel computers. It was hard to justify purchasing expensive parallel computers unless there was an immediate demand for intensive computation, such as in the supercomputer market.

To give an illustration, suppose that a 100-processor system was produced, and it could achieve a perfect speedup of $100\times$ for an application under consideration. That is, if the same application was run on a single processor system, it would be $100\times$ slower. However, when a single processor's speed grew at Moore's law (doubling every 18 months, or 60% annual growth), one year later the application was only $\frac{100}{1.6} = 62.5\times$ slower on the latest single processor system, and two years later the application was only $\frac{100}{1.6^2} = 39\times$ slower on the latest single processor system, and so on. Ten years later, running the application on the latest single processor system yielded the same speed as in the 10-year old 100-processor system. It was difficult to justify purchasing a large and highly expensive parallel computer if its performance would be eclipsed by a single processor system in just a few years. Few companies could find economic justification for such a purchase, except a few large companies and national laboratories at the cutting edge of scientific discovery. This example used optimistic assumptions – using pessimistic assumptions would reduce the advantage of parallel computers over a single-processor system. For example, on a 100-processor system, the realizable speedup for most applications is often much less than 100. Second, even

for applications that can reach a 100× speedup, it often took weeks to months of highly-manual parallel performance tuning to extract the speedup, while running the application on a single processor system often required simple tuning effort. Third, the market for highly parallel systems were small so they could not benefit from pricing advantage of high-volume products. As a result, a parallel machine might cost more than 1000× the cost of a single processor machine. Finally, designing parallel computer architectures was a highly complex task and the logic for interconnecting processors to form a tightly-coupled parallel computer was not simple. This caused a significant extra development time that when a parallel computer was launched, the processors it was composed of were often already much slower than the latest single-processor system that was already available. Overall, the combination of all these factors made it hard for parallel computers to be a commercial success.

Let us go back to the discussion of classes of techniques that enabled a single processor system performance to grow fast. Instruction-level parallelism refers to the execution of multiple instructions in parallel. One major innovation was a simplified instruction set commonly known as *Reduced Instruction Set Computer* (RISC). In contrast to *Complex Instruction Set Computer* (CISC), RISC achieves the same computation task with a larger number of instructions. However, the instructions in RISC were amenable to *pipelining*. With pipelining, while the first instruction is decoded, the second instruction is fetched; while the first instruction is executed, the second instruction is decoded, and the third instruction is fetched; and so on. Hence, several instructions can be processed simultaneously at different stages in the processor. Later on, a *superscalar* architecture which widens the pipeline so that multiple *independent* instructions can be processed simultaneously at the same pipeline stage, gave further boost to uniprocessor system performance. In addition, *out-of-order* execution helped to improve the performance of superscalar architectures by letting younger (independent) instructions to execute even though older instructions stall in the pipeline.

Cache memory is a temporary storage built from *static random access memory* (SRAM) to keep data that is recently accessed. Because a program typically exhibits *temporal locality* (it tends to access data or instruction that it recently accessed) and *spatial locality* (it tends to access data or instruction that neighbored another that it recently accessed), cache memory works very well in providing data and instruction to the processor core most of the time. Take for example the code shown in Code 1.1. The variable `sum`, `i`, and `N` are accessed at each loop

iteration, exhibiting temporal locality. The elements in array `arr[]` are accessed consecutively in consecutive iterations, exhibiting spatial locality. Programmers, attempting to better organize their code and make it more modular, naturally introduce temporal locality. Spatial locality is also often naturally present because of the way compiler lays out data structures in memory.

Code 1.1 A simple code summing up elements of an array.

```
1  ...
2  for (i=0; i<N; i++)
3    sum = sum + arr[i];
4  ...
```

Deepening the processor pipeline (which allows a huge increase in clock frequency), widening a superscalar architecture (which allows more instructions to be processed per clock cycle), and adding to cache capacity and number of cache levels were responsible for the huge and persistent performance improvement of uniprocessor systems in the 80s and 90s. All these improvements were achieved at the hardware level and were largely transparent to programmers.

In contrast, parallel architectures require parallel programming and tuning, which is not transparent to programmers. Programmers often had to spend much time and effort in porting their code into parallel computers, and when a parallel program is produced, often a significant tuning effort was needed to make it scalable. The tuning effort in general grows with the size of the machine and with the complexity of the memory hierarchy of the machine. Furthermore, ILP, which was initially a technique for a parallel computer, turned out to require register-level communication that was not feasible to implement across multiple processors. Cache memory also favored uniprocessor architecture as most of the time data can be found and read in a few cycles on a single processor. With parallel architectures, data often needs to be loaded from another processor on a separate chip. Hence, the development of ILP and cache memory technology did not benefit parallel architectures much, and parallel architectures were mainly relegated to a niche of expensive servers and supercomputers.

Low-cost distributed computers were being developed in the 90s by assembling many uniprocessor systems with an off-the-shelf network. This gave birth to *network of workstations*, which was later more commonly referred to as *clusters*. Compared to parallel computers, distributed computers were a lot cheaper, but had a high communication latency between processors. However, some classes

of applications did not have much inter-processor communication, and were quite scalable when they ran on clusters.

Things started to change in the 90s. ILP started to become harder and costlier to extract. It was hard to improve a branch predictor that was already 90% accurate, for example. Deepening the processor pipeline boosted the clock frequency of a processor, but did not change the length of critical dependence loops between instructions. Hence, at some point there was a diminishing return on breaking a pipeline stage into smaller ones. Also, widening the processor pipeline introduced quadratic increase in complexity because it needs to perform dependence checking between instructions that are decoded in the same cycle. Caches were still effective, but also showed diminishing return in which doubling the cache space reduces its miss rate by less than a half. Power consumption of the processor chip also increased, which required a fan, larger heat sink, etc. Overall, the inevitable diminishing return in ILP was becoming a reality. On the other hand, Moore's law continued on, leaving a big question of what is the most *power-performance* efficient way to use the extra transistors available. Chipmakers had no choice but to abandon the diminishing return and power inefficiency of ILP in favor of implementing multiple processors on a chip. This new architecture is referred to as *Chip Multi-Processors* (CMP) or more popularly *multicore* architectures.

As of the time of writing of this book (January 2008), the trend continued towards more and more processors being integrated in a single chip. This makes parallel computers ubiquitous in the servers, desktops, and even mobile systems. Some chips adopt the approach of having fewer powerful processor cores such as Intel Core Duo, while others adopt the approach of having many simple processor cores, such as Sun Niagara.

Table 1.2 shows some examples of recent multicore systems at the time of this writing (OOO stands for out-of-order).

Table 1.2: Examples of current multicore processors.

Aspects	**Intel Clovertown**	**AMD Barcelona**	**IBM Cell**
# Cores	4	4	8+1
Clock Freq	2.66GHz	2.3GHz	3.2GHz
Clock type	OOO Superscalar	OOO Superscalar	2-issue SIMD
Caches	32KB L1, 2×4MB L2 (shared by 2)	64KB L1, 512KB L2 (per core), 2MB L3 (shared)	256KB local store (per core)
Chip power	120 Watts	95 Watts	100 Watts

In addition to parallel computers on a chip (multicore), larger parallel com-

puters can also be constructed using multicore processors as building blocks. An example is the SGI Altix 4700 system which integrates up to 512 sockets of dual-core Intel Itanium 2 processors (for a total of 1024 cores) as a single machine – with a single Linux operating system (OS) image.

1.2 Definition of Parallel Computers

Almasi and Gottlieb [Almasi and Gottlieb, 1989] defined parallel computers as:

> "*A parallel computer is a collection of processing elements that communicate and cooperate to solve a large problem fast.*"

While the definition appears straightforward, there is a broad range of architectures that fit the definition. For example, take the phrase "*collection of processing elements*". What constitutes a processing element? A processing element can be a functional unit, a thread context on a processor, a processor core, a processor chip, or an entire node (processors in a node, local memory, and disk). More recently, the de facto processing elements are processors, hence parallel computers are also referred to as *multiprocessors*. When the processing elements are processor cores on a single chip, the system is referred to as a *multicore* system. In the rest of the book, we will use the term multiprocessors to refer to a collection of processors regardless of whether the processors are implemented in different chips or a single chip, and multicore to refer specifically to multiple processors implemented on a single chip.

There is also a choice of a high number of simple processing elements versus a few powerful processing elements. Such a choice has significant implications on the scalability of the systems and how fast sequential programs can run.

The term "*communicate*" refers to the processing elements sending data to each other. The choice of communication mechanisms determine two important classes of parallel architectures: *shared memory* systems in which parallel tasks that run on the processing elements communicate by reading and writing to a common memory location, or *message passing* systems in which all data is local and parallel tasks must send explicit messages to each other to pass data. The communication medium such as what interconnect network is used to connect the processing elements is also an important issue that determines communication latency, throughput, scalability, and fault tolerance.

The term "*cooperate*" refers to the synchronization of the progress of execution of a parallel task relative to other tasks. *Synchronization* allows sequencing of

operations to ensure correctness. One important issue is synchronization granularity, which implies how often tasks need to synchronize with each other, affecting scalability and load balance properties.

The phrase "*solve a large problem fast*" indicates a choice of using a general versus special purpose machine. Even a general purpose machine can be designed and tuned with specific computation in mind, which makes it fast and scalable for such computation but may be slow for other types of computation.

The purpose of the discussion above is to give readers a glimpse into the diversity of the design space of parallel computer architectures.

1.3 Why Parallel Computers?

What motivates people to use parallel computers? One obvious reason is the *absolute performance* that a parallel computer provides compared to that provided by a single (uni) processor system. For example, the simulation of a single protein folding can take years on the most advanced processor, but it only takes days to perform on a large and powerful multiprocessor. The pace of scientific discovery lies on how fast a particular computation problem can be solved on the computer. In addition, some applications have a real time constraint, which necessitates the computation to be fast enough to be useful. For example, weather modeling for predicting the precise direction of hurricanes and tornadoes must be performed in a timely manner, otherwise the prediction is not useful.

Another reason to use a multiprocessor or multicore system is that it is more attractive in terms of *cost-adjusted performance*, or *power-adjusted performance*. For a small-sized machines consisting of a single processor chip, the industries' shift to the multicore architecture is motivated partly by the superior power-adjusted performance of multicores compared to a single core system. That is, for the same amount of power consumption, a multicore system achieves a higher total performance compared to a single processor system. For medium-sized machines consisting of between 4 and 16 processor chips, the cost of making such a multiprocessor is relatively small compared to having 4 or 16 independent systems, especially with regard to the cost of software, and maintenance of the systems. However, large-scale multiprocessors rarely achieve low cost-adjusted or power-adjusted performance because of the high cost of providing a very high scalability. Nevertheless, such systems are still used for computation that requires absolute performance.

1.4 Flynn's Taxonomy of Parallel Computers

Flynn defined a taxonomy of parallel computers [Flynn, 1972] based on the number of *instruction streams* and *data streams*, as shown in Table 1.3, and illustrated in Figure 1.1. An instruction stream is a sequence of instructions followed from a single program counter. A data stream is an address in memory which the instruction operates on. A control unit fetches instructions from a single program counter, decodes them, and issues them to the processing element. The processing element is assumed to be a functional unit. Instruction and data are both supplied from the memory.

Table 1.3: Flynn's Taxonomy of Parallel Computers

		Number of Data Streams	
		Single	Multiple
Number of Instruction Streams	Single	***SISD***	***SIMD***
	Multiple	***MISD***	***MIMD***

SISD is not considered a parallel architecture as it only has one instruction and one data stream. However, SISD exploits parallelism at the instruction level. Even though there is only one stream of instructions, parallelism between the instructions from the stream can be exploited when the instructions are independent from one another. Processor cores today are an example of SISD machine that exploits instruction-level parallelism.

SIMD is a parallel architecture in which a single instruction operates on multiple data. An example of SIMD architectures can be found in vector processors. SIMD-style extension has also been added to many processors, for example Intel's MMX instructions. As a simple illustration, consider multiplying a scalar `a` and an array `X`. With SISD, we need to execute a loop where in each iteration, we perform a multiplication between `a` and one element from array `X`. With SIMD, the entire operation can be performed with one scalar-vector multiply instruction without the use of a loop. SIMD is known for its efficiency in terms of the instruction count needed to perform a computation task. Using a loop to perform a single operation on multiple data is inefficient: one instruction is needed for

Figure 1.1: Illustration of Flynn's taxonomy of parallel computers.

each data item, and there are loop overheads such as loop index computation and conditional branches. In a SIMD architecture, sometimes only one instruction is needed to operate on a large data.

MISD is an architecture in which multiple processing elements execute from different instruction streams, and data is passed from one processing element to the next. An example of this machine is the *systolic array*, such as the CMU iWarp [Borkar et al., 1990]. The requirement that data is passed from one processing element to the next means that it is restricted to a certain type of computations, but is hard to apply in general.

MIMD is the architecture used in most parallel computers today. It is the most flexible architecture since there is no restriction on the number of instruction streams or data streams, although it is less efficient in terms of the number of instructions it uses to perform a single computation task compared to that used by a SIMD architecture.

Today, SIMD-style execution can be found in multimedia extension of most instruction set architectures, such as Intel MMX and SSE instructions. They are

not true SIMD because rather than a single instruction operating on many data items, the instructions still operate on a single data that is a packed form of multiple data items. The most common implementation of parallel computers takes the form of MIMD architectures.

1.4.1 Classes of MIMD Parallel Computers

One obvious question is that at which level processors in a MIMD architecture are physically interconnected. The choices are illustrated in Figure 1.2. Figure 1.2(a) shows an architecture in which the processors share the cache. Examples of this include many current multicore systems such as Intel and AMD dual core processors, IBM Power5, Sun Niagara, etc. Yet another alternative is to provide an interconnection between private caches (Figure 1.2(b)). This is the traditional architecture of *Symmetric Multi-Processors* or SMP. Examples include Pentium Pro Quad and Sun Enterprise. The interconnection may utilize a bus, ring, crossbar, or multistage network. Because all processors can access the memory with the same latency, then this class of MIMD computers are also referred to as *uniform memory access* (UMA) architectures. Figure 1.2(c) shows an architecture in which each processor has its own caches and local memory, but the hardware provides an interconnect across all local memories to give the abstraction of a single memory. However, the memory access latency varies since remote memory takes longer to access than local memory. Such architectures are referred to as *non-uniform memory access* (NUMA) or as *Distributed Shared Memory* (DSM). They include examples such as SGI Origin and Altix systems, and IBM p690 systems. Figure 1.2(d) shows an architecture in which each processor is a complete node with its own caches, local memory, and disk; and the interconnection is provided at the level of I/O connection. Since I/O connection latency is very high, the hardware may not provide abstraction of a single memory, although a software layer can accomplish the same abstraction with higher overheads. As a result, they are often referred to as *distributed* computer systems, or more popularly as clusters of workstations. Distributed computer systems are often not referred to as parallel computers.

The focus of this book is on the architectures of shared memory multiprocessors. Hence, we will mostly discuss the architectures shown in Figure 1.2(a)–(c).

Figure 1.2: Classes of MIMD parallel computers: shared cache (a), uniform memory access (b), non-uniform memory access (c), and distributed computer (d).

1.5 Examples of High-End Parallel Computers

The highest end of parallel computer market is dominated by supercomputers. One famous list of top supercomputers is maintained by `www.top500.org`. The list ranks supercomputing systems based on its Linpack benchmark performance. Linpack is a numerical linear algebra library that is highly parallel and floating point intensive [J. Dongarra, 1987]. The top ten of computers in that list can change dramatically from one year to another. For example, in 2004 the top honor went to the Earth Simulator, which was ranked the tenth in just a short two years later!

The Earth Simulator has 5,120 processors organized as 640 8-way nodes. Each processor is a 500 MHz NEC CPU. The peak performance of each processor is 8 *giga floating point operations per cycle* (GFLOPS), providing 41 TFLOPS total. It achieved more than 30 TFLOPS sustained performance. Each CPU has a 2 GB memory organized as 4 512 MB FPLRAM modules, providing a total of 10 TB memory. Shared memory abstraction is provided within a node. The nodes are interconnected using a 640×640 crossbar switch with a 16 GB/s inter-node bandwidth. The power consumption is 20 kVA per node. What is even more interesting is that three programming models are offered simultaneously: data parallelism using automatic vectorization in a CPU, shared memory using OpenMP in a node, and message passing using MPI across nodes.

■ *Did you know?*

Housing such a powerful supercomputer is no small feat. It requires a significant design effort in the building, power generation and delivery, as well as cooling, as evident in the following quote about the Earth Simulator (from `www.top500.org`*):*

> *"The machine room sits at approximately 4th floor level. The 3rd floor level is taken by hundreds of kilometers of copper cabling, and the lower floors house the air conditioning and electrical equipment. The structure is enclosed in a cooling shell, with the air pumped from underneath through the cabinets, and collected to the two long sides of the building. The aeroshell gives the building its "pumped-up" appearance. The machine room is electromagnetically shielded to prevent interference from nearby expressway and rail. Even the halogen light sources are outside the shield, and the light is distributed by a grid of scattering pipes under the ceiling. The entire structure is mechanically isolated from the surroundings, suspended in order to make it less prone to earthquake damage. All attachments (power, cooling, access walkways) are flexible."*

Other interesting statistics include the cost of the massive computer which is $40 million, with annual maintenance of $15 million. There is roughly one processor that fails per week.

Just to illustrate how fast the list of top 500 supercomputers changes, in August 2006, the number one honor went to IBM BlueGene system with 65,536 processors, where each processor is a PowerPC 440 with 700 MHz clock frequency. The peak performance is 186 TFLOPS. And in January 2008, IBM BlueGene/L took the top spot with Linpack performance of 478.2 TFLOPS, which is roughly 12 times the Earth Simulator's Linpack performance just four years earlier!

Part I

Shared Memory Parallel Programming

Chapter 2

Parallel Programming Models

Contents

2.1 Overview 19
2.2 Comparing Shared Memory and Message Passing Models . 22
2.3 A Simple Example 24

The objective of this chapter is to discuss two popular programming models that are widely used in parallel programming, important differences between them, and the relationship between the programming models and the architecture of the system. This chapter also describes why in the following chapters, we will focus on only one of the programming models, the shared memory programming model. With parallel architectures moving onto a single chip, the shared memory architecture can be supported with relative ease, providing a natural fit with the shared memory programming model.

2.1 Overview

A programming model is an abstraction provided by the hardware to programmers. It determines how easily programmers can specify their algorithms into computation tasks that the hardware understands, and how efficiently those tasks can be executed by the hardware. For uniprocessor systems, the sequential programming model has worked really well in hiding the hardware details from programmers, allowing programmers to effectively express their algorithms in an intuitive manner. In contrast, for multiprocessor systems, the right programming

model that can hide hardware details from the programmer still remains elusive.

There are at least two major parallel programming models that are widely used: *shared memory* and *message passing* programming models. There have been ongoing heated debates as to which one of them is superior. In the author's view, the reason why there are two widely used programming models is that they fulfill different needs and work better under different circumstances. Therefore, it is not clear if one will render the other unnecessary.

Let us define a *parallel task* as a unit of computation that can be executed independently from others. Parallel tasks can run on different processors. In the shared memory model, the abstraction is that parallel tasks can access any location of the memory. Hence, they communicate through reading and writing common memory locations. This is similar to threads from a single process which share a single address space. In message passing, parallel tasks have their own local memory and one task cannot access another task's memory. Hence, to communicate data, they have to rely on explicit messages sent to each other. This is similar to the abstraction of processes which do not share an address space.

As an analogy to the difference between these models, consider threads that want to communicate versus processes that want to communicate on a single processor system. Suppose that we are to create two threads (or processes) in which the main thread (process) asks the worker thread (process) to add two numbers and print the resulting sum.

In the thread model shown in Code 2.1, threads from a single process share a single address space, so the child thread automatically knows the value of variables `a` and `b` that are initialized by the main thread. However, the child thread must be prevented from accessing `a` and `b` until after the main thread initializes them. Hence, the `signal` is initially set to 0 and the child thread waits until it is set to 1. Finally, the main thread must not continue until the child thread finishes its execution, hence it waits until the signal is set back to 0 by the child thread.

In the process model, since processes have their private address spaces, a child process does not automatically know the values of `a` and `b`. The main and the child process must communicate through explicit message sending and receiving, as shown in Code 2.2. The code example assumes a simplified messaging interface in which `sendMsg(destID, d1, d2, ...)` represents sending a message containing data `d1`, `d2`, etc. to destination process `destID`. Similarly, `recvMsg(srcID, &d1, &d2, ...)` means receiving a message from source process `srcID` containing values that will be assigned to variables

Code 2.1 Thread model of communication.

```
int a, b, signal;
...
void dosum(<args>) {
  while (signal == 0) {}; // wait until instructed to work
  printf(``child thread> sum is %d\n'', a + b);
  signal = 0; // my work is done
}

void main() {
  a = 5, b = 3;
  signal = 0;
  clone(&dosum,...)              // spawn child thread
  signal = 1;                    // tell child to work
  while (signal == 1) {}         // wait until child done
  printf(``all done, exiting\n'');
}
```

d1, d2, etc. Because the processes do not share an address space, the main process must send a message containing the value of a and b, which must be received by the child process.

Code 2.2 Process model of communication.

```
int a, b;
...
void dosum() {
  recvMsg(mainID, &a, &b);
  printf(``child process> sum is %d'', a + b);
}

void main() {
  if (fork() == 0) { // i am the child process
   dosum();
  }
  else {             // i am the parent process
    a = 5, b = 3;
    sendMsg(childID, a, b);
    wait(childID);
    printf(``all done, exiting\n'');
  }
}
```

Overall, the simple code examples illustrate a few things. The thread model

is lighter in the sense that it does not incur much communication overhead since data is not explicitly communicated. However, it needs synchronization in order to control the order in which the operations are performed by various threads. The process model is heavier in the sense that it suffers from the overheads of sending and receiving explicit messages that contain the data that is communicated. On the other hand, the explicit sends and receives of messages also serve to synchronize the sequence of operations in both processes. These differences also exist when we compare shared memory versus message passing programming models.

2.2 Comparing Shared Memory and Message Passing Models

Table 2.1: Comparison between shared memory and message passing programming models.

Aspects	Shared Memory	Message Passing
Communication	implicit (via loads/stores)	explicit messages
Synchronization	explicit	implicit (via messages)
Hardware support	typically required	none
Development effort	lower	higher
Tuning effort	higher	lower

Table 2.1 summarizes the differences between shared memory and message passing programming model. The first two aspects (communication and synchronization) are similar to the differences that exist in comparing the thread versus process programming models. The shared memory programming model does not communicate data between threads but explicit synchronization is needed to order data accesses among threads. The message passing programming model communicates data between threads through sending and receiving messages, which also implicitly act as synchronization that control the relative ordering of data access by different threads.

Providing an abstraction of shared memory is typically achieved through special hardware support. For small systems, such as multicore systems, processor cores may already share the lowest level cache; supporting the shared memory abstraction is simple and in some cases automatic. However, when multiple nodes, each of which has its own processor and memory, are interconnected to form a shared memory system, hardware support is necessary to give an illusion that all nodes' memories form a single memory that is addressable by all processors. Message passing programming model, on the other hand, does not require such hardware support. Providing a shared memory abstraction is difficult to achieve with at a low cost when the number of processors is large. Current shared memory system is often limited to a few hundred to a few thousand processors (e.g. 128 processors in SGI Origin 2000 in 1998-1999, 2048 processors in SGI Altix 3700 in 2000). For very large systems with thousands or more processors, shared memory abstraction is typically not provided.

Shared memory parallel programs are often easier to develop initially. The reason is that programmers need not worry about how data is laid out and how it is mapped to processors because data can be communicated between processors regardless of where it is located or how it is laid out. In addition, since a shared memory abstraction provides implicit communication, a typical shared memory program code structure is often not much different than its sequential version. In fact, in many cases, all programmers need to do is to insert into the program source code directives that identify parallel tasks, scope of variables, and synchronization points. The compiler translates those directives into code that spawn and control parallel threads. On the other hand, writing a message passing program requires the programmers to immediately think about how to partition data among processors and how to communicate data among them through explicit message sending and receiving. Hence, the initial development effort tends to be lower (sometimes significantly) in shared memory programs.

However, once the parallel program is up and running, and when programmers want the program to scale to a large number of processors, where data is located and how it is laid out start to impact performance significantly. For example, accessing a remote memory may take multiple times longer than accessing the local memory. Hence, it is important to lay out data closest to the processors which access it the most. While message passing programmers already think about this issue when they develop their programs, shared memory programmers eventually face the same issue during post-development time. Hence, the effort to tune the

programs to be highly scalable is often higher in shared memory programs than in message passing programs.

Note that message passing programs may also benefit from a shared memory abstraction provided by the hardware. For example, if there is no shared memory, to send a message, the message must be bundled into packets and sent off the network. With shared memory abstraction, sending and receiving a message is simple: sending a message involves writing the message into a message buffer in memory and passing the buffer pointer to the receiver, while receiving a message involves reading the message from the message buffer using the pointer that was passed by the sender. Since passing a pointer is faster than sending packets of a message, even message passing programs benefit from shared memory abstraction.

This book focuses on how to build hardware support for shared memory multiprocessors. As a result, the emphasis of the discussion in this book is on the shared memory programming model.

■ *Did you know?*

Naturally, shared memory programming model fits better with shared memory multiprocessor systems. However, shared memory programs can also be run on distributed computers with the abstraction of shared memory provided by a software layer at the page granularity. Such systems are referred to as software virtual shared memory *(SVM) systems. In general, SVMs suffer from a high inter-processor communication latency, which limits their practicality.*

Interestingly, the message passing programming model benefits from running on a shared memory multiprocessor. For example, in order to send a message from one processor to another, on a shared memory multiprocessor the sending processor only needs to communicate a pointer to a message buffer to the receiving processor. The content of the message itself does not need to be sent as the receiving processor can read it directly from the main memory, allowing low-latency message communication. Such a direct message communication is supported by several message passing libraries.

2.3 A Simple Example

This section shows a simple example contrasting the shared memory and message passing models. The codes used for comparison are for illustration purpose only,

and they are not optimal in terms of performance. The codes are written in pseudo-code style, so they do not correspond directly to any particular syntax required for popular shared memory programming standards (e.g. OpenMP) or message passing programming standards (e.g. MPI).

Suppose we want to execute a task shown in the following code on two processors (Code 2.3). The code attempts to add an array `b` and `c` into array `a` in the first loop, and in the second loop the elements of array `a` are added together into the variable `sum`.

Code 2.3 A simple code.

```
1 for (i=0; i<8; i++)
2   a[i] = b[i] + c[i];
3 sum = 0;
4 for (i=0; i<8; i++)
5   if (a[i] > 0)
6     sum = sum + a[i];
7 Print sum;
```

The shared memory program corresponding to it is shown in Code 2.4.

Note that here we need to declare whether the variables are visible to all threads (declared as *shared*) or visible only to the local thread (declared as *private*). In the example, each thread computes the range of iterations that it will work on. `start_iter` and `end_iter` are declared as private so each thread computes different values. The statement `barrier` is a synchronization primitive inserted to ensure that all threads reach that point before any of them is allowed to move past it. The update to variable `sum` is performed by multiple threads, so to avoid multiple simultaneous updates, we guard it with a *critical section* through the statements `lock(mylock)` and `unlock(mylock)`. A critical section allows only one thread to enter at a time. The critical section is for illustration purpose only since such a fine grain critical section does not produce optimal performance.

The message passing program corresponding to it is shown in Code 2.5.

In the example, we assume that initially only the main process has access to the arrays. Hence, before the main process is able to ask the other process to execute part of the loop, it must send parts of the arrays that the other process will need. Since the other process will execute the second part of the loop, the main process only sends the bottom halves of array `b` and `c`. Finally, the local sum that is computed by the other thread needs to be collected back by the main process,

Code 2.4 The shared memory code corresponding to Code 2.3.

```
begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter = 4, sum = 0;
shared double sum = 0.0, a[], b[], c[];
shared lock_type mylock;

start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];
barrier;

for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0) {
    lock(mylock);
    sum = sum + a[i];
    unlock(mylock);
  }
barrier;     // necessary

end parallel // kill the child thread

Print sum;
```

and accumulated to the sum before it is printed.

The examples in Code 2.4 and Code 2.5 reiterate the differences in explicit synchronization in shared memory programs versus explicit communication and implicit synchronization in message passing programs. However, there is an additional point to make. In the message passing programs, data must be sent from one thread to all other threads that wish to read it. These threads keep a copy of the data in their local memory, even for read-only data. In shared memory programs, there is only one copy of most data (especially the read-only one) kept in the main memory. Hence, shared memory programs have smaller code and data footprints in the main memory. In addition, the total physical memory is the addition of all local memories in a shared memory system, while in a non-shared memory system each processor's main memory is limited by its own physical memory size. In some cases, the latter benefit alone is sufficient for some programmers to choose to run their applications on shared memory multiprocessor systems over distributed systems.

Code 2.5 The message passing code corresponding to Code 2.3.

```
id = getmyid(); // assume id=0 thread 0, id=1 thread 1
local_iter = 4;
start_iter = id * local_iter;
end_iter = start_iter + local_iter;

if (id == 0)
  send_msg (P1, b[4..7], c[4..7]);
else
  recv_msg (P0, b[4..7], c[4..7]);

for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];

local_sum = 0;
for (i=start_iter; i<end_iter; i++)
  if (a[i] > 0)
    local_sum = local_sum + a[i];
if (id == 0) {
  recv_msg (P1, &local_sum1);
  sum = local_sum + local_sum1;
  Print sum;
}
else
  send_msg (P0, local_sum);
```

Chapter 3

Shared Memory Parallel Programming

Contents

3.1 Steps in Parallel Programming **30**
3.2 Identifying Code and Loop-Level Parallelism **32**
3.2.1 Iteration-space Traversal Graph and Loop-Carried Dependence Graph 34
3.2.2 Finding Parallel Tasks Across Iterations 35
3.2.3 DOACROSS Parallelism 40
3.2.4 Parallelism Across Statements in a Loop 43
3.2.5 DOPIPE Parallelism Across Statements in a Loop . . 45
3.3 Identifying Algorithm-Level Parallelism **48**
3.4 Determining the Scope of Variables **51**
3.4.1 Privatization 53
3.4.2 Reduction Variables and Operation 55
3.4.3 Summary of Criteria 56
3.5 Synchronization **57**
3.6 Mapping Tasks to Processors and Memory **59**
3.7 A Brief Introduction to OpenMP **60**
3.8 Example: Matrix Multiplication Parallelization **64**
3.9 Exercise Problems **68**

In the previous chapter, we have mentioned that in this book, we will focus mostly on the architecture of shared memory multiprocessors. Since the predom-

inant programming model used in such systems will be the shared memory programming model, we focus our discussion on it. The objective of this chapter is to discuss steps needed to create shared memory parallel programs. The focus is on identifying parallel tasks by analyzing the code or algorithm, how to determine the scope of variables, coordinate the parallel tasks, and express parallelism to the compiler. The performance of the resulting shared memory parallel program is a secondary concern in this chapter, but it will be the focus of the next chapter.

It is a good idea to study shared memory parallel programming prior to studying the architecture of shared memory multiprocessors, in order to get a handle on how software uses the systems, what constructs are important, and what software issues are relevant when considering the design space of a shared memory multiprocessor.

3.1 Steps in Parallel Programming

In general, shared memory parallel programming involves several steps shown in Figure 3.1. The first step in creating a parallel program is identifying sources of parallelism in the code. Various analyses techniques at different levels (e.g. code level and algorithm level) can be performed to identify parallelism. These will be discussed in Section 3.2 and 3.3. Once parallel tasks are identified, they can be grouped into threads. Then, we need to determine the scope of each variable used by the tasks (Section 3.4). The choice is whether the variable should be shared by all threads or should be private to each thread. The next step involves coordinating the threads through synchronization. We will discuss what types of synchronization should be inserted and where they should be inserted (Section 3.5). Collectively, the parallelism identification step, variable scope determination, and synchronization, can be referred to as *task creation* since during these steps we create tasks that can cooperate to perform a single computation.

The next step deals with grouping tasks into threads. Typically, there are many more tasks than available processors, so grouping multiple tasks into a single thread can increase the granularity of work performed by each thread, which lowers overheads of parallelism. Another goal of task-to-thread mapping is to balance the work load of the threads. The final step deals with mapping the threads to processors, and laying out data in memory. The goal of this step is to achieve communication locality, in which processors that communicate with each other are located in close proximity to one another, and data locality, in which data ac-

Figure 3.1: Steps in shared memory parallel programming.

cessed by each processor is located as close as possible to the processor. These last two steps can be collectively referred to as *task mapping*. Sometimes task mapping can be hidden from programmers, for example, by using the default mapping that the system employs. However, relying on such default mapping sometimes produces suboptimal performance and scalability.

The first step in task creation is identifying parallel tasks. There are three levels at which this can be done: *program-level*, *algorithm-level*, and *code-level*. An example of parallel tasks at the program level is parallel compilation of files. For example, parallel make tools (e.g. `pmake`) spawn processes that compile different source files independently from one another. Program-level parallelism is effective, but is restricted to the case in which multiple instances of a program are run simultaneously. In addition, exploiting program-level parallelism is sometimes trivial. Consequently, we will focus our discussion on identifying parallelism at the code level and algorithm level. Identifying code-level parallelism is a versatile technique since we only need to analyze code structures, without needing to

know the algorithms that the code represents. However, sometimes knowing the algorithms can give extra parallelism opportunities that are missed in code-level parallelism.

3.2 Identifying Code and Loop-Level Parallelism

The goal of identifying parallelism at the code-level is that given a code, we find parallel tasks without having or relying on the knowledge of the algorithms. While the focus of our discussion is in shared memory parallel programming, the code-level parallelism identification is also applicable for message passing programming.

Code-level parallelism identification often focuses on code sections that show regular and analyzable structures and ones that contribute significantly to the execution time. In numerical programs, such structures are typically found in loops. Supercomputing applications, which have been a key driver for parallel computers, typically rely on simulations of physical phenomena. Many models of physical phenomena heavily use matrices as their main data structures to model objects in the physical space. Accessing these multi-dimensional data structures (the matrices) requires the use of nested loops to iterate over different dimensions. Therefore, analyzing loop structures is a very fruitful way to identify parallelism for these applications. Non-numerical applications seldom use matrices. Instead, they use linked data structures such as linked lists, hash tables, trees, and graphs. While traversing these structures also often involve loops, the code structures of these loops are much more complex since many addresses of pointers cannot be inferred until run-time, and sometimes the loops involve recursion. As a result, loop-level code-analysis is less fruitful in identifying parallelism in non-numerical applications.

To allow a systematic way of performing code analysis, it is necessary to define a framework. We will start by defining several notations. Let S denote a statement in the source code. The loop iteration space is denoted inside a bracket []. For example, for a doubly nested loop with i iterating on the outer loop and j iterating on the inner loop, we will denote the iteration space as $[i, j]$. We can express a statement S executed in a particular iteration $[i, j]$ as $S[i, j]$. Let $S1 \rightarrow S2$ denote that statement $S1$ comes before $S2$ in the order of program execution. Furthermore, let us define:

- $S1 \rightarrow^T S2$ denotes *true dependence*, i.e. $S1 \rightarrow S2$, and $S1$ writes to a

location that is read by $S2$. In other words, $S1$ is the *producer* of data that is read by the *consumer* ($S2$).

- $S1 \rightarrow^A S2$ denotes *anti dependence*, i.e. $S1 \rightarrow S2$, and $S1$ reads a location written by $S2$.

- $S1 \rightarrow^O S2$ denotes *output dependence*, i.e. $S1 \rightarrow S2$, and $S1$ writes to the same location written by $S2$.

To illustrate the dependences, Code 3.1 shows a code section with four statements.

Code 3.1 A simple code with four statements.

```
1 S1: x = 2;
2 S2: y = x;
3 S3: y = x + z;
4 S4: z = 6;
```

The dependences corresponding to the code are:

- $S1 \rightarrow^T S2$ because x is written in statement $S1$ and read in statement $S2$.

- $S1 \rightarrow^T S3$ because x is written in statement $S1$ and read in statement $S3$.

- $S3 \rightarrow^A S4$ because z is read in statement $S3$ and written in statement $S4$.

- $S2 \rightarrow^O S3$ because y is written in statement $S2$ and also written in statement $S4$.

Next, let us define *loop-carried dependence* as a dependence that exists between a statement in one iteration with another statement in a different iteration, while *loop-independent dependence* as a dependence that exists between statements within a loop iteration. The difference between them are shown in Example 3.2.

Code 3.2 A code segment illustrating dependences in loop structures.

```
for (i=1; i<n; i++) {
  S1: a[i] = a[i-1] + 1;
  S2: b[i] = a[i];
}
for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S3: a[i][j] = a[i][j-1] + 1;
for (i=1; i<n; i++)
  for (j=1; j< n; j++)
    S4: a[i][j] = a[i-1][j] + 1;
```

In the first statement in the first loop ($S1$), we have a write to $a[i]$ and a read from $a[i-1]$, which means that a value written to $a[i]$ will be read in the next iteration (the $i+1^{th}$ iteration). Hence, we have $S1[i] \rightarrow^T S1[i+1]$ dependence, which is loop-carried. For example, in iteration $i = 4$, statement $S1$ writes to $a[4]$ and reads from $a[3]$, which was earlier written in iteration $i = 3$. In addition, the value written to $a[i]$ is read by statement $S2$ in the same iteration, so we have a loop-independent dependence of $S1[i] \rightarrow^T S2[i]$.

In the second loop, a value written to $a[i][j]$ will be read in the next j iteration, hence we have a dependence $S3[i][j] \rightarrow^T S3[i][j+1]$, which is loop-carried for the $for\ j$ loop but is loop-independent for the $for\ i$ loop.

Finally, in the third loop, a value is written to $a[i][j]$ will be read in the next i iteration, hence we have a dependence $S4[i][j] \rightarrow^T S4[i+1][j]$, which is loop-carried for the $for\ i$ loop but is loop-independent for the $for\ j$ loop. Overall, the dependences are:

- $S1[i] \rightarrow^T S1[i+1]$
- $S1[i] \rightarrow^T S2[i]$
- $S3[i][j] \rightarrow^T S3[i][j+1]$
- $S4[i][j] \rightarrow^T S4[i+1][j]$

3.2.1 Iteration-space Traversal Graph and Loop-Carried Dependence Graph

An *iteration-space traversal graph* (ITG) shows graphically the order of traversal in the iteration space. ITG does not show dependences; it only shows the order in which loop iterations are visited.

Loop-carried Dependence Graph (LDG) shows true/anti/output dependences graphically, in which a node is a point in the iteration space and the directed edge shows the direction of the dependence. Note that a node represents all statements in a single iteration, so the LDG does not show loop-independent dependences. LDG can essentially be obtained by drawing the dependences for each of the iteration.

For example, the following code (Code 3.3) has the ITG and LDG shown in Figure 3.2. The LDG is essentially an *unrolled* version of the dependence $S3[i,j] \rightarrow^T S3[i,j+1]$. In the figure, we can visually observe that the dependence is loop-carried on the $for\ j$ loop but not on the $for\ i$ loop.

Code 3.3 A simple code with loop-carried dependence.

```
1 for (i=1; i<4; i++)
2   for (j=1; j<4; j++)
3     S3: a[i][j] = a[i][j-1] + 1;
```

Figure 3.2: ITG and LDG corresponding to Code 3.3.

Example 3.1 and Example 3.2 show more complex cases.

3.2.2 Finding Parallel Tasks Across Iterations

One of the most fruitful ways to identify parallelism is by analyzing which loop iterations can be executed in parallel. In order to do that, we start by analyzing

■ **Example 3.1** An illustration of iteration-space traversal graph (ITG) and loop-carried dependence graph (LDG).

For the following code:

```
1 for (i=1; i<n; i++) {
2   for (j=1; j<n; j++) {
3     S2: a[i][j] = b[i][j] + c[i][j];
4     S3: b[i][j] = a[i][j-1] * d[i][j];
5   }
6 }
```

Identify all dependences, both loop-carried and loop-independent, and draw its ITG and LDG.

Answer:

Here, a value read from $b[i][j]$ in statement $S2$ is written in statement $S3$ in the same iteration. So we have an anti-dependence $S2[i,j] \rightarrow^A S3[i,j]$, which is loop independent.

A value written to $a[i][j]$ in statement $S2$ is read in the next j iteration in statement $S3$. Hence, we have a true dependence $S2[i,j] \rightarrow^T S3[i,j+1]$.

The ITG and LDG are as shown:

the loop-carried dependences. The first principle is that dependences must be obeyed, especially true dependences. Note that anti and output dependences can be removed through *privatization* (discussed later in Section 3.4.1); for now we assume that all the dependences must be obeyed. The dependence between two iterations can be observed visually in the LDG in the form of an edge connecting the two nodes that represent the iterations. It may also be observed as a path (a

■ **Example 3.2** An illustration of iteration-space traversal graph (ITG) and loop-carried dependence graph (LDG).

In the following code:

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    S1: a[i][j] = a[i][j-1] + a[i][j+1] +
            a[i-1][j] + a[i+1][j];
```

Identify all dependences, both loop-carried and loop-independent, and draw its ITG and LDG.

Answer:

A value written to $a[i][j]$ is read in the next j iteration (by a part of the statement "$a[i][j-1]$") and in the next i iteration (by a part of the statement "$a[i-1][j]$"). Hence, there are two true dependences: $S1[i,j] \rightarrow^T S1[i,j+1]$ and $S1[i,j] \rightarrow^T S1[i+1,j]$.

In addition, the part of the statement $a[i][j+1]$ reads in iteration $[i,j]$ the value that is not yet written as it will be written at iteration $[i][j+1]$. Hence, we have an anti-dependence $S1[i,j] \rightarrow^A S1[i,j+1]$. Similarly, the part of the statement $a[i+1][j]$ reads in iteration $[i,j]$ the value that will be written at iteration $[i+1][j]$, hence we have an anti-dependence $S1[i,j] \rightarrow^A S1[i+1,j]$.

The ITG and LDG are as shown:

series of edges) connecting the two nodes. Only when there is no edge or path that connects two nodes can we say that the two nodes have no dependences between them. Iterations that do not have dependences with each other can be executed in parallel. For example, a very simple loop is shown in Code 3.4. Its LDG is shown

in Figure 3.3.

Code 3.4 A simple example.

```
1 for (i=2; i<=n; i++)
2   S: a[i] = a[i-2];
```

Figure 3.3: The LDG corresponding to Code 3.4.

From the LDG, we can see that odd iterations do not have edges pointing to even iterations, and even iterations do not have edges pointing to odd iterations. Hence, we can extract two parallel tasks: one task executing odd iterations and another executing even iterations. To achieve that, we can split the loop into two smaller loops. The two loops can now be executed in parallel with respect to each other, although each loop still needs to be executed sequentially.

Code 3.5 Split loops from the original loop in Code 3.4.

```
1 for (i=2; i<=n; i+=2)
2   S: a[i] = a[i-2];
3 for (i=3; i<=n; i+=2)
4   S: a[i] = a[i-2];
```

For another example of identifying loop-level parallelism, consider the code in Example 3.1, in which the LDG shows loop-carried dependences (edges in the graph) across the j iterations, but not across i iterations. This means that all $for\ i$ iterations are independent of one another and each of them can be a parallel task. When all iterations of a loop are parallel tasks, we refer to the loop as exhibiting a *DOALL* parallelism. In OpenMP standard, expressing a DOALL parallelism is simple; we only need to add a directive right above the $for\ i$ statement (we will discuss it more later).

A more complex example is for the code in Example 3.2, for which the LDG shows loop-carried dependences across the i as well as the j iterations. It may at first seems that no parallel tasks can be extracted from the code. However, note that there are no loop-carried dependences between *anti-diagonal* loop iterations. In each anti-diagonal, no two nodes in the graph have edges or paths of edges that point to each other (Figure 3.4). However, the same cannot be said for nodes in diagonals. While no direct edges exist between nodes in a diagonal, but paths between them exist.

Figure 3.4: The LDG from Example 3.2 showing no dependences between nodes in anti-diagonals. Anti-diagonals are shown in dash-dot angled boxes

Unfortunately, it is not easy to specify such parallel tasks to the compiler. For example, OpenMP parallel directives only allow us to specify DOALL parallelism for a particular loop and do not allow us to specify anti-diagonal parallelism across two loops in a loop nest. One way to remedy this limitation is to restructure the code so that one loop iterates over anti-diagonals, and an inner loop iterates over nodes in an anti-diagonal. We can then specify DOALL parallelism for the inner loop. The pseudo-code for this is shown in Code 3.6.

Code 3.6 Restructured pseudo-code that iterates over anti-diagonals, performing an equivalent computation to the original loop in Example 3.2.

```
1 Calculate number of anti-diagonals
2 foreach anti-diagonal do {
3   Calculate number of points in current anti-diagonal
4   foreach point in the current anti-diagonal do {
5     compute the current point in the matrix
6   }
7 }
```

3.2.3 DOACROSS Parallelism

The DOALL parallelism described in the previous section is simple as it specifies that all iterations of a loop are parallel tasks. Typically, the number of parallel tasks in a DOALL parallel loop is very high, so identifying DOALL parallelism should be attempted before identifying other types of parallelism. However, in some loops such parallelism is not available due to loop-carried dependences across loop iterations. How can we extract parallelism in such cases? In this section, we will look at *pipelined parallelism* that allow us to extract parallel tasks even for loops that exhibit loop-carried dependences.

For example, take a look at the code shown below (Code 3.7).

Code 3.7 A loop with loop-carried dependence.

```
1 for (i=1; i<=N; i++)
2   S: a[i] = a[i-1] + b[i] * c[i];
```

The loop has loop-carried dependence $S[i] \rightarrow^T S[i+1]$ so clearly it does not have a DOALL parallelism. However, upon a closer inspection we can notice that part of the statement that multiplies $b[i]$ with $c[i]$ do not have loop-carried dependence. This presents an opportunity for parallelism. There are two ways to exploit this opportunity. The first choice is to split the loop into two loops: the first loop only executing the part of the statement that is free from loop-carried dependence, and the second loop only executing the part of the statement with loop-carried dependence. The resulting code is shown below (Code 3.8). The code shows that in order to separate the two statements, the first loop must store its multiplication of $b[i]$ and $c[i]$ results temporarily in an array $temp[i]$. With this solution, the first loop has DOALL parallelism and the second one does not.

Unfortunately, the solution has a high storage overhead since we have to introduce a new array $temp[i]$, whose size increases with the array size used in the code.

Code 3.8 A split version of the loop in Code 3.7.

```
1 for (i=1; i<=N; i++)          // this loop has DOALL parallelism
2   S1: temp[i] = b[i] * c[i];
3 for (i=1; i<=N; i++)          // this loop does not
4   S2: a[i] = a[i-1] + temp[i];
```

Another solution for extracting parallel tasks in a loop with a partial loop-carried dependence is to express a DOACROSS parallelism, in which each iteration is still a parallel task (similar to DOALL), but synchronization is inserted in order to ensure that the consumer iteration only reads data that has been produced by the producer iteration. This is achieved by inserting *point-to-point synchronization* between parts of the statements that show loop-carried dependence. The synchronization primitives needed are: *post*, which is called by the producer of data to signal that the data is ready, and *wait*, which is called by the consumer to block until data is ready. To simplify the discussion, we assume that the primitives are named, i.e. `wait(name)` unblocks only when there is a corresponding `post(name)`. The loop now becomes:

Code 3.9 Exploiting DOACROSS parallelism in the loop from Code 3.7.

```
1 post(0);
2 for (i=1; i<=N; i++)  {
3   S1: temp = b[i] * c[i];
4   wait(i-1);
5   S2: a[i] = a[i-1] + temp;
6   post(i);
7 }
```

In Code 3.9, the original statement is broken into two: $S1$, that has no loop-carried dependence, and $S2$, that has loop-carried dependence. The temporary multiplication result is stored in a private (per-thread) variable `temp` in statement $S1$ and is read in statement $S2$. With DOACROSS, `temp` is now a private scalar rather than a shared array. Thus, the storage overheads only scales with the number of threads rather than the number of iterations. Note the statement `post(i)`. It is a signal by the producer that the value of $a[i]$ has been produced

and is ready for consumption by whoever is waiting on it. Also note the statement `wait(i-1)`. It is a signal by the consumer that it must wait until the value $a[i-1]$ has been produced by the producer, which in this case the previous i iteration. Also note that in the first iteration, statement $S2$ reads from $a[0]$ which is not produced by any iteration. Hence, we can add `post(0)` to ensure that the first iteration does not block forever. With the transformation and insertion of synchronization, the iterations of the loop are now parallel tasks, and the execution is shown in Figure 3.5.

Figure 3.5: Execution of the first three iterations of the DOACROSS code in Code 3.9.

One may wonder what execution time saving we obtain. Let us for now assume that the synchronization latency is zero. Let T_{S1} and T_{S2} denote the execution time of statement $S1$ and $S2$ respectively. If the loop is executed sequentially, the execution time is $N \times (T_{S1} + T_{S2})$. If executed with DOACROSS, $S1$ can be executed in parallel with respect to all statements in other iterations. The statement that gets serialized is $S2$, in which each $S2$ waits until the same statement from the previous iteration has produced the data it waits for. Hence, the execution time is $T_{S1} + N \times T_{S2}$, which is an improvement over the original execution time. If we assume a simple case in which $T_{S1} = T_{S2} = T$ and $N \gg T$, the speedup ratio is $\frac{2NT}{(N+1)T} \approx 2\times$. In practice, how much improvement we actually get depends on how large is the execution time of parts of the loop that do not have loop-carried dependence (e.g., T_{S1}) versus the parts of the loop that have to be serialized due to

loop-carried dependence (e.g., T_{S2}), and the magnitude of synchronization overheads. Since synchronization overheads are often substantial, programmers must be careful when using DOACROSS parallelism. One way to reduce the synchronization overheads is to group many parallel tasks (iterations) into a thread, so since the synchronization can be performed between threads rather than between tasks, the frequency of synchronization can be greatly reduced.

3.2.4 Parallelism Across Statements in a Loop

When a loop has loop-carried dependence, another way to parallelize it is to *distribute* the loop into several loops that execute different statements from the original loop body. For example, consider the loop in Code 3.10.

Code 3.10 An example of loop distribution.

```
1 for (i=0; i<n; i++) {
2   S1: a[i] = b[i+1] * a[i-1];
3   S2: b[i] = b[i] * coef;
4   S3: c[i] = 0.5 * (c[i] + a[i]);
5   S4: d[i] = d[i-1] * d[i];
6 }
```

The loop has loop carried dependence $S1[i] \rightarrow^T S1[i+1]$ and $S4[i] \rightarrow^T S4[i+1]$, and anti-dependence $S1[i] \rightarrow^A S2[i+1]$. It also has a loop-independent dependence $S1[i] \rightarrow^T S3[i]$. The lack of loop-carried dependences in all statements presents an opportunity for parallelism. For example, note that S4 has no dependences with other statements in the loop body. Hence the loop can be distributed into two loops where one of them executes the first three statements and the second loop executes the last statement. The resulting code is shown in the first part of Code 3.11. The two loops can now be executed in parallel with respect to each other, although each loop must still be executed sequentially.

Instead of each parallel task performing similar computation on different data as in DOALL and DOACROSS parallelism, here each parallel task performs different computation on different sets of data. We can refer it to as *function parallelism* as opposed to *data parallelism* exhibited by DOALL and DOACROSS parallelism. The characteristics of function parallelism are that the degree of parallelism is typically modest and does not increase with a larger input size. This is because the source of parallelism is the code structure rather than the data that the code works on. Since different parallel tasks execute different computation, it

is often difficult to balance the load across tasks. Hence, most scalable programs have abundant data parallelism. However, function parallelism can enhance parallelism when data parallelism is limited.

To get an idea of how much speedup can be obtained through the transformation shown in the first part of Code 3.11, let $T_{S1}, T_{S2}, T_{S3}, T_{S4}$ denote the execution time of statement $S1, S2, S3, S4$ respectively. If the loop is executed sequentially, the execution time is $N \times (T_{S1} + T_{S2} + T_{S3} + T_{S4})$. With the transformation, the new execution time is $max(N \times (T_{S1} + T_{S2} + T_{S3}), N \times T_{S4})$. If we assume a simple case in which $T_{S1} = T_{S2} = T_{S3} = T_{S4} = T$ and $N \gg T$, the speedup ratio is $\frac{4NT}{3NT} \approx 1.33\times$, which is nice but quite limited.

Code 3.11 An example of loop distribution.

After distributing statement $S4$, the resulting two loops can be executed in parallel with respect to one another:

```
1 for (i=0; i<n; i++) {
2   S1: a[i] = b[i+1] * a[i-1];
3   S2: b[i] = b[i] * coef;
4   S3: c[i] = 0.5 * (c[i] + a[i]);
5 }
6 for (i=0; i<n; i++)
7   S4: d[i] = d[i-1] * d[i];
```

After distributing statement $S2$ and $S3$:

```
1 for (i=0; i<n; i++)
2   S1: a[i] = b[i+1] * a[i-1];
3 for (i=0; i<n; i++)
4   S4: d[i] = d[i-1] * d[i];
5 for (i=0; i<n; i++)
6   S2: b[i] = b[i] * coef;
7 for (i=0; i<n; i++)
8   S3: c[i] = 0.5 * (c[i] + a[i]);
```

Note the dependence of $S1[i] \rightarrow^A S2[i+1]$ implies that statement $S2$ in iteration $i+1$ must be executed after statement $S1$ in iteration i. Hence, the dependence is not violated if all $S2$'s are executed after all $S1$'s. Similarly, the dependence of $S1[i] \rightarrow^T S3[i]$ implies that statement $S3$ in iteration i must be executed after statement $S1$ in iteration i. Hence, no dependence is violated if all $S3$'s are executed after all $S1$'s. Therefore, we can further distribute the loop into four loops, as shown in the second part of Code 3.11. The execution of the loops is

Figure 3.6: Execution of distributed loops from Code 3.11.

illustrated in Figure 3.6. The first two loops executing statement $S1$ and $S4$ can be executed in parallel with respect to one another. Once the first loop completes, the third loop executing statement $S2$ and the fourth loop executing statement $S3$, can both run with DOALL parallelism. The mechanism for ensuring that the third and fourth loops execute only after the first loop completes execution can be enforced with a point-to-point synchronization, or with a barrier.

With the new strategy, the new execution time for the loop becomes $max(N \times T_{S1} + max(T_{S2}, T_{S3}), N \times T_{S4})$. Again, if we assume a simple case in which $T_{S1} = T_{S2} = T_{S3} = T_{S4} = T$ and $N \gg T$, the speedup ratio is $\frac{4NT}{(N+1)T} \approx 4\times$, which is a significant improvement over the $1.33\times$ speedup ratio achieved by the previous strategy.

3.2.5 DOPIPE Parallelism Across Statements in a Loop

We can also exploit pipelined parallelism for loops that exhibit loop-carried dependence. For example, consider the loop shown in Code 3.12. One dependence the loop has is $S1[i] \rightarrow^{T} S1[i+1]$, hence the loop does not have DOALL parallelism. Let T_{S1} and T_{S2} denote the execution time of statement $S1$ and $S2$ respectively. Note that one option is using the technique from the previous section to distribute the loop. The first statement can be executed in the first loop sequentially, and the second loop executed with DOALL parallelism after the first loop completes execution. In such a case, the execution time becomes $N \times T_{S1} + \frac{N}{P} \times T_{S2}$, where N is the number of iterations of the loop, and P is the number of proces-

sors. Unfortunately, if we only have a few processors, then the speedup is not much.

Code 3.12 A loop amenable to pipelined parallelism.

```
for (i=1; i<=N; i++) {
  S1: a[i] = a[i-1] + b[i];
  S2: c[i] = c[i] + a[i];
}
```

An alternative solution that can achieve better parallelism in this case is by distributing the loop and introducing pipelined parallelism. With pipelined parallelism, right after $a[i]$ is produced by the statement $S1$ in the first loop, the second loop executes statement $S2$ that reads the just-produced value of $a[i]$. This type of parallelism is referred to as DOPIPE, and the resulting code is show in Code 3.13. The illustration of how DOPIPE parallelism is executed is shown in Figure 3.7.

Code 3.13 DOPIPE parallel version of the loop from Code 3.12.

```
for (i=1; i<=N; i++) {
  S1: a[i] = a[i-1] + b[i];
  post(i);
}
for (i=1; i<=N; i++) {
  wait(i);
  S2: c[i] = c[i] + a[i];
}
```

Assuming zero-latency synchronization and that statement $S1$ takes the same amount of time to compute as statement $S2$, the execution time becomes $N \times T_{S1} + T_{S2}$.

This compares favorably to the alternative of executing the first loop to completion before starting the second loop. Of course, just like any function parallelism, DOPIPE parallelism suffers from a limited degree of parallelism and difficulty in ensuring load balance across parallel tasks.

Figure 3.7: Execution of the iterations of the DOPIPE code in Code 3.12.

> ■ *Did you know?*
>
> *The amount of synchronization overheads can be reduced in the DOPIPE parallelism by introducing a* synchronization interval. *Rather than calling a* `post(a[i])` *each time one* $a[i]$ *is produced, we can use one* `post(a[i])` *for every* n *iterations instead, where* n *is the synchronization interval. This is simple to do as programmers only need to wrap the* `post` *and* `wait` *statements by* `if (i % n == 0) { ... }`.

In a multicore architecture, multiple processor cores often share the lowest level cache. In such architectures, one unique benefit of DOPIPE parallelism is that it is cache efficient in that data values produced by the producer task are almost immediately consumed by the consumer task. Hence, its temporal locality is good. This is in contrast to DOALL parallelism in which each task works on a different iteration using largely different sets of data. If there are many tasks assigned to a thread, many threads will compete for the cache space, and there is a possibility of the cache being thrashed between different working sets that belong to different threads.

DOPIPE parallelism is a form of pipelined parallelism applied at the loop level. For some applications, pipelined parallelism may exist at the level of tasks,

such as across loops, across functions, or across logical stages in the algorithm. Such task-level pipelined opportunities require dependence analysis that is rarely provided by a compiler. It is usually up to the programmer to identify and exploit pipelined parallelism.

3.3 Identifying Algorithm-Level Parallelism

Sometimes, only analyzing code structure does not give us the maximum amount of parallelism. Analyzing the algorithm may reveal more opportunities for extracting parallel tasks. This is because the code structure embeds unnecessary serialization that is the artifact of a sequential programming language.

For example, consider an algorithm that updates the force of a water particle by its four neighbors, shown in Code 3.14.

Code 3.14 A code that simulates ocean current.

The algorithm for the computation of the main loop is:

```
While not converging to a solution do:
  foreach timestep do:
    foreach cross section do a sweep:
      foreach point in a cross section do:
        compute force interaction with neighbors
```

and the actual main loop code introduces an artificial traversal order:

```
for (i=1; i<=N; i++) {
  for (j=1; j<=N; j++) {
    S1: temp = A[i][j];
    S2: A[i][j] = 0.2 * (A[i][j]+A[i][j-1]+A[i-1][j]
                        +A[i][j+1]+A[i+1][j]);
    S3: diff += abs(A[i][j] - temp);
  }
}
```

Analyzing the code reveals that the only parallelism opportunity is within anti-diagonals, and the code must be restructured to exploit it. However, the basic algorithm for the computation does not actually specify any particular order in which the elements of the cross section must be updated. The algorithm only specifies that in one sweep, each point in a cross section must be updated once by considering the interaction with its neighbors. Once the algorithm is translated

into code, a specific traversal order is artificially introduced: following column order first, then following row order. One can legitimately question whether any update order will produce acceptable results. If the answer is yes, then we can change the update order of the pixel as follows.

Figure 3.8: LDGs of the original sweep versus after black and red points (shown in grey) are separated.

First, we can assign a color of either black or red to each matrix element. If the row number plus the column number of the element is even, then we color it black, otherwise we color it red. For example, $A[1][4]$ will be red because $1 + 4 = 5$ is odd, while $A[2][2]$ will be black because $2 + 2 = 4$ is even. We can see that there are no direct dependences (direct edges in the LDG) between red and black nodes, although they do have indirect dependences (paths in the LDG)

between them. Since in a single sweep only direct dependences are relevant, the new value of any black node is independent of the old values of all other black nodes. This can be observed from Figure 3.8. Similarly, the new value of any red node is independent of the old values of all other red nodes. Therefore, if we use a loop that makes a sweep over black nodes first, then the loop exhibits a DOALL parallelism. Similarly, if we use a loop that makes a sweep over red nodes, then the loop exhibits a DOALL parallelism. However, updating all black nodes first before updating any red nodes necessitates changing the traversal order of points in the cross section. If the algorithm can tolerate this, as is the case here, then the transformation can be applied, as shown in Code 3.15.

Code 3.15 Red-Black partitioning on the ocean simulation.

```
1 // Black sweep with DOALL parallelism for both outer
2 // and inner loops
3 for (i=1; i<=N; i++) {
4   offset = (i+1) % 2;
5   for (j=1+offset; j<=N; j+=2) {
6     S1: temp = A[i][j];
7     S2: A[i][j] = 0.2 * (A[i][j]+A[i][j-1]+A[i-1][j]
8                          +A[i][j+1]+A[i+1][j]);
9     S3: diff += abs(A[i][j] - temp);
10  }
11 }
12
13 // Red sweep with DOALL parallelism for both outer
14 // and inner loops
15 for (i=1; i<=N; i++) {
16   offset = i % 2;
17   for (j=1+offset; j<=N; j+=2) {
18     S1: temp = A[i][j];
19     S2: A[i][j] = 0.2 * (A[i][j]+A[i][j-1]+A[i-1][j]
20                          +A[i][j+1]+A[i+1][j]);
21     S3: diff += abs(A[i][j] - temp);
22  }
23 }
```

Another algorithm analysis technique that may yield even more parallelism is by analyzing whether the algorithm can tolerate non-deterministic execution. That is, whether it can tolerate dynamic changes in the order of update. If it can, we can simply ignore dependences and execute all iterations of the sweep in parallel, without needing to partition the sweep into black and red sweeps. The

update order will now be timing dependent, in that whichever thread reaches a matrix element, it will update the value of that element using its nearby neighbors regardless of whether the neighbors have been updated or not. One may think that the result will be chaotic. However, the results may not be as chaotic as one may expect because the non-determinism is limited within a single sweep and not across sweeps.

Another example is an algorithm that performs smoothing of an image shown in Code 3.16. The image is greyscale two-dimensional, and each pixel has greyscale intensity that is stored in each element of the `gval` matrix.

Code 3.16 A smoothing filter of a 2-D greyscale image.

```
for (i=1; i<N-1; i++) {
  for (j=1;j<N-1; j++) {
    gval[i][j] = (gval[i-1][j-1]+gval[i-1][j]+gval[i-1][j+1]+
                  gval[i][j-1]+gval[i][j]+gval[i][j+1]+
                  gval[i+1][j-1]+gval[i+1][j]+gval[i+1][j+1])/9;
  }
}
```

Analyzing the code reveals no parallelism opportunity since there are loop-carried dependences across rows, columns, and even anti-diagonals, so here even red-black partitioning cannot be applied. However, the main goal of the smoothing algorithm is to replace each pixel's intensity value by the average of pixel intensities of itself and all neighboring pixels. The ITG introduces artificial serialization in the order in which pixels are updated (i.e., following column order first, then following row order). One can, again, legitimately ask if any update order will produce acceptable results. In this case, the answer is likely to be yes since a slight change in image resulting from smoothing is likely not noticeable. Consequently, we can ignore all dependences and extract DOALL parallelism on both the $for\ i$ loop and the $for\ j$ loop.

3.4 Determining the Scope of Variables

After we identify parallel tasks either through code analysis or algorithm analysis, we have parallel tasks that can execute in parallel with respect to one another. Typically, there are more parallel tasks than the number of available processors, so multiple tasks are often grouped and assigned to a thread. The number of threads

between them. Since in a single sweep only direct dependences are relevant, the new value of any black node is independent of the old values of all other black nodes. This can be observed from Figure 3.8. Similarly, the new value of any red node is independent of the old values of all other red nodes. Therefore, if we use a loop that makes a sweep over black nodes first, then the loop exhibits a DOALL parallelism. Similarly, if we use a loop that makes a sweep over red nodes, then the loop exhibits a DOALL parallelism. However, updating all black nodes first before updating any red nodes necessitates changing the traversal order of points in the cross section. If the algorithm can tolerate this, as is the case here, then the transformation can be applied, as shown in Code 3.15.

Code 3.15 Red-Black partitioning on the ocean simulation.

```
// Black sweep with DOALL parallelism for both outer
// and inner loops
for (i=1; i<=N; i++) {
  offset = (i+1) % 2;
  for (j=1+offset; j<=N; j+=2) {
    S1: temp = A[i][j];
    S2: A[i][j] = 0.2 * (A[i][j]+A[i][j-1]+A[i-1][j]
                         +A[i][j+1]+A[i+1][j]);
    S3: diff += abs(A[i][j] - temp);
  }
}

// Red sweep with DOALL parallelism for both outer
// and inner loops
for (i=1; i<=N; i++) {
  offset = i % 2;
  for (j=1+offset; j<=N; j+=2) {
    S1: temp = A[i][j];
    S2: A[i][j] = 0.2 * (A[i][j]+A[i][j-1]+A[i-1][j]
                         +A[i][j+1]+A[i+1][j]);
    S3: diff += abs(A[i][j] - temp);
  }
}
```

Another algorithm analysis technique that may yield even more parallelism is by analyzing whether the algorithm can tolerate non-deterministic execution. That is, whether it can tolerate dynamic changes in the order of update. If it can, we can simply ignore dependences and execute all iterations of the sweep in parallel, without needing to partition the sweep into black and red sweeps. The

update order will now be timing dependent, in that whichever thread reaches a matrix element, it will update the value of that element using its nearby neighbors regardless of whether the neighbors have been updated or not. One may think that the result will be chaotic. However, the results may not be as chaotic as one may expect because the non-determinism is limited within a single sweep and not across sweeps.

Another example is an algorithm that performs smoothing of an image shown in Code 3.16. The image is greyscale two-dimensional, and each pixel has greyscale intensity that is stored in each element of the `gval` matrix.

Code 3.16 A smoothing filter of a 2-D greyscale image.

```
for (i=1; i<N-1; i++) {
  for (j=1;j<N-1; j++) {
    gval[i][j] = (gval[i-1][j-1]+gval[i-1][j]+gval[i-1][j+1]+
                  gval[i][j-1]+gval[i][j]+gval[i][j+1]+
                  gval[i+1][j-1]+gval[i+1][j]+gval[i+1][j+1])/9;
  }
}
```

Analyzing the code reveals no parallelism opportunity since there are loop-carried dependences across rows, columns, and even anti-diagonals, so here even red-black partitioning cannot be applied. However, the main goal of the smoothing algorithm is to replace each pixel's intensity value by the average of pixel intensities of itself and all neighboring pixels. The ITG introduces artificial serialization in the order in which pixels are updated (i.e., following column order first, then following row order). One can, again, legitimately ask if any update order will produce acceptable results. In this case, the answer is likely to be yes since a slight change in image resulting from smoothing is likely not noticeable. Consequently, we can ignore all dependences and extract DOALL parallelism on both the $for\ i$ loop and the $for\ j$ loop.

3.4 Determining the Scope of Variables

After we identify parallel tasks either through code analysis or algorithm analysis, we have parallel tasks that can execute in parallel with respect to one another. Typically, there are more parallel tasks than the number of available processors, so multiple tasks are often grouped and assigned to a thread. The number of threads

that execute tasks are typically chosen to be the same or less than the number of available processors. In this section, we assume that we have an unlimited number of processors, and each task is assigned to a unique thread.

The next step to perform is what we will refer to as *variable partitioning*, which in essence is a step that determines whether variables should have a *private*-per-thread scope or a *shared*-by-all-threads scope. This step is specific to shared memory programming; in message passing model, all variables are private.

In this step, we need to analyze the use of different variables and categorize them into:

- *Read-Only*: variable is only read by all tasks
- *R/W Non-conflicting*: variable is read, written, or both by only one task
- *R/W Conflicting*: variable written by one task may be read by a different task

Consider the example in Code 3.17. If we define each iteration in the $for\ i$ loop as a parallel task, then read-only variables include `n`, matrix `c`, and matrix `d`, since none of them is modified by any tasks. Read/Write non-conflicting variables include matrices `a` and `b` since the matrix elements modified by each task are disjoint from elements modified by other tasks. For example, iteration $i = 1$ only reads and modifies $b[1][j]$ whereas iteration $i = 2$ only reads and modifies $b[2][j]$, so there is no overlap or conflict. Similarly, iteration $i = 1$ only reads $a[1][j-1]$ which is produced by the same task at the previous j iteration, without overlaps with data read or written by other iterations. Finally, the loop index variables themselves i and j are Read/Write conflicting since they are read and incremented by different tasks.

Code 3.17 A code example.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S2: a[i][j] = b[i][j] + c[i][j];
    S3: b[i][j] = a[i][j-1] * d[i][j];
  }
```

Similarly, consider the following code (Code 3.18). If we define each iteration in the $for\ j$ loop as a parallel task, then read-only variables include `n`, matrix `c`, `i`, and matrix `d`. Note that `i` is a read-only variable because given a $for\ i$ iteration,

none of $for\ j$ iterations modify its value. Read/Write non-conflicting variables include matrices a, b, and e since the matrix elements modified by each task are disjoint from elements modified by other tasks. For example, iteration $j = 1$ only reads and modifies $b[i][1]$ whereas iteration $j = 2$ only reads and modifies $b[i][2]$, so there is no overlap or conflict. Similarly iteration $j = 1$ only reads $a[i-1][j]$ which is produced by the same task at the previous i iteration, without overlaps with data read or written by other iterations. Finally, the loop index variable itself, i.e. j, is Read/Write conflicting since it is read and incremented by different tasks.

Obviously, Read/Write conflicting variables prevent parallelization because they introduce dependences between threads. Therefore, we need techniques to remove such dependences. One of such techniques is *privatization*, through which per-thread copies of a Read/Write conflicting variable are created so that each thread can work solely on its own copy. Another technique is *reduction*, through which per-thread copies of a Read/Write conflicting variable are created so that each thread can produce a partial result in its own copy, and at the end of the parallel section all partial results are combined into a global result. We will now describe these techniques in more details.

Code 3.18 A code example.

```
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++) {
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i-1][j] * d[i][j];
    S3: e[i][j] = a[i][j];
  }
```

3.4.1 Privatization

Read/Write conflicting variables prevent parallelization. So one important concept is called *privatization*. Privatization involves making private copies of a shared variable, so that each thread works on its local copy rather than the shared copy. This removes read/write conflicts and allows parallelization. By default, in shared memory model, any memory location is addressable by all threads. Hence, local copies are created by compiling a single variable into several different variables stored at different memory locations.

Under what situation is a variable privatizable? One situation is when in the original sequential program execution order, a variable is defined (or written) by a

task before being used (or read) by the same task. In this case, a task can write to a private copy of the variable (rather than to the shared copy) and read from that private copy. It does not matter what values other tasks write to the variable since in program order, the value read from the variable is always the same value that was earlier written by the same task.

Another situation in which a variable is privatizable is when a variable is not defined before it is read by the same task, but the values that the task is supposed to read from the variable are known ahead of time. In this case, the local copy of the variable can be defined (or initialized) with the correct value regardless of other tasks.

Privatization can be applied to both *scalar* variables and *arrays* or *matrices*. Of course, when applied to scalar variables, the additional space overhead is small. When applied to arrays or matrices, the additional space overhead and the overhead to initialize them can be substantial since they typically scale with the input size.

As an example, consider the previously-discussed code (Code 3.17), which has Read/Write conflicting variables i and j. Note that here i is incremented in each $for\ i$ iteration, so its value is always known ahead of time. For example, the task that executes the fifth iteration can assume that i should have a value of 5 at the beginning of its execution, without waiting for the task corresponding to the fourth iteration to produce its value. The variable j, on the other hand, is always defined (written to) by each $for\ i$ iteration task when it enters the inner loop. Hence, variable j can also be privatized. The result of privatization is that each task has private copies of both variables i and j. One way to think of private copies is to think that the scalar variables are replaced with references into arrays of variables with the task IDs as the index. For example a task with an ID of x accesses $i[x]$ and $j[x]$, hence it does not conflict with private copies of other tasks since all tasks have unique IDs.

Another example is in Code 3.18, in which each $for\ j$ iteration is a parallel task. Here, the Read/Write conflicting variable is j. j is incremented in each $for\ j$ iteration, so its value is always known ahead of time for any tasks. For example, the task that executes the fifth iteration can assume that j should have a value of 5, without waiting for the fourth iteration to produce its value. Hence, j is privatizable.

Privatization eliminates inter-thread dependence because each thread works on its own copy of a variable. Since no thread needs to read a value that is produced

by another, the threads become independent and can execute in parallel. However, in cases when a Read/Write conflicting variable cannot be privatized or reduced (discussed in the next section), its access must be guarded by synchronization in order to ensure that a thread that consumes the value of the variable waits until the value has been produced by another thread.

3.4.2 Reduction Variables and Operation

A technique related to privatization is *reduction*. Reduction is a type of computation in which the result of the computation can be composed of partial results computed by parallel tasks. The parallelism strategy for reduction operation involves having each parallel task compute a partial result, and at the end the partial results of all tasks are combined to form the final result.

What operations are amenable to reduction? One basic requirement is that a reduction should reduce elements of some array or matrix down to one scalar value. The second requirement is that the final result must be composable from partial results computed by all tasks. Operations performed on all array elements that are *commutative* and *associative* satisfy these requirements. For example, let us say that we want to add up all array elements into its sum. This satisfies the first requirement. We can divide the work by having each thread summing up a part of the array and storing the partial sum into its private variable. In the end, we can have the master thread add all partial sums into the final sum. The final sum will be correct because addition is commutative and associative, which implies that regardless of the order of summation performed across threads, when we sum up the partial sums, we get the right global sum value.

More specifically, suppose that the computation is the following: $y = y_{init} \circ a[0] \circ a[1] \circ \ldots \circ a[n]$. An operator $\circ$ is commutative if $u \circ v = v \circ u$, and is associative if $(u \circ v) \circ w = u \circ (v \circ w)$. We refer to $\circ$ as the *reduction operator* and y as the reduction variable. From here, we can derive that reduction operators include sum (addition of elements), product (multiplication of elements), maximum, minimum, and logical operations (and, or, xor, etc.). In OpenMP, programmers can specify the reduction operator and variable to the compiler, which will break down the computation among threads into partial results and combine them into the final result. However, there are operators other than addition, multiplication, and logical operations that cannot be expressed in OpenMP but exhibit commutativity and associativity. Such operators require programmers to modify their own code to implement reduction. An example of that is shown in Example 3.3.

■ **Example 3.3** Example of Exploiting Reduction.

Consider the following code:

```
sum = 0;
for (i=0; i<8; i++) {
   a[i] = b[i] + c[i];
   if (a[i] > 0)
      sum = sum + a[i];
}
Print sum;
```

For the code above, manually exploiting reduction involves rewriting the code as:

```
begin parallel // spawn a child thread
private int start_iter, end_iter, i;
shared int local_iter = 4, sum = 0;
shared double sum = 0.0, a[], b[], c[], localSum[];

sum = 0;
start_iter = getid() * local_iter;
end_iter = start_iter + local_iter;
for (i=start_iter; i<end_iter; i++)
  a[i] = b[i] + c[i];
  if (a[i] > 0)
     localSum[id] = localSum[id] + a[i];
barrier;
end parallel // kill the child thread

for (k=0; k<getNumProc(); k++)
  sum += localSum[k];
Print sum;
```

Reduction allows parallel execution of threads as it removes data dependences between them due to Read/Write conflicting variables. It allows each thread to store a partial result in its own copy of a variable. In the end, the main thread can gather all partial results and generate the full (global) result.

3.4.3 Summary of Criteria

A variable that is read-only should be declared as shared for performance reasons. The program is still correct if a read-only variable is declared as private and is initialized to a correct value. However, this comes at the expense of unnecessary

storage overheads.

A variable that is Read/Write non-conflicting should also be declared as shared for performance reasons. Similar to the prior case, privatizing a Read/Write non-conflicting variable introduces unnecessary storage overheads.

For a variable that is Read/Write Conflicting, care must be taken. Normally, such variable needs to be declared as shared but accesses to it must be protected by a critical section. Furthermore, point-to-point synchronization may be needed if the correctness of program execution depends on the proper sequencing of instructions. Note that critical section is expensive since it serializes accesses to a shared variable and implementations of locks are often not scalable to a high number of processors. Hence, as much as possible, critical section should to be avoided. The most straightforward strategy is to check whether the variable satisfies the criteria for a reduction variable. If it does, it should be made a reduction variable and the operation that is performed on it should be treated as a reduction operation. If it is not a reduction variable, the next step is to check whether it is privatizable. If it is privatizable, then it should be privatized if it is profitable to do so, i.e. the performance impact of extra storage overheads is tolerable and is smaller than the performance impact from having critical section.

Another alternative is to re-analyze the code and choose a different parallel region. It is often the case that a Read/Write Conflicting variable may change into a Read/Write non-conflicting variable when the scope of parallelism changes. For example, consider the following code (Code 3.19) which shows multiplication of matrices A and B and the result is stored in matrix Y. If the parallel region is the $for\ k$ loop (i.e. the parallel tasks are $for\ k$ iterations), $Y[i][j]$ is a Read/Write conflicting variable since different tasks may read and write to it simultaneously. So it needs to protected using a critical section. However, if the parallel region is changed to, say, the $for\ i$ loop, then for any i and j, $Y[i][j]$ is only written to or read from by a single task. The variable becomes Read/Write non-conflicting, and there is no longer a need to protect its access with a critical section.

3.5 Synchronization

In the shared memory programming model, synchronization acts as a mechanism through which programmers control the sequence of operations that are performed by parallel threads. Note that synchronization is performed among threads, rather than among tasks. So, at this step we have assumed that tasks have been assigned

Code 3.19 A matrix multiplication code example.

```
for (i=0; i<N; i++) {
  for (j=0; j<N; j++) {
    Y[i][j] = 0.0;
    for (k=0; k<N; k++) {
      Y[i][j] = Y[i][j] + A[i][k] * B[k][j];
    }
  }
}
```

to threads. For simplicity, however, we assume that there are as many threads as tasks, such that the mapping of a task to a thread is one-to-one.

Three types of synchronization primitives are in wide use. The first one is *point-to-point* synchronization involving two parallel tasks. The example is *post* and *wait* that we used while describing DOACROSS and DOPIPE parallelism. A post operation acts like depositing a token signifying that data is produced, and a wait operation blocks until there is a token deposited, preventing the consumer from continuing, until data is ready. Post and wait are often referred to as *semaphore* synchronization primitives.

The second type of popular synchronization is a *lock*. A lock can be obtained by only one parallel task, and once the lock is held by that task, other tasks cannot obtain it until the current task releases the lock. The two operations that can be performed on a lock are acquiring the lock (`lock(name)`) and releasing the lock (`unlock(name)`). Therefore, a lock is useful to form a *critical section*, which is a region of code that only allows at most one thread to execute it at any time. A critical section is useful for ensuring that only one thread at a time access a Read/Write conflicting variable that is not privatizable and for which reduction cannot be applied.

The third type of popular synchronization is *barrier*. A barrier defines a point that a thread is allowed to go past only when all threads have reached it (illustrated in Figure 3.9). In the figure, there are four threads reaching the barrier points at different time. Threads 1, 3, and 4 have to wait inside the barrier until the last thread (Thread 2) reaches it. Only then, can they execute code after the barrier. This example illustrates that a barrier is simple and easy to use, it makes the total execution time of the parallel execution dependent on the execution time of the slowest thread. Hence, when using barriers, load balance is very important. The efficiency of the barrier implementation is also a critical goal in designing parallel

computers.

Figure 3.9: Illustration of a barrier.

A barrier should be inserted if the code section following it uses values that are written by the parallel code section preceding it. For example, numerical simulations often work in time-steps where in one time-step the main data structure (matrices) are updated. Barriers are a logical synchronization to use between these time-steps. They ensure that before going to the next time-step, all threads have completed computation for the current time-step.

3.6 Mapping Tasks to Processors and Memory

Task mapping deals with two aspects. The first aspect is how to map tasks to threads. Often, there are more tasks than available processors, so one question is that what tasks should be grouped together into a thread. The second aspect is how to map threads to processors, so as to make sure that the communicating processors are as close to each other as possible.

One consideration in task mapping is whether to map tasks to threads *statically* or *dynamically*. Static task mapping means that tasks are pre-assigned to threads before execution. For example, this is achieved by specifying which iterations of a DOALL loop are to be executed by a thread. Dynamic task mapping means that tasks are not assigned to threads prior to execution. During execution, whichever thread is idle grabs a task (or a group of tasks) from the task queue and executes it. Dynamic task mapping introduces extra overheads for the task queue

management, but sometimes makes it is easier to ensure that all threads' loads are balanced.

From the perspective of iteration space, contiguous blocks of iterations can be grouped into threads (referred to as *block partitioning*), or iterations may be assigned in a round-robin manner to threads (referred to as *cyclic partitioning*).

3.7 A Brief Introduction to OpenMP

This section briefly introduces the OpenMP standard version 2.0. More comprehensive information can be found at `http://www.openmp.org`. A longer introduction to OpenMP can be found at `http://en.wikipedia.org/wiki/OpenMP`. In this book we only cover the basics of OpenMP related to expressing various parallelism concepts that were covered earlier.

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports shared memory programming. OpenMP is jointly standardized by a group of major computer hardware and software vendors. It currently supports C/C++ and Fortran languages. The OpenMP standard consists of a set of compiler directives with which programmers can express parallelism to a compiler that supports OpenMP. The compiler replaces the directives into code that calls library routines or reads environment variables that influence the run-time behavior of the program.

The directive in OpenMP follows the following format:

```
#pragma omp directive-name [clause[ [,] clause]...]  new-line
```

To express a DOALL parallelism for a loop, we can insert the following directive right above the loop.

```
#pragma omp for [clause[[,] clause] ...  ]  new-line
```

where a `clause` is one the following:

- `private(variable-list)`
- `firstprivate(variable-list)`
- `lastprivate(variable-list)`
- `reduction(operator:  variable-list)`
- `ordered`
- `schedule(kind[, chunk_size])`

- `nowait`

To express a *for* loop that has DOALL parallelism, we can insert the following directives to a code (Code 3.20). The source file must include the header file `omp.h` which contains OpenMP function declarations. A parallel region is started by `#pragma omp parallel` and is enclosed by curly brackets. The start of a parallel region will spawn threads, and the number of threads spawned can be specified by an environment variable (e.g., `setenv OMP_NUM_THREADS n`) or directly in the code by calling a function (e.g., `omp_set_num_threads(n)`). The code inside the brackets are executed by all threads. Within a parallel region, we can have multiple parallel loops and each of the loop does not spawn new threads as they have been created when entering the parallel region. If we only have one loop that we want to execute in parallel, we can combine the start of the parallel loop and the start of the parallel region with one directive `#pragma omp parallel for`.

Code 3.20 A DOALL parallelism example in OpenMP.

```
1 #include <omp.h>
2 ...
3 #pragma omp parallel  // start of parallel region
4 {
5   ...
6   #pragma omp parallel for default(shared)
7   for(i=0; i<n; i++)
8     A[i]= A[i]*A[i]- 3.0;
9 } // end of parallel region
```

The code shows a *for i* loop that has no loop-carried dependences. With OpenMP, we only need to insert the `pragma` directive `parallel for`. The `default(shared)` clause states that unless otherwise specified, all variables within the scope of the loop are shared. Note that the loop index variable `i` needs to be private per thread, to avoid conflicting updates by different threads. However, the `private` declaration for it is usually unnecessary as OpenMP compilers automatically privatize loop index variables.

To express function parallelism in which we want to execute some code section in parallel with another code section, we can use the following construct (Code 3.21). The code shows that the two loops should be executed in parallel with respect to one another, although each loop is sequentially executed.

Code 3.21 A function parallelism example in OpenMP.

```
1 #pragma omp parallel shared(A,B)private(i)
2 {
3   #pragma omp sections nowait
4   {
5     #pragma omp section
6     for(i=0; i<n; i++)
7       A[i]= A[i]*A[i]- 4.0;
8     #pragma omp section
9     for(i=0; i<n; i++)
10      B[i]= B[i]*B[i] + 9.0;
11  } // end omp sections
12 } // end omp parallel
```

In OpenMP, a variable can have a type of `shared`, `private`, `reduction`, `firstprivate`, or `lastprivate`. `shared` and `private` are variable scopes that we have discussed in Section 3.4. Note that if a variable exists outside the parallel region and has the same name as the variable inside the parallel region and the variable is declared private, then they do not share any storage, meaning that a write to them will write to different locations in memory.

`reduction` variable is a privatized variable that are combined at the end of the parallel section to produce a single result value, and has been discussed in Section 3.4.1. `firstprivate` and `lastprivate` are special types of private variables. `firstprivate` initializes the private copies of a variable from the value of the original variable prior to entering the parallel region. `lastprivate` means that upon loop exit, the value of the original copy of the variable is assigned the value seen by the thread that is assigned the last loop iteration (for parallel loops only).

For synchronization primitives, OpenMP implicitly inserts a barrier after each parallel loop or parallel sections. If programmers find that no barrier is necessary, he can insert a `nowait` clause. For example, in the following code, any thread that finishes execution of its tasks is allowed to move on to the next section of code (Code 3.22):

Code 3.22 Removing implicit barriers in OpenMP.

```
1 #pragma omp parallel for nowait default(shared) private(i)
2 for(i=0; i<n; i++)
3   A[i]= A[i]*A[i]- 3.0;
4 ...
```

In OpenMP, we can also indicate a critical section with the following directive: `#pragma omp critical` and enclose in curly brackets the code that is inside the critical sections. This makes it unnecessary for programmers to declare lock variables, and to use a pair of lock and unlock. However, OpenMP also allows named lock variable.

It is also possible to specify that only a single thread should execute a portion of the code, using the directive `#pragma omp single` and enclosing in curly brackets the code that is to be executed by just one thread. If that single thread needs to be the master thread, then the directive to use is `#pragma omp master`.

■ *Did you know?*

While OpenMP 2.0 can express DOALL parallelism for loops well, it has several major limitations:

- *Anti-diagonal traversal cannot be specified through OpenMP. Programmers must restructure a loop first to achieve such traversal before using OpenMP directives.*
- *Partial DOALL parallelism that exists in loop cannot be expressed through OpenMP directives. Programmers must first distribute the loop into smaller loops and use OpenMP directives for the smaller loops that exhibit full DOALL parallelism.*
- *DOACROSS and DOPIPE parallelism cannot be specified through OpenMP directives, and point-to-point synchronization required by them also cannot be specified through OpenMP directives.*
- *Reduction operators are limited in OpenMP to only simple ones that are provably commutative and associative. They include addition, multiplication, logical operations, and in Fortran, maximum and minimum.*

Hopefully the limitations will be addressed in future OpenMP standards.

When assigning iterations, we can also specify that the iterations should be executed in program order using the clause `ordered`. However, this should not be used unless there are reasons to do this as it is an expensive construct that introduces serialization.

The `schedule` clause controls how iterations of a loop are assigned to threads and it has two parameters: type of scheduling, and *chunk* size. A chunk is a group

of consecutive iterations. The type of scheduling is one of the following:

- `static`: each chunk is assigned to a processor statically.
- `dynamic`: each chunk is placed in a task queue. Each processor, when it is idle or when it finishes computing a task, grabs the next one in the task queue and executes it.
- `guided`: the same as dynamic, except that the task sizes are not uniform, early tasks are exponentially larger than later tasks.
- `runtime`: the choice of scheduling is not determined statically. Rather, the OpenMP runtime library checks the environment variable `OMP_SCHEDULE` at run time to determine what scheduling to use.

3.8 Example: Matrix Multiplication Parallelization

This section discusses an example in parallelizing the matrix multiplication program. It applies the concepts that were earlier presented, and shows the performance implication of different parallelization strategies. Suppose that there are two 2-dimensional rectangular matrices `A` and `B` that we want to multiply together, and the result is stored in the third matrix `Y`. The code for the multiplication is shown below (Code 3.23).

Code 3.23 Matrix multiplication code.

```
1 #define MATRIX_DIM 500
2
3 double A[MATRIX_DIM][MATRIX_DIM],
4   B[MATRIX_DIM][MATRIX_DIM],
5   Y[MATRIX_DIM][MATRIX_DIM];
6 int i, j, k;
7 . . .
8
9     for (i=0; i<MATRIX_DIM; i++) {
10      for (j=0; j<MATRIX_DIM; j++) {
11        Y[i][j] = 0.0;   // S1
12        for (k=0; k<MATRIX_DIM; k++) {
13          Y[i][j] = Y[i][j] + A[i][k] * B[k][j]; // S2
14        } // for k
15      } // for j
16    } // for i
17 . . .
```

From the code, we can see a loopnest consisting of three loops. The two outer loops iterate over each dimension of the matrix. Each iteration in the second loop assigns a value to one element of the result matrix `Y`. The innermost loop visits a row in matrix `A` and a column in matrix `B`, and their sum of products is accumulated to the current element in matrix `Y`.

Applying dependence analysis of the loopnest, we find that the innermost statement first reads the last value of $Y[i][j]$ set by the previous k iteration, then adds it with $A[i][k] \times B[k][j]$, and writes the new value to $Y[i][j]$. The new value will be read by the same statement in the next k iteration. Therefore, the we have a true dependence $S2[i][j][k] \rightarrow^T S2[i][j][k+1]$, where $S2$ represents the innermost statement. There are no other dependences, except between $S1$ and $S2$ for the first k iteration. The true dependence is loop-carried for the $for\ k$ loop but is loop independent for both $for\ i$ and $for\ j$ loops. Therefore, both the $for\ i$ and $for\ j$ loops exhibit DOALL parallelism.

Moving on to determining the scope of variables, we can see from the code that matrices A and B are never written, so they are read-only variables. Loop index variables i, j, k are read-write conflicting but are privatizable since we know their values for all iterations beforehand. Finally, consider elements of array Y. If the parallel tasks are iterations of the $for\ i$ loop, elements of array Y are read/write non-conflicting variables since each parallel task reads from and writes to different rows of the matrix. Likewise, if the parallel tasks are iterations of the $for\ j$ loop, elements of array Y are read/write non-conflicting variables since each parallel task reads from and writes to different columns of the matrix, for a single row. Hence, for both loops, all variables should be declared shared except the loop index variables. Finally, if the parallel tasks are iterations of the $for\ k$ loop, elements of array Y are read/write conflicting, because each k iteration reads from the value written by the previous k iteration. Examining the operation performed on Y, it is a summation of the products of elements from a row of matrix A and a column of matrix B. Since an addition operation is commutative and associative, it is a reduction operator, and each element of array Y can be made a reduction variable. All other variables (except the loop index variables) can be declared as shared.

Using OpenMP, if we want to parallelize the $for\ i$ loop only, we can place the OpenMP directive `#pragma omp parallel for default (shared)` right before the $for\ i$ loop to parallelize the $for\ i$ loop, or right before the $for\ j$ loop to parallelize the $for\ j$ loop, or at both places to parallelize both

of consecutive iterations. The type of scheduling is one of the following:

- `static`: each chunk is assigned to a processor statically.
- `dynamic`: each chunk is placed in a task queue. Each processor, when it is idle or when it finishes computing a task, grabs the next one in the task queue and executes it.
- `guided`: the same as dynamic, except that the task sizes are not uniform, early tasks are exponentially larger than later tasks.
- `runtime`: the choice of scheduling is not determined statically. Rather, the OpenMP runtime library checks the environment variable `OMP_SCHEDULE` at run time to determine what scheduling to use.

3.8 Example: Matrix Multiplication Parallelization

This section discusses an example in parallelizing the matrix multiplication program. It applies the concepts that were earlier presented, and shows the performance implication of different parallelization strategies. Suppose that there are two 2-dimensional rectangular matrices `A` and `B` that we want to multiply together, and the result is stored in the third matrix `Y`. The code for the multiplication is shown below (Code 3.23).

Code 3.23 Matrix multiplication code.

```
1 #define MATRIX_DIM 500
2
3 double A[MATRIX_DIM][MATRIX_DIM],
4   B[MATRIX_DIM][MATRIX_DIM],
5   Y[MATRIX_DIM][MATRIX_DIM];
6 int i, j, k;
7 . . .
8
9     for (i=0; i<MATRIX_DIM; i++) {
10      for (j=0; j<MATRIX_DIM; j++) {
11        Y[i][j] = 0.0;  // S1
12        for (k=0; k<MATRIX_DIM; k++) {
13          Y[i][j] = Y[i][j] + A[i][k] * B[k][j]; // S2
14        } // for k
15      } // for j
16    } // for i
17 . . .
```

From the code, we can see a loopnest consisting of three loops. The two outer loops iterate over each dimension of the matrix. Each iteration in the second loop assigns a value to one element of the result matrix `Y`. The innermost loop visits a row in matrix `A` and a column in matrix `B`, and their sum of products is accumulated to the current element in matrix `Y`.

Applying dependence analysis of the loopnest, we find that the innermost statement first reads the last value of $Y[i][j]$ set by the previous k iteration, then adds it with $A[i][k] \times B[k][j]$, and writes the new value to $Y[i][j]$. The new value will be read by the same statement in the next k iteration. Therefore, the we have a true dependence $S2[i][j][k] \rightarrow^T S2[i][j][k+1]$, where $S2$ represents the innermost statement. There are no other dependences, except between $S1$ and $S2$ for the first k iteration. The true dependence is loop-carried for the $for\ k$ loop but is loop independent for both $for\ i$ and $for\ j$ loops. Therefore, both the $for\ i$ and $for\ j$ loops exhibit DOALL parallelism.

Moving on to determining the scope of variables, we can see from the code that matrices A and B are never written, so they are read-only variables. Loop index variables i, j, k are read-write conflicting but are privatizable since we know their values for all iterations beforehand. Finally, consider elements of array Y. If the parallel tasks are iterations of the $for\ i$ loop, elements of array Y are read/write non-conflicting variables since each parallel task reads from and writes to different rows of the matrix. Likewise, if the parallel tasks are iterations of the $for\ j$ loop, elements of array Y are read/write non-conflicting variables since each parallel task reads from and writes to different columns of the matrix, for a single row. Hence, for both loops, all variables should be declared shared except the loop index variables. Finally, if the parallel tasks are iterations of the $for\ k$ loop, elements of array Y are read/write conflicting, because each k iteration reads from the value written by the previous k iteration. Examining the operation performed on Y, it is a summation of the products of elements from a row of matrix A and a column of matrix B. Since an addition operation is commutative and associative, it is a reduction operator, and each element of array Y can be made a reduction variable. All other variables (except the loop index variables) can be declared as shared.

Using OpenMP, if we want to parallelize the $for\ i$ loop only, we can place the OpenMP directive `#pragma omp parallel for default (shared)` right before the $for\ i$ loop to parallelize the $for\ i$ loop, or right before the $for\ j$ loop to parallelize the $for\ j$ loop, or at both places to parallelize both

loops. If instead we want to parallelize the *for k* loop, we can put the OpenMP directive `#pragma omp parallel for default(shared)` right before the *for k* loop, but in addition, we either have to guard the access to $Y[i][j]$ with `#pragma omp critical` (Code 3.24), or transform $Y[i][j]$ into a reduction variable. Unfortunately, the latter choice cannot be implemented using the OpenMP reduction clause because the reduction variable must be scalar, and $Y[i][j]$ is a part of a matrix.

Code 3.24 Matrix multiplication code.

```
1 . . .
2     for (i=0; i<MATRIX_DIM; i++) {
3       for (j=0; j<MATRIX_DIM; j++) {
4         Y[i][j] = 0.0;
5 #pragma omp parallel for default(shared)
6         for (k=0; k<MATRIX_DIM; k++) {
7 #pragma omp critical {
8             Y[i][j] = Y[i][j] + A[i][k] * B[k][j];
9           } // critical
10        } // for k
11      } // for j
12    } // for i
13 . . .
```

The resulting parallel program was compiled using Intel icc compiler with `-O3` optimization level on a workstation having two Intel Xeon CPU running at 2.0 GHz, with each core capable of running two threads simultaneously (2-way hyperthreaded). Each CPU has a 512KB L2 cache. The matrix multiplication is executed 10 times.

Figure 3.10 shows the speedup ratios by using different parallelization strategies: parallelizing only the *for i* loop, only the *for j* loop, only the *for k* loop with reduction, and only the *for k* loop with critical section guarding the innermost statement. The sequential execution time is 20 seconds. Parallelizing the *for i* loop gives an execution time of 9.47 seconds, yielding a speedup ratio of 2.11. This is not surprising because while all four threads can run simultaneously, the threads that share processor resources are slowed down due to contention. Hence, a perfect speedup ratio of 4 cannot be approximated. Parallelizing the *for j* loop gives an execution time of 10.13 seconds, yielding a speedup ratio of 1.97. The reason for the slight increase in execution time is because while in the *for i* loop only one parallel region is encountered, in the *for j* loop paralleliza-

tion, each $for\ i$ iteration corresponds to one parallel region. Since there are a total of 500 parallel regions, the overhead for setting up a new parallel region, entering and exiting it is approximately $\frac{10.13-9.47}{500} \times 1000 = 1.3$ milliseconds or less. Parallelizing the $for\ k$ loop with reduction loop slows it down further to 101.32 seconds, due to the overheads of a lot more parallel regions. Finally, parallelizing the $for\ k$ loop with a critical section guarding the innermost statement requires more than 720 seconds to execute.

Figure 3.10: Speedup ratio from using different parallelization strategies

Overall, the evaluation shows that the granularity of parallel regions is very important because parallel overheads can be quite significant. A higher priority should be given for parallelizing the outermost loop in a loopnest compared to parallelizing inner loops. In addition, critical sections that are frequently encountered can greatly degrade performance.

3.9 Exercise Problems

1. (20 *points*) **Parallel Programming Models**. Consider a linear transformation algorithm to compute $Y = A \times B + C$, where Y, A, B, C have a dimension $n \times p, n \times m, m \times p, n \times p$, respectively. Assume that all n, m, and p are divisible by 2. The algorithm is shown:

```
int i, j, k;
float A[n][m], B[m][p], Y[n][p], C[n][p], x;

...
// begin linear transformation
for (i=0; i<n; i++) {
  for (j=0; j<p; j++) {
    x = 0;
    for (k=0; k<m; k++)
      x = x + A[i][k] * B[k][j];
    Y[i][j] = x + C[i][j];
  }
}
```

As can be seen from the code, all the loops (for i, for j, for k) are parallel.

(a) For shared memory programming, show the appropriate and correct OpenMP directive to parallelize the "for j" loop only. Indicate where you would place the directive.

(b) For shared memory programming, show the appropriate and correct OpenMP directive to parallelize the "for k" loop only. Indicate where you would place the directive.

(c) For message passing model, parallelize the "for j" loop only. Show the resulting parallel code utilizing `send(destination-id,data)` and `recv(sender-id,data)` on two processors. Only send and receive data that is absolutely needed to be communicated. Enclose your parallel region with `begin_parallel` and `end_parallel`.

2. (15 *points*) **Code Analysis**. For the code shown below:

```
...
for (i=1; i<=N; i++) {
```

```
  for (j=1; j<=i; j++) {  // note the index range!
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i][j-1];
    S3: c[i][j] = a[i][j];
  }
}
```

(a) Draw its Iteration-space Traversal Graph (ITG)

(b) List all the dependences, and clearly indicate which dependence is loop-independent vs. loop-carried.

(c) Draw its Loop-carried Dependence Graph (LDG).

3. (15 *points*) **Code Analysis**. For the code shown below:

```
...
for (i=1; i<=N; i++) {
  for (j=2; j<=N; j++) {
    S1: a[i][j] = a[i][j-1] + a[i][j-2];
    S2: a[i+1][j] = a[i][j] * b[i-1][j];
    S3: b[i][j] = a[i][j];
  }
}
```

(a) Draw its Iteration-space Traversal Graph (ITG)

(b) List all the dependences, and clearly indicate which dependence is loop-independent vs. loop-carried.

(c) Draw its Loop-carried Dependence Graph (LDG).

4. (15 *points*) **Code Analysis**. For the code shown below:

```
...
for (i=1; i<=N; i++) {
  for (j=1; j<=i; j++) {  // note the index range!
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i-1][j-1];
    S3: c[i][j] = a[i][j];
  }
}
```

(a) Draw its Iteration-space Traversal Graph (ITG)

(b) List all the dependences, and clearly indicate which dependence is loop-independent vs. loop-carried.

(c) Draw its Loop-carried Dependence Graph (LDG).

5. (15 *points*) **Code Analysis**. For the code shown below:

```
...
for (i=1; i<=N; i++) {
  for (j=1; j<=i; j++) {  // note the index range!
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i-1][j-1] * b[i+1][j-1]
          * c[i-1][j];
    S3: c[i+1][j] = a[i][j];
  }
}
```

(a) Draw its Iteration-space Traversal Graph (ITG)

(b) List all the dependences, and clearly indicate which dependence is loop-independent vs. loop-carried.

(c) Draw its Loop-carried Dependence Graph (LDG).

6. (30 *points*) **Other Parallelism**. For the code shown below:

```
...
for (i=1; i<=N; i++) {
  for (j=1; j<=i; j++) {  // note the index range!
    S1: a[i][j] = b[i][j] + c[i][j];
    S2: b[i][j] = a[i-1][j-1];
    S3: c[i][j] = a[i][j];
    S4: d[i][j] = d[i][j-1] + 1;
  }
}
```

(a) Show the code that exploits function parallelism.

(b) If S4 is removed from the loop, exploits DOACROSS parallelism to the fullest, and show the code.

(c) If S4 is removed from the loop, Exploits DOPIPE parallelism to the fullest, and show the code.

7. (20 *points*) Consider a linear transformation algorithm to compute $Y = A \times B + C$, where Y, A, B, C have a dimension $n \times p, n \times m, m \times p, n \times p$,

respectively. Assume that all n, m, and p are divisible by 2. The algorithm is shown:

```
int i, j, k;
float A[n][m], B[m][p], Y[n][p], C[n][p], x;

...
// begin linear transformation
for (i=0; i<n; i++) {
  for (j=0; j<p; j++) {
    x = 0;
    for (k=0; k<m; k++)
      x = x + A[i][k] * B[k][j];
    Y[i][j] = x + C[i][j];
  }
}
```

As can be seen from the code, all the loops (for i, for j, for k) are parallel.

(a) Indicate each variable (i,j,k,A,B,C,Y,x) whether it should be declared private or shared variable when we parallelize:

   i. for i loop only.

   ii. for j loop only.

   iii. for k loop only.

(b) Parallelize the algorithm for the "for i" loop only, using the message passing programming model on 2 processors. Use *send(destination, data)* and *recv(source, data)* as your primitives similar to the example given in the class.

(c) Without using the OpenMP directives, parallelize the algorithm for the "for i" loop only, using the shared memory programming model on 2 processors. Use "begin parallel" and "end parallel" to mark your parallel region, and insert proper synchronization (e.g., locks and barriers). Remove unnecessary serialization, if applicable.

(d) Without using the OpenMP directives, parallelize the algorithm for the "for k" loop only, using the shared memory programming model on 2 processors, without using OpenMP directives. Use "begin parallel" and "end parallel" to mark your parallel region, and insert proper synchronization (e.g., locks and barriers). Remove unnecessary serialization, if applicable.

8. (15 *points*) Consider a linear transformation algorithm to compute $Y = A \times B + C$, where Y, A, B, C have a dimension $n \times p, n \times m, m \times p, n \times p$, respectively. Assume that all n, m, and p are divisible by 2. The algorithm is shown:

```
int i, j, k;
float A[n][m], B[m][p], Y[n][p], C[n][p], x;

...
// begin linear transformation
for (i=0; i<n; i++) {
  for (j=0; j<p; j++) {
    x = 0;
    for (k=0; k<m; k++)
      x = x + A[i][k] * B[k][j];
    Y[i][j] = x + C[i][j];
  }
}
```

As can be seen from the code, all the loops (for i, for j, for k) are parallel. Indicate each variable (i,j,k,A,B,C,Y,x) whether it should be declared private or shared variable when we parallelize:

(a) for i loop only.

Variable	Private	Shared
i		
j		
k		
A		
B		
C		
Y		
x		

(b) for j loop only.

Variable	Private	Shared
i		
j		
k		
A		
B		
C		
Y		
x		

(c) for k loop only.

Variable	Private	Shared
i		
j		
k		
A		
B		
C		
Y		
x		

9. (*10 points*)**Reduction Operation**. For the following code, parallelize the code by identifying the reduction variable and reduction operation. When necessary, transform non-reduction operation into a reduction operation by rewriting the code (show the new code!).

```
// pre: x and y contain a value > 1
for (i=0; i<n; i++)
  y = y * exp(x,A[i]);
print y;
```

where $exp(a, b) = a^b$.

10. (*20 Points*) Ocean parallelization. Given the code for a single sweep in ocean (without the diff computation):

```
for (i=1; i<=n; i++) {
  for (j=1; j<=n; j++) {
    A[i][j] = 0.2 * (A[i][j] + A[i][j-1] + A[i][j+1]
```

```
                    + A[i-1][j] + A[i+1][j]);
    }
  }
```

(a) Rewrite the code so that in the new loop nest, each iteration of one of the loops is a parallel task. Exploit parallelism that exists in each anti-diagonal. Then, highlight the loop that is parallel in your new code.

(b) Rewrite the code to exploit parallelism using red-black partitioning. Idenitfy the parallel loops.

11. (*20 Points*) **Code Analysis**. For the code shown below:

```
for (i=4; i<=n; i++)
  A[i] = A[i] + A[i-4];
```

(a) Draw the loop carried dependence graph

(b) By analyzing the loop carried dependence graph, identify the parallelism in the code, rewrite it and express the parallel loop(s) by using OpenMP parallel for directive.

12. This following loop is taken from tomcatv of Spec benchmark, translated into an equivalent C version:

```
for (j=2; j<n; j++) {
  for (i=2; i<n; i++) {
    x[i][j] = x[i][j] + rx[i][j];
    y[i][j] = y[i][j] + ry[i][j];
  }
}
```

Assume that n is a multiple of 2. Using OpenMP directives, show parallel version of the loop that exploit:

(a) Function parallelism only

(b) Data parallelism only

(c) Both function and data parallelism

Chapter 4

Issues in Shared Memory Programming

Contents

4.1 Common Correctness Pitfalls **76**
- 4.1.1 Result Preservation 76
- 4.1.2 Incorrect or Missing Synchronization 78
- 4.1.3 Incorrect Variable Scope 80

4.2 Parallelizing Compiler Limitations **81**
- 4.2.1 A Case Study on the MIPSpro Compiler 81

4.3 Performance Considerations **87**
- 4.3.1 Amdahl's Law 87
- 4.3.2 Parallel Thread Granularity 90
- 4.3.3 Synchronization Granularity 91
- 4.3.4 Inherent and Artifactual Communication 92
- 4.3.5 Scheduling and Load Balancing 96
- 4.3.6 Memory Hierarchy Considerations 99

4.4 Exercise Problems **106**

In the previous chapter, we have looked at shared memory parallel programming techniques. The discussion focused on how correct shared memory parallel programs can be constructed. The objective of this chapter is to fill in the coverage by discussing practical issues that come up when creating and tuning shared memory parallel programs. Section 4.1 covers common correctness pitfalls in shared

memory programming, Section 4.2 discusses the limitation of how much parallelism an automatic parallelizing compiler can identify, and Section 4.3 covers performance and scalability considerations for shared memory programming.

4.1 Common Correctness Pitfalls

4.1.1 Result Preservation

One thing that programmers must be aware of when creating parallel programs is that parallel execution changes the order of operations, and such change may affect the output of the program compared to that of the sequential execution of the program. One reason for why this may be so is due to the rounding errors that occur in floating point number representations.

For example, suppose that we have three floating point numbers to add: A, B, and C. A has a decimal value of 1.3125, while B and C have a decimal value of 0.03125. Suppose that we have a floating point number format that represents the sign with one bit, exponent with four bits, and mantissa with four bits. The exponent uses biased-7 representation, i.e., `0010` represents 2^{2-7}. The mantissa shows the normalized number, with a "1" at the hidden most significant bit (which is not recorded in the mantissa bits). We want to sum A, B, and C. The resulting sum depends on the order of addition due to the insufficient accuracy in the number representation. This is illustrated in Figure 4.1. Hidden bits are shown in parentheses.

A has an exponent of 0111, and a mantissa of 0101. Both B and C have an exponent of 0010, and a mantissa 0000. Suppose that in the sequential execution of the program, we add B and C first, then add the result with A. This is shown in the top half of the figure. Floating point addition works in the following three steps. The first step is equalizing the exponents of the numbers that are to be added. The second step is adding the mantissa's of the two numbers. And the final step is normalizing the addition result.

For the example above, when we first add B and C, the result of that addition has a mantissa of 0000 with hidden bits of (10). Since only a "1" is allowed in the hidden bit, the result is normalized by shifting the mantissa by one bit position, while increasing the exponent value by 1. Hence, the result is a number with an exponent of 0011 and mantissa of 0000. When the number is added with A, it needs to have its exponent value equalized to A's exponent. Hence, after exponent equalization, its exponent value becomes 0111 and its mantissa becomes 0001.

Decimal	Normalized Floating Point		
	Sign	Exponent	Mantissa
A= 1.3125	0	0111	(1)0101
B=0.03125	0	0010	(1)0000
C=0.03125	0	0010	(1)0000

Adding B and C first before adding A:

B+C:

(i) Equalize exponent	0	0010	(1)0001	→	0	0010	(1)0000
	0	0010	(1)0000	→	0	0010	(1)0000 +
(ii) Add mantissa					0	0010	(10)0000
(iii) Normalize result					0	0011	(1)0000

(B+C) + A:

(i) Equalize exponent	0	0011	(1)0000	→	0	0111	(0)0001
	0	0111	(1)0101	→	0	0111	(0)0101 +
(ii) Add mantissa					0	0111	(1)0110
(iii) Normalize result					0	0111	(1)0110

Adding A and B first before adding C:

A+B:

(i) Equalize exponent	0	0111	(1)0101	→	0	0111	(1)0101
	0	0010	(1)0000	→	0	0111	(0)0000(1) +
(ii) Add mantissa					0	0111	(1)0101
(iii) Normalize result					0	0111	(1)0101

(A+B) + C:

(i) Equalize exponent	0	0111	(1)0101	→	0	0111	(1)0101
	0	0010	(1)0000	→	0	0111	(0)0000(1) +
(ii) Add mantissa					0	0111	(1)0101
(iii) Normalize result					0	0111	(1)0101

Figure 4.1: Loss of accuracy as a result of floating point operation reordering.

Adding that with A results in a number with an exponent of 0111 and a mantissa of 0110. The decimal value of the outcome is 1.375, and no accuracy is lost.

If the order of addition is changed, i.e.. A is first added with B, then to C, then the result is different. First, when A is added to B, B's exponent is equalized to that of A, requiring the mantissa to be shifted by five positions. The most significant (hidden) bit of B is shifted out (indicated by the wavy arrow), and B's mantissa becomes 0000 (a decimal value of 0). We have lost some accuracy in representing

B. When A and B are added, we obtain the result with a mantissa 0101 and an exponent of 0111, which is the same with the original value of A. When this result is later added with C, the same loss of accuracy occurs and the computation result has a mantissa of 0101 and exponent of 0111. Hence, (A+B)+C yields a different result compared to A+(B+C). Such operation reordering occurs frequently in reduction operations used in parallel execution of loops.

Many programs can usually tolerate slight changes in results, but it is not safe to assume that all programs can. For example, in numerical algorithms, a slight change in the value of a key floating point number can make a difference between the algorithm converging to a solution, versus the algorithm diverging from the solution and failing to terminate. It is also possible that the parallel execution of the programs run for a different number of time-steps compared to the sequential execution of the same programs. In any case, both programmers or performance tuners must be aware of this possibility and verify that their algorithms can tolerate a small change in results.

4.1.2 Incorrect or Missing Synchronization

Using extraneous synchronization such as unnecessary barriers or locks typically only leads to reduced performance and does not affect the correctness of program execution. However, missing a synchronization primitive leads to a *possible* incorrect execution. In contrast to a bug in sequential programs, a bug in parallel programs is often timing dependent, i.e. it manifests as a program error only under certain timing conditions between different threads.

For example, the following code (Code 4.1) shows parallel loops in which the programmer incorrectly omits barrier synchronizations. In the first loop, the computation result in array $A[i]$ is used by tasks in the second loop. If the barrier is not omitted (by not including the `nowait` clause), we make sure that by the time the second loop executes, all elements of array $A[i]$ have been updated by all threads. With the barrier omitted, there is no guarantee that all elements of array $A[i]$ have been updated when each element is accumulated to the `sum` in the second loop. This could produce an incorrect value of `sum` as some elements of array $A[i]$ still have their old values when they are accumulated.

However, an important note is that an error in the value of `sum` does not necessarily occur. First, it is possible that even without a barrier, all threads take the same amount of time to execute their tasks. Hence, the first loop is completed before any thread starts to execute the second loop, which does not manifest in

Code 4.1 A code that misses a barrier synchronization.

```
1 #pragma omp parallel for default(shared) nowait
2 for(i=0; i<n; i++)
3     A[i]= A[i] + B[i];
4 #pragma omp parallel for default(shared) reduction(+:sum) \
5       nowait
6 for(i=0; i<n; i++)
7     sum = sum + A[i];
8 print sum;
```

errors. Therefore, the manifestation of the error is highly timing dependent. Secondly, it is also possible (in fact, it is likely), that a thread that executes a certain iteration range in the first loop also executes the same iteration range on the second loop. Hence, the elements of array $A[i]$ that are written by the thread in the first loop are also the same elements that are read by the same thread in the second loop. As a result, for any thread, it will have finished its computation in the first loop before it goes on the second loop. In such a case, even without a barrier synchronization, the code still executes correctly. However, this situation changes when a thread may execute different iteration ranges in the two loops. For example, if the `schedule()` clause with different chunk sizes is added to the two loops, or when different scheduling policies are applied to the loops, it is likely that a thread executes different iteration ranges in the two loops, potentially causing incorrect execution results.

Another case of missing synchronization is forgetting to protect access to conflicting shared variables with a critical section (i.e., lock-unlock pair). It may result in various threads updating the variables incorrectly. An example of such error is shown in Code 4.2. The code shows that the conflicting variable `sum` is neither declared in a reduction clause nor enclosed with a critical section.

Code 4.2 A code that misses a critical section.

```
1 #pragma omp parallel for default(shared)
2 for(i=0; i<n; i++)
3     sum = sum + A[i];
4 print sum;
```

In the example above, suppose two threads execute `sum = sum + A[i]` simultaneously. The statement is not *atomic*, i.e. it consists of multiple machine

instructions: loads of `sum` and `A[i]`, an addition instruction, and a store to `sum`. Imagine two threads loading the old value of `sum` simultaneously. One thread adds to the sum with an array element and stores it to memory. The other thread also adds to the sum and when it stores the result to memory, it overwrites what the earlier thread has written. Hence, the omission of critical section will cause the value of `sum` that is printed to be different than the correct value.

4.1.3 Incorrect Variable Scope

Declaring a variable that needs to be declared private, as shared, may cause correctness problems. However, declaring a variable that needs to be declared shared, as private, may not always cause correctness problems. For shared-only and Read-/Write non-conflicting variables that are declared as private, as long as they are initialized to the values of the shared copies, they do not cause a correctness problem, but would cause an unnecessary storage overhead. However, Read/Write conflicting variables that are declared as private cause a correctness problem and may manifest in incorrect program output.

The following code (Code 4.3) shows several incorrect variable scope declarations. For example, array `c` is declared as private when it should be declared as shared. The outcome is non-deterministic, depending on whether the private copy of `c` is initialized to the shared copy of `c`. If it is, then the code only suffers from storage and initialization overheads. Otherwise, it will result in wrong program output. Array `a` is declared as private when it should be declared as shared. The outcome is also non-deterministic as it depends on whether the updated elements of the array `a` are merged into the shared copy. Finally, the variable `temp` is declared as shared when it should be declared as private. The result of this is that various threads will be using the same location for `temp` and overwrite each other, causing wrong program output.

Code 4.3 A code that has incorrect variable scopes.

```
1 #pragma omp parallel for shared(b, temp) private(a, c)
2 for (i=0; i<N; i++) {
3   temp = b[i] + c[i];
4   a[i] = a[i] * temp;
5 }
```

4.2 Parallelizing Compiler Limitations

One may wonder if some of the code analysis techniques that were covered in Chapter 3 can be automated and performed by a compiler. The answer is yes, but only to some extent. Many OpenMP compilers (compilers that can translate OpenMP directives into parallel code) also have the ability to examine loop code structures and insert OpenMP directives automatically. Such facility makes shared memory parallel programming extremely easy since all programmers need to do is to use a parallelizing compiler and they will have a correct parallel program.

Unfortunately, however, automatic parallelization is likely to be limited in its effectiveness. A compiler must achieve several goals simultaneously. The most important goal is *producing correct code*. When a compiler is not sure whether a transformation will produce correct code, it will opt to conservatively avoid the transformation. The second goal of a compiler is that it must compile source code within a realistic time. Hence, when an analysis that may expose parallelism requires a long time to perform, it is likely to be skipped by the compiler. The third goal of a compiler is to produce a high-performing code, meaning that the code must be fast and have a small instruction and data footprint. When a transformation is *risky*, i.e. it may not always produce better performance, the compiler often skips the transformation, or groups it together with other risky transformations at high optimization levels. Often, high optimization levels are disabled for compiling parallel program, hence they are unavailable to parallel program compilation.

Overall, it is still critical to understand code and algorithm analysis techniques since programmers often find that they cannot rely on the compiler to perform them automatically.

4.2.1 A Case Study on the MIPSpro Compiler

In this case study, we use an SGI MIPSpro compiler that was considered as a state-of-the-art parallelizing compiler at the time the study was conducted (in 1999). The compiler has automatic parallelization techniques in a package called *Power Fortran Analyzer* that can be invoked with the compiler option `-pfa`. The compiler was applied to SPECfp [Standard Performance Evaluation Corporation, 2006] 1995 applications that include swim, tomcatv, and mgrid. These applications were written in Fortran language and perform scientific simulations. *swim* is a shallow water simulation, *tomcatv* is a vectorized mesh generation, and *mgrid* is a 3-dimensional potential field simulation.

The goal of this case study is to extract as much parallelism as possible. In the real world, sometimes some parallelism is not worth extracting, such as when the code has a negligible contribution on the overall execution time, or when the tasks are too small that parallelization overheads dominate. Nevertheless, whether extracting parallelism is profitable or not depends on many architecture-specific parameters which are not the focus of this case study. Hence, in this case study, we focus on extracting the maximum amount of parallelism.

Code 4.4 swim's parallelization

```
1 DO 3500 ICHECK = 1, MNMIN
2 DO 4500 JCHECK = 1, MNMIN
3 PCHECK = PCHECK + ABS(PNEW(ICHECK,JCHECK))
4 UCHECK = UCHECK + ABS(UNEW(ICHECK,JCHECK))
5 VCHECK = VCHECK + ABS(VNEW(ICHECK,JCHECK))
6 4500   CONTINUE
7 UNEW(ICHECK,ICHECK) = UNEW(ICHECK,ICHECK)
8 1  * ( MOD (ICHECK, 100) /100.)
9 3500   CONTINUE
```

The equivalent C code:

```
1 for (ICHECK=1; ICHECK<=MNMIN; ICHECK++) {
2   for (JCHECK=1; JCHECK<=MNMIN; JCHECK++) {
3     PCHECK = PCHECK + ABS(PNEW[ICHECK][JCHECK]);
4     UCHECK = UCHECK + ABS(UNEW[ICHECK][JCHECK]);
5     VCHECK = VCHECK + ABS(VNEW[ICHECK][JCHECK]);
6   }
7   UNEW[ICHECK][ICHECK] = UNEW[ICHECK][ICHECK]
8     * (MOD(ICHECK,100)/100);
9 }
```

Code 4.4 shows the section of code in swim that the compiler does not parallelize. The compiler complains that both loops are not parallelizable because there are "scalar dependences" on PCHECK, UCHECK, and VCHECK. What this means is that essentially, the compiler finds loop-carried dependences that involve the scalar variables PCHECK, UCHECK, and VCHECK. For example, the value written to PCHECK in one loop iteration is read in the next. The compiler fails to identify that PCHECK, UCHECK, and VCHECK are reduction variables with a reduction operator of floating point addition. The reason for this failure is the function call `ABS(.)`. The function returns an absolute value of a number, hence it only operates on the function argument and has no side effects, i.e. it does

not create dependences with other loop iterations. If the compiler had chosen to perform *inter-procedural analysis* (analyzing dependences across function calls) or perform *function inlining* (replacing a call to a function with the function body) prior to parallelizing the code, it would have discovered the parallelism and parallelized the loops. However, inter-procedural analysis is often time consuming, while function inlining often causes code size explosion. These practical limitations prevent the compiler from discovering the opportunity to parallelize the code. The compiler conservatively assumes that the `ABS()` function may access some variables that introduce dependences with other loop iterations. To remedy this situation, the programmer can force parallelization of the loop, for example through inserting the following OpenMP directives (Code 4.5).

Code 4.5 Parallelized loops in swim

```
1 #pragma omp parallel for default(shared) \
2     reduction(+: PCHECK, UCHECK, VCHECK)
3 for (ICHECK=1; ICHECK<=MNMIN; ICHECK++) {
4   for (JCHECK=1; JCHECK<=MNMIN; JCHECK++) {
5     PCHECK = PCHECK + ABS(PNEW[ICHECK][JCHECK]);
6     UCHECK = UCHECK + ABS(UNEW[ICHECK][JCHECK]);
7     VCHECK = VCHECK + ABS(VNEW[ICHECK][JCHECK]);
8   }
9   UNEW[ICHECK][ICHECK] = UNEW[ICHECK][ICHECK]
10    * (MOD(ICHECK,100)/100);
11 }
```

Another example of the compiler not extracting the maximum amount of parallelism can be seen in the following code (Code 4.6). The parallelizing compiler does not parallelize the inner loop (*for i* loop) because it is "contained within a parallel construct". What this means is, essentially, the outer loop is parallelized but the inner loop is not. It is a common heuristic an automatic parallelizing compiler uses: if there are several loops in a loop nest that have DOALL parallelism, only the outermost loop is chosen for parallelization. The obvious reason is that parallelizing inner loops incur higher overheads since each iteration of the outer loop enters and exits a parallel section. However, there may be cases in which the heuristic is too restrictive, for example when the number of iterations of the outermost loop is small, or when parallelizing the outermost loop introduces load imbalance. In addition, parallelizing an inner loop may not produce significant parallel overheads if the body of the inner loop is huge compared to the overhead

Code 4.6 Another loop in swim

```
DO 50 J=1,NP1
DO 50 I=1,MP1
PSI(I,J) = A*SIN((I-.5D0)*DI)*SIN((J-.5D0)*DJ)
P(I,J) = PCF*(COS(2.D0*(I-1)*DI)
1                +COS(2.D0*(J-1)*DJ))+50000.D0
50 CONTINUE
```

The equivalent code in C will look like this:

```
for (J=1; J<=NP1; J++) {
  for (I=1; I<=NP1; I++) {
    PSI[I][J] = A*SIN((I-.5D0)*DI)*SIN((J-.5D0)*DJ)
    P[I][J] = PCF*(COS(2.D0*(I-1)*DI)
                +COS(2.D0*(J-1)*DJ))+50000.D0
  }
}
```

of entering and exiting the parallel section, such as when the inner loop body contains other loops.

Another reason for not parallelizing a loop may have to do with operations that the compiler cannot analyze, such as when I/O statements are involved. This is shown in the following code (Code 4.7).

Code 4.7 A loop in tomcatv

```
DO    10     J = 1,N
DO    10     I = 1,N
READ(10,600,END=990) X(I,J),Y(I,J)
10 CONTINUE
```

The equivalent code in C will look like this:

```
for (J=1; J<=N; J++) {
  for (I=1; I<=N; I++) {
    fscanf(filehandle, ``%f %f\n'',&X[I][J],&Y[I][J]);
  }
}
```

Essentially, the loop reads from a file, one line at a time. The compiler assumes conservatively the I/O library is not parallel, which is a reasonable heuristic. However, programmers can get around this by using a parallel I/O library which allows

multiple file pointers to read different sections of the file at the same time. If such library is not available, programmers can still exploit parallelism by dividing up the input file into many smaller files, such that each parallel thread can read from a unique file. Such a method is worthwhile when the file being read is huge and file operations restrict the scalability of the program.

Code 4.8 shows another example in which the compiler cannot extract parallelism from the code. The loop visits matrix elements RX and RY, takes their absolute values, finds the maximum of the values, and assigns the maximum values to RXM[ITER] and RYM[ITER].

Code 4.8 A loop in tomcatv

```
1 DO      80     J = 2,N-1
2 DO      80     I = 2,N-1
3 RXM(ITER) = MAX(RXM(ITER), ABS(RX(I,J)))
4 RYM(ITER) = MAX(RYM(ITER), ABS(RY(I,J)))
5 80    CONTINUE
```

The equivalent code in C will look like this:

```
1 for (J=1; J<=N-1; J++) {
2   for (I=1; I<=N-1; I++) {
3     RXM[ITER] = MAX(RXM[ITER], ABS(RX[i][j]));
4     RYM[ITER] = MAX(RYM[ITER], ABS(RY[I][J]));
5   }
6 }
```

The reason the compiler does not parallelize the loops is "array dependence for RXM and RYM". Essentially, the compiler detects there are loop carried dependences between the reads to RXM/RYM and the writes to RXM/RYM. As in Code 4.4, the compiler probably does not perform inter-procedural analysis and function inlining. However, there is another reason why it may not parallelize the code: it cannot infer whether reduction is applicable. From the algorithm point of view, we know that the operation that takes the maximum value of two numbers is commutative and associative, i.e. no matter what the order of traversal of the matrix elements, in the end the maximum values of the elements will be identical. However, some compilers cannot infer the commutativity and associativity of the MAX() function call. Hence, even if they perform inter-procedural analysis or function inlining, they still may not be able to parallelize the loops.

In order to parallelize the loops, programmers have to apply reduction on

RXM[ITER] and RYM[ITER]. Unfortunately, though, in C/C++, they cannot use OpenMP directives for that. The reason is the `reduction (operator: variable_list)` clause requires the `operator` to be a language-defined binary operators such as '+', '*', '&', '—', etc. Since `MAX` is a function rather than a language-defined operator, it cannot be specified as a reduction operator. In Fortran, however, MAX() is a function that is included in the language definition, hence the compiler can rely on it to be commutative and associative. In fact, in Fortran, programmers can specify MAX or MIN as a reduction operator.

Note that for Code 4.8, programmers can also exploit function parallelism. Both statements in the loop body have no dependences with each other, hence the loop nest can be distributed into two with one loop nest executing just the first statement, and the second loop nest executing only the second statement. The result will be two parallel tasks that have similar loads. Of course, if there are many processors, parallelizing the iterations of the loop nest yields a much higher degree of parallelism.

An example that shows that the compiler does not perform the necessary analysis to identify parallelism can be seen in the following code (Code 4.9). The loop consists of two statements that perform function calls. The parallelizing compiler reports that the loop is not parallelizable because it contains "call to mg3p and resid". Essentially, this indicates that the compiler is not willing to analyze dependences inside and across those functions.

Code 4.9 A loop in mgrid

```
1 DO 20 IT=1,NIT
2 CALL MG3P(U,V,R,N,A,C,NV,NR,IR,MM,LMI)
3 CALL RESID(U,V,R,N,A)
4 CONTINUE
```

The equivalent code in C will look like this:

```
1 for (IT=1; I<=NIT; I++) {
2   MG3P(U,V,R,N,A,C,NV,NR,IR,MM,LMI);
3   RESID(U,V,R,N,A);
4 }
```

To get around this situation, one thing the programmer can do is to force interprocedural analysis through a compiler flag (if it is available), or manually inline the function calls. This will allow the compiler to analyze dependences in the bodies of the functions and across the two function calls. Manual analysis of

the function bodies may also be necessary to identify all sources of parallelism (DOALL, DOACROSS, across loop statements, and DOPIPE).

To summarize, parallel programming cannot be fully relegated to an automatic parallelizing compiler yet. Programmers or performance tuners need to analyze code and algorithm in order to be able to extract the maximum amount of parallelism.

4.3 Performance Considerations

4.3.1 Amdahl's Law

Fundamentally, what programmers expect from parallel execution of their algorithm or code is to obtain smaller execution time compared to the sequential execution of the algorithm or code. Suppose that the sequential execution of the algorithm or code takes T_1 time units, while parallel execution of the algorithm or code on p processors takes T_p time units. Suppose that out of the entire execution of the program, s fraction of it is not parallelizable while $1 - s$ fraction is parallelizable. Then, the *speedup* (Amdahl's formula) of the parallel execution compared to the sequential execution is:

$$\begin{aligned} speedup_p &= \frac{T_1}{T_p} \\ &= \frac{T_1}{T_1 \times s + T_1 \times \frac{1-s}{p}} \\ &= \frac{1}{s + \frac{1-s}{p}} \end{aligned} \tag{4.1}$$

For a very large number of processors, assuming that the serial fraction s is constant, the speedup of parallel execution becomes:

$$\begin{aligned} speedup_\infty &= \lim_{p \to \infty} \frac{1}{s + \frac{1-s}{p}} \\ &= \frac{1}{s} \end{aligned} \tag{4.2}$$

The formula for $speedup_\infty$ expresses the maximum attainable speedup on an unlimited number of processors. While the formula is simple, it has important implications. For example, if the serial fraction s is 5%, then the maximum speedup

ratio on an unlimited number of processors would only be 20! However, if the serial fraction s can be reduced to 0.5%, then the maximum speedup ratio on an unlimited number of processors increases to 200. An important observation here is that obtaining a high degree of scalability is hard; it requires all parts of a program to be almost completely parallelizable. All other parts that are not parallelized must be very small in order to produce a scalable parallel execution performance.

In practice, s does not just contain parts of the program code that cannot be parallelized. It also includes the effect of load imbalance and various parallel overheads such as synchronization overheads, communication overheads, and thread management overheads.

■ *Did you know?*

Under the perfect situation in which there is no fraction of the program that cannot be parallelized completely (i.e. $s = 0$), the speedup will be $speedup_p = p$, which is referred to as a linear speedup. *Linear speedup is the* theoretical *maximum speedup attainable. However, in practice many programs have been known to show* super-linear *speedups on some number of processors. The reason for this is that as the number of processors grows, the combined cache space also grows. At some tipping point, adding extra processors change the situation from the working set of the program exceeding the total cache space to the situation in which the working set fits into the total cache space. When this occurs, cache capacity misses are reduced significantly and the execution of all threads become much faster, leading to the super-linear speedups.*

Complicating the effort of making scalable parallel programs is the fact that the serial fraction s does not always stay constant when the number of processors increases. For example, synchronization overheads typically increase with the number of processors, often super-linearly. Hence, as the number of processors increases, the time to perform barrier synchronization, acquire locks, etc. is longer and takes a larger fraction of the original execution time. Communication overheads also tend to increase because a higher number of processors cannot be interconnected with very short physical distances. The communication latency between any random pair of nodes increases with the number of processors. Thread management overheads also increase with a higher number of processors because thread management requires synchronization and communication. Hence, s tends to increase as the number of processors increases, independent of the serial fraction inherent in the algorithm. Consequently, it is often the case that achieving a

highly scalable parallel program requires s to be reduced by using a very large input set. This allows the overheads of synchronization, communication, and thread management to contribute a smaller fraction of execution compared to that of computation. Fortunately, many important scientific simulations do have very large input sets.

■ *Did you know?*

In the Amdahl's law, there is an ambiguity as to what should constitute as the execution time of a sequential execution of a program (T_1). The choices include:

1. *The same parallel program used in parallel execution, but it runs with a single thread on a single processor.*
2. *The sequential version of the program.*
3. *The best sequential version of the program.*

The first choice is incorrect to use; the sequential execution should not include parallel overheads, such as thread creation, calls to parallel libraries, etc. For example, when a program containing OpenMP directives is compiled with an OpenMP compiler, it includes calls to OpenMP libraries even when it runs on a single thread. However, using thread-safe libraries, which include the overheads of entering and exiting critical sections, for sequential execution may be acceptable if they are the only libraries that are available.

The second choice of using the sequential version of the program is more correct since parallel overheads are not included, but care must also be taken. A potential pitfall of the choice is whether the sequential version of the program is a realistic implementation for sequential execution.

The correct choice is the last option, which uses the best sequential version of the program to measure T_1, both algorithm-*wise and* code-*wise. For example, some algorithms are developed for the purpose of extracting parallelism (e.g. iterating over anti-diagonals in the Ocean example), but it may not be the best version chosen for a sequential implementation. The second aspect is whether the code is the best sequential code given the algorithm. For example, some aggressive compiler optimizations that can benefit sequential execution are often not performed when the program is compiled for parallel execution, but they can produce a better performing sequential code when they are applied.*

4.3.2 Parallel Thread Granularity

At the heart of deciding the granularity of parallel threads are potential trade-offs of load balance, communication overheads, and extra work performed by the threads. Parallel thread granularity is a very important factor affecting scalability of a parallel program. It affects the relative cost of thread management: creation, task assignment, and synchronization. The larger the task granularity, the smaller thread management overheads are relative to the execution time of the tasks.

On a loop nest structure, one easy way to increase parallel task granularity is through choosing to parallelize the outer loop and not parallelize inner loops. For example, in the matrix multiplication code in Code 3.19, both the $for\ i$ and $for\ j$ loops have DOALL parallelism. Parallelizing the $for\ i$ loop requires one set of thread creation and termination. On the other hand, parallelizing the $for\ j$ loop requires N sets of thread creation and termination, each set for one $for\ i$ iteration.

Parallelizing an inner loop rather than the outer loop can be considered when the outer loop is not parallel, or when it may have a *poor load balance*. For example, consider the following code (Code 4.10). The `if` statement in the outer loop indicates that not all iterations the body of the loop will be executed. Without knowing the actual values of $a[i][0]$, we do not know which iterations will execute the loop body and which ones will not. If we parallelize the outer loop, we may have load imbalance, because it is possible for some threads to be assigned outer loop iterations that do not always execute the loop body, and other threads to be assigned iterations that execute the loop body. Clearly, load imbalance restricts the scalability of the program, so the situation needs to be addressed.

Code 4.10 A loop showing a conditionally executed inner loop.

```
for (i=0; i<N; i++) {
  if (a[i][0] > 0)
    for (j=0; j<N; j++) {
      a[i][j] = a[i][j] + 1;
    }
}
```

There are at least two possible ways of how to improve the load balance in this example. The first solution is to use a dynamic task-to-thread mapping scheme (to be discussed in Section 4.3.5). Rather than statically assigning iterations to threads, we can have a dynamic assignment in which a central task queue is set up to keep tasks in the form of groups of loop iterations. At run time, whichever

thread becomes idle, it grabs a task from the task queue, executes it, and when it becomes idle again, it grabs another task, and so on. This process is repeated until there is no more task in the task queue. Another solution is to parallelize the inner loop. Since the body of the inner loop is not conditionally executed, each iteration of the inner loop will execute its body. Hence, statically assigning the inner loop iterations to threads can achieve good load balance, at the expense of larger parallel overheads (entering and exiting parallel sections on potentially each outer loop iteration). The increase in parallel overheads may be tolerable if the computation of the body of the inner loop is substantial. Overall, programmers must assess the situation, profile the performance for likely input, and choose the solution that gives the best performance.

4.3.3 Synchronization Granularity

Point-to-point synchronization is quite cheap since it only involves two threads (two processors). However, *global* synchronization, such as locks and barriers is expensive; the time spent in it generally grows with the number of processors. Hence, the size of critical section needs to be as small as possible. However, sometimes there are trade-offs between critical section granularity and the complexity of programming, especially for programs with non-loop structures.

One example is when we have a linked list accessed by multiple threads, in which a thread performs node insertion or deletion. Threads concurrently perform insertion, deletion, and traversal on the linked list. The choice of the granularity of the critical section matters a lot here. On one end of the spectrum, we can have a coarse-grain critical section in which when any operation modifies any part of the list (insertion and deletion of a node), the operation is enclosed in a critical section. Hence, only one thread is allowed to perform an operation that modifies the linked list at any given time. Such an approach is safe and very simple. All programmers need to do is to identify operations that modify the list, and if they use OpenMP, such operations can be enclosed with `#pragma omp critical`. However, obviously parallelism is exploited only when there are multiple threads that read from (but not modify) the linked list.

A more aggressive solution is to reduce the size of the critical section by associating a lock with each linked list node. A thread that performs a linked list insertion or deletion operation locks all the nodes that are involved in the operation, but leaves other nodes unlocked. This allows multiple insertion and deletion operations to proceed simultaneously as long as they modify different parts of

the list. However, such extra parallelism occurs at the expense of a higher programming complexity. For example, with fine grain locks, deadlocks become a possibility, and memory management functions such as node allocation and deallocation must be changed in order not to conflict with other operations. This will be discussed more in Chapter 5. For now, it suffices to conclude that in many cases, there are trade-offs between parallelism and programming complexity.

4.3.4 Inherent and Artifactual Communication

In this section, we will evaluate communication overheads and how they relate to task-to-thread mapping decision. There are two types of communication overheads: *inherent* communication which occurs due to the algorithm, and *artifactual* communication which occurs due to the way data is laid out and the architecture mechanisms used to pass it around. Artifactual communication overheads cannot be inferred without first understanding how parallel architectures are designed, so we will first discuss inherent communication.

One metric that is useful in evaluating inherent communication is the *communication to computation ratio* (CCR). To compute CCR, we can estimate the amount of communication of a thread divided by the amount of computation of the thread. The parameters are the number of processors and the input size.

To illustrate how inherent communication is computed, consider the Ocean application from Code 3.14. In one sweep the loop iterates over i and over j to visit all elements of the matrix. Suppose that we have p processors and the matrix dimension of $N \times N$ greatly exceeds p. There are at least three possible ways to assign tasks to threads, as shown in Figure 4.2.

The figure assumes that all iterations are parallel (i.e. we relax the loop-carried dependences). In this case, we can partition the tasks block-wise (a), row-wise (b), or column-wise (c). For all cases, the amount of computation for each thread, expressed in the number of matrix elements that need to be updated, are identical, which is $\frac{N \times N}{p}$. However, the amount of communication among threads differs with different mapping strategy. Recall that in Ocean, to update one matrix element, we use matrix elements that are its nearest neighbors. Hence, for the block-wise partitioning strategy, in each block, all border elements are accessed by more than one thread, but interior elements are only accessed by the thread that the block is assigned to. There are approximately $4 \times \frac{N}{\sqrt{p}}$ elements at the borders (top, bottom, left, and right borders). Therefore, amount of communication, expressed in the number of elements that are communicated between the thread and

Figure 4.2: Assigning tasks to threads block-wise (a), row-wise (b), and column-wise(c).

other threads is $4 \times \frac{N}{\sqrt{p}}$. Consequently, the CCR is as follows:

$$
\begin{aligned}
CCR &= \frac{communication}{computation} \\
&= \frac{4 \times \frac{N}{\sqrt{p}}}{(\frac{N^2}{p})} \\
&= \frac{4 \times \sqrt{p}}{N}
\end{aligned}
$$

$$=\Theta(\frac{\sqrt{p}}{N}) \tag{4.3}$$

For row-wise partitioning, for each partition ($\frac{N}{p}$ rows), there are $2 \times N$ border elements that are communicated with other threads. Hence, the CCR can be computed as:

$$\begin{aligned} CCR &= \frac{communication}{computation} \\ &= \frac{2 \times N}{(\frac{N^2}{p})} \\ &= \frac{2 \times p}{N} \\ &= \Theta(\frac{p}{N}) \end{aligned} \tag{4.4}$$

For column-wise partitioning, each partition consists of $\frac{N}{p}$ columns, and there are $2 \times N$ border elements that are communicated with other threads. Hence, the CCR is identical to that of row-wise partitioning.

If the problem size stays constant, and we increase the number of processors, CCR increases too, but at different rate in different mapping schemes. In bock-wise partitioning, it increases at the rate of $\sqrt{p}$, while at row-wise and column-wise partitioning, it increases at the rate of p. Hence, clearly in terms of inherent communication, block-wise mapping is superior.

However, when artifactual communication is considered, the question is not as clear cut. Artifactual communication must consider the ping-pong effect of sharing data between two processors, i.e. as data is shuffled back and forth between communicating processors. The overhead of such communication not only depends on the frequency of data shuffling, but also on the latency of each data shuffle. The frequency of data shuffling depends on the layout of data in memory. For example, matrix elements in consecutive columns of a row are laid out contiguously in C/C++, but in Fortran, matrix elements in consecutive rows of a column are laid out contiguously. Data shuffling occurs at the granularity of a cache block (typically 64 or 128 bytes), which can hold 16 to 32 float-type matrix elements, or 8 to 16 double-type matrix elements. The amount of total (inherent plus artifactual) communication does not depend on how many matrix elements are communicated. Rather, it depends on how many cache blocks are shared between processors.

To illustrate the problem, suppose that each processor must compute 64 matrix elements and that we have a cache block that can hold 4 matrix elements. In block-wise partitioning (see Figure 4.3(a)), in the top and bottom rows, the matrix elements are contained in six cache blocks. However, each matrix element in the left and right border requires one cache block. In all, there are up to 18 cache blocks that are shared with other processors. In a row-wise partitioning (see Figure 4.3(b)), the elements in the top and bottom rows only occupy up to 8 cache blocks, so only eight cache blocks are shared among processors. Finally, in a column-wise partitioning (see Figure 4.3(c)), each of the element occupies one cache block, so overall there are up to $2 \times 16 = 32$ cache blocks that are shared. Hence, row-wise partitioning achieves the least artifactual communication, followed by block-wise partitioning, and finally by column-wise partitioning.

Figure 4.3: Number of cache blocks shared by more than one processors in the block-wise partitioning (a), row-wise partitioning (b), and column-wise partitioning (c), assuming the memory layout is as shown.

This example illustrates a few points. First, despite having an identical amount of inherent communication, row-wise and column-wise partitioning differ significantly in terms of total communication. Hence, it is very important to select a partitioning strategy that minimizes such communication. Secondly, block-wise par-

titioning may be less cache efficient than row-wise partitioning if it incurs much artifactual communication due to communicated matrix elements at the extreme columns within the block.

The conclusion is that while inherent communication reflects the communication requirement of an algorithm, the actual real-world performance is more determined by the amount of total communication, which is not necessarily identical to the behavior of inherent communication.

4.3.5 Scheduling and Load Balancing

Another issue regarding mapping of tasks to threads is load balance. A program cannot become highly scalable unless the loads of threads are balanced. One important factor that determines load balance is whether mapping of tasks to threads are decided at compile time (*statically*) or at run-time (*dynamically*). If the load is known at compile time, static mapping is preferable as the assignment of tasks to threads can be hardwired into the code. Hence, static mapping tends to have a low overhead. In dynamic task mapping, tasks are not pre-assigned to threads prior to execution. During execution, whichever thread becomes idle, it grabs a task from the task queue and executes it. Dynamic task mapping introduces extra overheads for the task queue management but sometimes allows load balance to be achieved more easily. It also tends to increase the amount of communication and reduces locality, since data may be laid out in a node that is far from the processor that actually uses it. Finally, it is possible to have a hybrid mapping in which the mapping is largely static, but periodically the load balance is assessed and the mapping is adjusted accordingly.

OpenMP provides a way to specify static or dynamic mapping through the use of the `schedule` clause. The format of the clause is `schedule(type, chunksize)`. `type` can be one of *static*, *dynamic*, *guided*, or *runtime*. *chunksize* specifies the number of consecutive iterations that should be assigned to one thread. A chunk of iterations is assigned to a thread, the next chunk of iterations is assigned to the next thread, and so on, in a round-robin manner.

Code 4.11 shows a code example in which the number of iterations of the inner loop changes for different iterations of the outer loop. For illustration purpose, let us assume n and p have the values of 8 and 2, respectively. Therefore, the outer loop will execute 8 iterations, while the inner loop will execute $1, 2, 3, \ldots, 8$ iterations. The choice of chunk size determines how much imbalance there is between threads.

Code 4.11 Code illustrating load balance and OpenMP schedule clause.

```
sum = 0;
#pragma omp parallel for reduction(+:sum) \
    schedule(static, chunksz)
for (i=0; i<n; i++) {
  for (j=0; j<=i; j++) {
    sum = sum + a[i][j];
  }
}
Print sum;
```

Figure 4.4 shows the impact of various chunk sizes in a static schedule on load balance. For example, if the chunk size is 4, then the first thread (thread 0) executes the first four iterations of the outer loop, while the second thread (thread 1) executes the last four iterations of the outer loop. The total number of inner loop iterations (or the number of times the "sum" statement is executed) is 36. Using a chunk size of four, thread 0 executes 1+2+3+4=10 inner loop iterations while thread 1 executes 5+6+7+8=26 inner loop iterations. Hence, the imbalance, expressed as the difference in the number of iterations executed by the two threads divided by all iterations, is $\frac{16}{36} = 44\%$.

Using a chunksize of 2, thread 0 executes the first and third two outer loop iterations, resulting in a total number of inner loop iterations of (1+2)+(5+6)=14. Thread 1 executes (3+4)+(7+8)=22 inner loop iterations. The imbalance has reduced to $\frac{8}{36} = 22\%$. Finally, using a chunksize of 1, thread 0 executes all odd outer loop iterations, with the total number of inner loop iterations of 1+3+5+7=16. Thread 1 executes all even outer loop iterations, with a total number of inner loop iterations of 2+4+6+8=20. The imbalance is reduced even more to $\frac{20-16}{36} = 11\%$. This example illustrates that smaller chunk sizes tend to achieve better load balance. However, unless the number of outer loop iterations is high enough, static schedule may not achieve an acceptable load balance.

One way to achieve a better load balance is to use a `dynamic` schedule with a small enough chunk size. For example, if chunk size is one, then there will be eight tasks, each corresponding to one outer loop iteration that will be placed in the task queue. An idle processor takes a task from the queue by entering a critical section, removing the task at the head of the queue, and then leaving the critical section. Consequently, if there are too few tasks, load imbalance can still occur, so dynamic scheduling may not be able to compensate for the load imbalance caused by large chunk sizes. However, small chunk sizes incur higher overheads due to

Figure 4.4: Number of iterations executed using `static` schedule with various chunk sizes: four (a), two (b), and one (c).

the task queue management and the frequency of critical sections when a thread retrieves a task from the queue.

To reduce the overhead of task queue access, a `guided` schedule is provided in OpenMP. In the guided schedule, task sizes are not uniform. Tasks nearer the head of the queue are larger (containing more iterations) while tasks farther away from the head of the queue are smaller (containing fewer iterations). In terms of task queue overhead, fewer tasks mean that the queue is accessed less frequently. In terms of load balance, the reasoning behind the guided schedule is that load imbalance is more likely more when there are fewer tasks left. This is because when there are still a large number of tasks, any load imbalance will gradually disappear as under-loaded threads grab more tasks from the task queue. In contrast, when there are a few tasks left, a load imbalance situation is hard to correct because under-loaded tasks may not find any tasks left in the task queue. In order to allow load imbalance to be corrected, latter tasks are made smaller so that there are more of them. In the guided schedule, the chunk size refers to the minimum size of tasks.

The final schedule choice is `runtime`, which is actually not a new type of schedule. Rather, it specifies that the type of schedule is not determined statically. At run-time, when the parallel region is encountered, an environment variable that specifies the schedule type (`OMP_SCHEDULE`) is read and is used for task-to-thread mapping. Hence, programmers can compile their code once, and through the environment variable they can experiment with different schedule types.

4.3.6 Memory Hierarchy Considerations

Multiprocessor systems often have a complex memory hierarchy. Memory components closer to the processor are faster but smaller. The hierarchy includes registers, caches (Level 1, Level 2, and sometimes Level 3), local memory, and remote memory (for NUMA architecture). The hierarchy is glued together by a communication architecture. Since memory components closer to the processor are faster, to improve performance, we must exploit both *spatial* and *temporal locality*. Temporal locality is a behavior in which a recently-accessed datum or instruction tends to be accessed again in the near future. Spatial locality is a behavior in which the neighbors of a recently-accessed datum or instruction tend to be accessed in the near future. Programs naturally have temporal and spatial locality. Cache memory exploits temporal locality by keeping more recently used blocks and discarding less recently used blocks. Cache memory exploits spatial

locality by fetching and storing multiple contiguous bytes as one cache block, typically 64 or 128 bytes. Although programs naturally have some degree of spatial and temporal locality, often the localities can be improved significantly through code transformation.

Temporal Locality Optimizations

One of the key obstacles for the cache in exploiting the temporal locality of a program is that the program's working set is sometimes larger than the cache. This is especially the case when a program reaccesses certain data that it accessed in the past, however the cache is not large enough to hold that data, resulting in repeated cache misses. For example, take a look at Code 4.12. In iteration $i = 1$, it accesses two rows of the matrix ($a[0][0..n]$ and $a[1][0..n]$). In the next iteration ($i = 2$), it accesses a new row ($a[2][0..n]$) and reaccesses the previous row (($a[1][0..n]$). If the cache is big enough that it still stores the second row, the reaccesses will exhibit temporal locality that is exploitable by the cache. However, if the cache is too small that some parts of the second row are already evicted out of the cache, the reaccesses will incur many cache misses.

Code 4.12 A code that may overflow the cache when n is large.

```
1 for (i=1; i<n; i++) {
2   for (j=0; j<n; j++) {
3     a[i][j] = a[i][j] + a[i-1][j];
4   }
5 }
```

To avoid such misses, we can use a *blocking* transformation, in which the iteration space of the inner loop is divided into blocks where we perform computation on one block completely before moving on to the next block. In other words, we alter the ITG to improve temporal locality by considering the cache size. The code after blocking transformation is shown below (Code 4.13). The code shows that we only access blockFactor columns before moving to the next $for\ i$ iteration. For example, for the first block of columns, the cache only needs to store $a[i][0..blockFactor]$ and $a[1][0..blockFactor]$ in the cache. blockFactor is assigned a value that is small enough to ensure that reaccesses of rows do not result in cache misses.

Code 4.13 The code example after blocking transformation.

```
for (block=1; block<n/blockFactor; block++) {
  for (i=1; i<n; i++) {
    for (j=block*blockFactor; j<(block+1)*blockFactor; j++){
      a[i][j] = a[i][j] + a[i-1][j];
    }
  }
}
```

Certainly if blockFactor is very small, we can have very good temporal locality. However, small blockFactor reduces the amount of spatial locality since we may not fully access bytes in a cache block before moving to the next row. Hence, blockFactor must be chosen at least such that matrix elements that are accessed are at the cache block boundary. For example, if each matrix element is 4 bytes and the cache block size is 64 bytes, then we need to set blockFactor to be multiples of $\frac{64}{4} = 16$. In addition, in general we want to set the blockFactor to be as large as possible since spatial locality is not limited to the size of cache blocks, but also the size of pages.

To find out the maximum value of blockFactor that still produces good temporal locality, we need to figure out how much data needs to be retained until the next time it is reused. That is, first we should determine which data will be reused and how far in the future it will be reused, from the time the data is first used. Call this time period as the *reuse interval*. Then, we need to figure out how much data will be brought into the cache during the reuse interval. The amount of data brought in in this interval should not exceed the cache size, or else the original data may be replaced from the cache before it is reused.

Spatial Locality Considerations

Spatial locality also typically exists naturally in programs, but the way programmers structure their code can greatly affect the amount of spatial locality. For example, the order of traversal in the iteration space may not match with the way data is laid out contiguously in memory. In Code 4.14, the order of iteration traversal is over rows of a column (i.e., $a[0][0], a[1][0], \ldots, a[n-1][0]$), then over rows of the next column (i.e., $a[0][1], a[1][1], \ldots, a[n-1][1]$), and so on. However, data is laid out consecutively in a different order: over columns of a row, then over columns of the next row, and so on. Hence, the order of traversal does not match the order in which data is laid out contiguously in memory, as illustrated

in Figure 4.5. Each iteration will likely bring a cache block but only use a small fraction of it. And by the time a column of the matrix is visited, it is likely that the cache blocks that were brought earlier have been evicted and need to be refetched as the traversal visits the next column.

Figure 4.5: C/C++ memory layout of arrays, showing the traversal pattern for Code 4.14.

Code 4.14 A loop nest that mismatches with how data is laid out in memory.

```
sum = 0;
for (j=0; j<n; j++) {
  for (i=0; i<n; i++) {
    sum = sum + a[i][j];
  }
}
print sum;
```

To remedy the situation, a loop transformation referred to as *loop interchange* can be employed. Essentially, loop interchange switches the outer loop with the inner loop, as shown in Code 4.15. Two loops can be interchanged correctly if they are perfectly nested, i.e. one loop is in the body of the other loop and there are no other statements except the inner loop in the body of the outer loop. The loop interchange needs to be performed prior to parallelization because as it changes the ITG, the LDG may also change, causing a change in what tasks can be run in

parallel.

Code 4.15 After loop interchange, the loop nest matches with how data is laid out in memory.

```
sum = 0;
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = sum + a[i][j];
  }
}
print sum;
```

> ■ *Did you know?*
>
> *In Fortran, the way matrix elements is laid out are the opposite that in C/C++, where different rows in the same column are laid out contiguously. For example, A(3,6) and A(4,6) are laid out contiguously but A(3,6) and A(3,7) are not.*

Another example of how spatial locality can be enhanced is when tasks are mapped onto threads. As discussed in Section 4.3.4, assigning consecutive rows to a thread (row-wise partitioning) results in the least number of cache blocks that are shared among processors. This results in good spatial locality. For example, when a cache block is brought into the cache of a processor, the processor likely accesses all the bytes in the cache block before the cache block needs to be shuffled back to another processor that shares the block. In contrast, block-wise partitioning has less spatial locality, whereas column-wise partitioning has the least amount of spatial locality.

Another important consideration for spatial locality is over the granularity of pages. This issue is specific to Non-Uniform Memory Architecture (NUMA) in which physically, there are many memories in different processing nodes, hence accessing the memory takes a different latency depending on the physical distance between the memory and the processor. The hardware gives an abstraction of a single shared memory by locating the correct memory module for a given address. In NUMA, it is critical for performance to allocate a page in the memory that is the closest to the processor which will access it the most. To achieve that, typically the OS adopts several simple page allocation policies.

When a process is launched, typically the Operating System (OS) does not

allocate all its memory right away so that the OS can quickly launch it. As the process runs, when it accesses a *page* (a unit of memory management – typically having a 4KB size) that is not yet allocated, it will suffer a page fault. The page fault handler of the OS is invoked in reaction to the page fault. The duty of the page fault handler is to support *virtual memory* functions such as allocating a page that has not been allocated, or bringing a page from the swap memory in disk into the physical memory. When the page fault handler allocates a page, it reserves a *frame* in the physical memory where the page can be held, and records the address mapping in a structure called a *page table*. Later on, when the process accesses data at a certain address (referred to as *virtual address*), the address is translated into the actual location in the physical memory (referred to as the *physical address*).

In short, a parallel program, when it begins its execution will suffer from a lot of page faults because its pages have not been allocated. The OS can choose which frame to allocate a page through a *page allocation policy*. One popular page allocation policy that the OS can use is to allocate the page at a frame in which the least recently used page resides. The old page at that frame is evicted to make room for the new page. Such a policy or its approximation is widely implemented in many OSes. However, in a NUMA system, such a policy is not aware of how close the page is to the processor that needs it, so it may not produce an optimal performance. Thus, in a NUMA system, other policies can be used. One policy is to allocate the frame on the node that incurs the page fault, i.e. on the same node where the processor that accesses it resides. Such a policy is referred to as the *first touch* policy. The rationale behind this policy is that the processor that first accesses the data in a page is likely to be the same processor that accesses data in the page the most. Clearly, this is an approximation, a heuristic.

The pitfall of the first touch policy occurs when the main data structure (e.g., the matrices) are initialized in a different traversal order than that of the main computation. Allocating pages according to the initialization traversal order which is inconsistent with the traversal order in the main computation leads to many remote memory accesses during the main computation. Worse, if the initialization is performed sequentially by the main thread, all pages of the data structures will be allocated on a single node. During main computation, all but one node will access a single remote memory. Not only does this incur a high remote memory access latency, but this also causes contention on that node, which increases the memory access latency significantly. Such a scenario is actually quite common

when programmers use an automatic parallelizing compiler. The compiler typically employs (reasonable) heuristics that skip parallelization of loops or loop nests that have a very small body. However, since an initialization loop or loop nest often has a small body, it is often not parallelized. A common solution for this problem is to manually force the parallelization of the inner loop, and ensure that the traversal order used in the initialization loop is the same with that used by the loops in the main computation.

■ *Did you know?*

An alternative solution to the non-optimal page allocation policy is an adaptive policy in which usage of pages are monitored and pages can be migrated from one node in which they have been allocated on, to the node that accesses them the most. Page migration *is implemented in the SGI Origin 2000 system [Laudon and Lenoski, 1997]. Unfortunately, however, page migration is an expensive operation because the physical address of a page changes. There is a possibility that many processors cache the address mapping (called the* page translation entry *or PTE) in an on-chip cache structured referred to as the* translation lookaside buffer *or TLB. Unfortunately, it is difficult to know which processors cache the PTE of a page. Hence, to migrate a page, the OS must conservatively invalidates every processor's TLB. This is a very expensive operation that may reverse the benefit that comes with page migration. Thus, in the Origin 2000, for example, page migration is supported but is turned off as a default setting.*

An alternative page allocation policy is *round robin*, in which pages are allocated in different memory nodes in a round robin manner. Round robin page allocation sounds like a bad idea since it virtually guarantees that there will be many remote memory accesses. However, it spreads pages across nodes in a balanced manner so contention on any particular node rarely occurs with this policy.

in Figure 4.5. Each iteration will likely bring a cache block but only use a small fraction of it. And by the time a column of the matrix is visited, it is likely that the cache blocks that were brought earlier have been evicted and need to be refetched as the traversal visits the next column.

Figure 4.5: C/C++ memory layout of arrays, showing the traversal pattern for Code 4.14.

Code 4.14 A loop nest that mismatches with how data is laid out in memory.

```
1 sum = 0;
2 for (j=0; j<n; j++) {
3   for (i=0; i<n; i++) {
4     sum = sum + a[i][j];
5   }
6 }
7 print sum;
```

To remedy the situation, a loop transformation referred to as *loop interchange* can be employed. Essentially, loop interchange switches the outer loop with the inner loop, as shown in Code 4.15. Two loops can be interchanged correctly if they are perfectly nested, i.e. one loop is in the body of the other loop and there are no other statements except the inner loop in the body of the outer loop. The loop interchange needs to be performed prior to parallelization because as it changes the ITG, the LDG may also change, causing a change in what tasks can be run in

parallel.

Code 4.15 After loop interchange, the loop nest matches with how data is laid out in memory.

```
sum = 0;
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
    sum = sum + a[i][j];
  }
}
print sum;
```

> ■ *Did you know?*
>
> *In Fortran, the way matrix elements is laid out are the opposite that in C/C++, where different rows in the same column are laid out contiguously. For example, A(3,6) and A(4,6) are laid out contiguously but A(3,6) and A(3,7) are not.*

Another example of how spatial locality can be enhanced is when tasks are mapped onto threads. As discussed in Section 4.3.4, assigning consecutive rows to a thread (row-wise partitioning) results in the least number of cache blocks that are shared among processors. This results in good spatial locality. For example, when a cache block is brought into the cache of a processor, the processor likely accesses all the bytes in the cache block before the cache block needs to be shuffled back to another processor that shares the block. In contrast, block-wise partitioning has less spatial locality, whereas column-wise partitioning has the least amount of spatial locality.

Another important consideration for spatial locality is over the granularity of pages. This issue is specific to Non-Uniform Memory Architecture (NUMA) in which physically, there are many memories in different processing nodes, hence accessing the memory takes a different latency depending on the physical distance between the memory and the processor. The hardware gives an abstraction of a single shared memory by locating the correct memory module for a given address. In NUMA, it is critical for performance to allocate a page in the memory that is the closest to the processor which will access it the most. To achieve that, typically the OS adopts several simple page allocation policies.

When a process is launched, typically the Operating System (OS) does not

4.4 Exercise Problems

1. (*10 Points*) **Amdahl's law**.

   (a) If 10% of the original computation is sequential, what is the maximum speedup of the program on 10 processors vs. on 1000 processors?

   (b) Suppose that you want to achieve a speedup of 20 with 32 processors. What fraction of the original computation can be sequential?

2. (*20 Points*) **Correctness Analysis**. Point out all correctness and/or performance problems of the following code. Then show the new code that does not have these problems.

   (a)
```
Private variables: i, x_lock;
Shared variables: tmp, x, work[], result[];

x = 0;
begin parallel
  start_iter = id * 50;
  end_iter = (id + 1) * 50;

  lock (x_lock);

  for (i=start_iter; i<end_iter; i++) {
    tmp = work[i] * work[i];
    x = x + tmp;
  }

  unlock (x_lock);

end parallel
print x;
```

   (b) A bank allows transfer from a joint checking account to a joint savings account and vice versa concurrently. This allows two account holders, Mr. X and Mrs. X, to access the accounts in parallel.

```
pragma omp parallel sections
pragma omp section
  // transfer from checking to savings
  lock (saving_lock);
  lock (checking_lock);
```

```
        saving = saving + amount1;
        checking = checking - amount1;
      unlock (checking_lock);
      unlock (saving_lock);
    pragma omp section
      // transfer from savings to checking
      lock (checking_lock);
      lock (saving_lock);
        saving = saving - amount2;
        checking = checking + amount2;
      unlock (saving_lock);
      unlock (checking_lock);
    pragma omp end sections
```

3. (20 *points*) **Parallel Programming Models**. Consider a linear transformation algorithm to compute $Y = A \times B + C$, where Y, A, B, C have a dimension $n \times p, n \times m, m \times p, n \times p$, respectively. Assume that all n, m, and p are divisible by 2. The algorithm is shown:

```
int i, j, k;
float A[n][m], B[m][p], Y[n][p], C[n][p], x;

...
// begin linear transformation
for (i=0; i<n; i++) {
  for (j=0; j<p; j++) {
    x = 0;
    for (k=0; k<m; k++)
      x = x + A[i][k] * B[k][j];
    Y[i][j] = x + C[i][j];
  }
}
```

As can be seen from the code, all the loops (for i, for j, for k) are parallel.

(a) For shared memory programming, show the appropriate and correct OpenMP directive to parallelize the "for j" loop only. Indicate where you would place the directive.

(b) For shared memory programming, show the appropriate and correct OpenMP directive to parallelize the "for k" loop only. Indicate where you would place the directive.

(c) For message passing model, parallelize the "for j" loop only. Show the resulting parallel code utilizing `send(destination-id,data)` and `recv(sender-id,data)` on two processors. Only send and receive data that is absolutely needed to be communicated. Enclose your parallel region with `begin_parallel` and `end_parallel`.

4. (*25 Points*) **OpenMP Scheduling**. Assuming that the loops shown in the code sections below are parallel, choose the appropriate OpenMP `schedule` clause which specifies the scheduling policy and the chunk size. Choose one loop to parallelize: either "for i" or "for j" loop, but not both. Assume that the number of iterations of all loops is always much larger than the number of processors p. The main goal of your scheduling choice is to *guarantee* that the load of all threads are balanced. The load of a thread is determined by how many times the innermost loop's body ("do work") the thread executes. If there are multiple solutions that meet that main goal, choose the solution with the least overheads. Assume that the number of loop iterations is never modified by statements in the loop's body.

Note that the format for such a clause is `schedule(kind[,chunk_size])`, where `kind` can be `static`, `dynamic`, or `guided`; and `chunk_size` is a loop-invariant integer expression.

(a)
```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      ... // do work
```

(b)
```
for (i=0; i<numiter[i]; i++)
    for (j=0; j<N; j++)
      ... // do work
```

(c)
```
for (i=0; i<N; i++)
    if (a[i][0] == 0)
      for (j=0; j<N; j++)
        ... // do work
```

(d)
```
for (i=0; i<N; i++)
    for (j=i; j<N; j++) //note the range!
      ... // do work
```

For this problem, use an additional assumption that you only use two threads. Use `static` scheduling but choose the chunk size such that the difference of number of iterations between the two threads is less than $x\%$ of the total number of iterations, where x is a parameter that users can determine.

(e) In `guided` scheduling in OpenMP, what does `chunk_size` represent? What are the implications of setting chunk size that is too small? What are the implications of setting chunk size that is too large?

5. (*15 Points*) **Task Queue Design**. In the class, it was mentioned that dynamic task scheduling in OpenMP utilizes a central task queue to map/assign tasks to threads.

(a) In the class, it was also mentioned that the scheduling overhead for dynamic scheduling is higher than that of a static task scheduling, provided that both the number of iterations and the number of threads are known at compile time. Explain two major sources of execution time overheads.

(b) Suppose there are two designs A and B. In design A, there is only one task queue shared by n threads. In design B, there are m task queues shared by n threads. A thread i can only grab a task from the task queue $i \textbf{ mod } m$. Give one major advantage and disadvantage of design B compared to design A.

(c) In the guided scheduling, explain why in a task queue, a larger task is always closer to the head of the queue than a smaller task, and not the other way around.

6. (*10 Points*) **Blocking Factor**. Choose the maximum blocking factor B for the following codes such that temporal reuse of matrix elements across iterations is fully accommodated. Assume that there is one level of fully associative 16KB cache with 64 byte line size implementing Least-Recently-Used (LRU) replacement policy. Assume each data element in the matrix occupies 8 bytes (thus, a cache line accomodates 8 data elements). Assume C-language memory layout.

(a)
```
double a[n][n];
for (i=1; i<n; i++)
```

```
    for (j=1; j<n; j++)
      a[i][j] = (a[i][j]+a [i-1][j]+a[i][j-1])/3;
```

(b)
```
double a[n][n];
for (i=3; i<n; i++)
  for (j=1; j<n; j++)
    a[i][j] = a [i-1][j] * a[i-2][j];
```

7. (*10 Points*) **Blocking Factor**. Choose the maximum blocking factor B for the following codes such that temporal reuse of matrix elements across iterations is fully accommodated. Assume that there is one level of fully associative 16KB cache with 64 byte line size implementing Least-Recently-Used (LRU) replacement policy. Assume each data element in the matrix occupies 8 bytes (thus, a cache line accomodates 8 data elements). Assume C-language memory layout.

(a)
```
double a[n][n];
for (i=1; i<n; i++)
  for (j=1; j<n; j++)
    a[i][j] = a[i][j] + a [i-1][j-1];
```

(b)
```
double a[n][n];
for (i=3; i<n; i++)
  for (j=1; j<n; j++)
    a[i][j] = a [i][j] * a[i-2][j];
```

8. (*10 Points*) **Performance Analysis**. Point out all performance problems of the following code. Then show the new code that does not have these problems.

```
Private variables: i, start_iter, end_iter;
Shared variables: x, y, work[], x_lock, y_lock;

x = 0;
begin parallel
  start_iter = id * 50;
  end_iter = (id + 1) * 50;
```

```
    lock (y_lock);

    for (i=start_iter; i<end_iter; i++) {

      lock (x_lock);
      x = x + work[i] * work[i];
      unlock (x_lock);

      if (work[i] > 0)
        y = y + work[i] / y;
    }

    unlock (y_lock);

    barrier
  end parallel

  print x;
  print y;
```

Chapter 5

Parallel Programming for Linked Data Structures

Contents

5.1 Parallelization Challenges in LDS 114

5.1.1 Loop-Level Parallelization is Insufficient 115

5.1.2 Approaches to Parallelization of LDS 117

5.1.3 Challenges in LDS Parallelization 117

5.2 Parallelization Techniques for Linked Lists 126

5.2.1 Parallelization Among Readers 126

5.2.2 Global Lock Approach 127

5.2.3 Fine-Grain Lock Approach 131

5.3 Exercise Problems 136

In Chapter 3, we have discussed parallel programming techniques that are applicable for applications that rely on matrices as their data structures. Accesses to a matrix are characterized by a nest of loops in which each loop iterates over elements in one dimension of the matrix. Since matrices are commonly used for scientific simulations and multimedia processing, the techniques described are useful for these applications.

The objective of this chapter is to look at parallelization techniques for non-scientific applications that mainly rely on *linked data structures* (LDS) as their main data structures. LDS include all data structures that use a collection of nodes that are linked together with pointers, such as linked lists, trees, graphs, hash

tables, etc. Many classes of applications heavily use LDS, including compilers, word processors, and database systems. Some of them, such as database systems, have a need for a very high degree of parallelism in order to be able to serve a large number of transactions simultaneously.

LDS accesses exhibit a high degree of loop-carried dependence, preventing the loop parallelization techniques from Chapter 3 to be applied successfully. Hence, they require a different set of techniques for an effective parallelization. In this chapter, we will focus on these techniques, and we will illustrate the practical application of the techniques on a linked list. Unfortunately, some LDS parallelization techniques are data structure-specific. For each type of LDS, there may be a large number of parallel algorithms for it. Since it is impossible to cover all of them in this chapter, we will focus our discussion on features that are common in these techniques and apply them to a simple LDS – a linked list. This will provide a basic framework on which readers can build further investigation into parallelization techniques for LDS.

5.1 Parallelization Challenges in LDS

Linked data structures (LDS) consists of different types of data structures such as linked lists, trees, hash tables, and graphs. The common feature of an LDS is that it has a collection of nodes which are linked to one another with pointers. A pointer is a data type that stores an address. To construct a singly linked list, for example, each node contains a pointer that stores the address of the next node in the list. For a doubly linked list, each node contains two pointers, one to store the address of the next node, and another to store the address of the previous node. A tree is organized hierarchically, with a root node at the top of the hierarchy, and leaf nodes at the bottom of the hierarchy. A node in a tree structure contains pointers that point to its children nodes, and sometimes also contains a pointer that points to its parent node. A graph is a collection of nodes in which each node can point to any nodes and may have an arbitrary number of pointers to other nodes.

While diversed, LDS traversal shares a common characteristic. It relies on reading a node to discover the addresses of other nodes it points to, read those nodes, follow next nodes that are pointed by the read nodes, and so on. Therefore, unlike the traversal of a matrix that uses an arithmetically-computed matrix index, in LDS the next node to be visited in a traversal is not known until the current node's content is read. This pattern creates a loop-carried dependence in a LDS

traversal. For the remaining of our discussion, we will illustrate parallelization of a singly-linked list, one of the simplest types of LDS.

5.1.1 Loop-Level Parallelization is Insufficient

Code 5.1 illustrates the data structure declaration of a singly-linked list, with a function `addValue` that finds a node with a key value specified in the function parameter `int key`, and adds a value `x` to the `data` field. Note that the key statement in the traversal is "`p = p->next`". When `p` points to a node, in order to get the next pointer value, the content of the node must be read before the value `p->next` is known. Therefore, there is a loop-carried dependence in the traversal loop. Such a dependence makes it hard to parallelize an LDS traversal loop.

Code 5.1 An example of a function that finds a node in a linked list and modifies it.

```
1 typedef struct tagIntListNode{
2   int key;  /* unique for each node */
3   int data;
4   struct tagIntListNode *next;
5 } IntListNode;
6 typedef IntListNode *pIntListNode;
7 ...
8 typedef struct {
9   pIntListNode  head;
10 } IntList;
11 typedef IntList* pIntList;
12 ...
13 void addValue(pIntList pList, int key, int x)
14 {
15   pIntListNode p = pList->head;
16   while (p != NULL) {
17     if (p->key == key)
18       p->data = p->data + x;
19     p = p->next;
20   }
21 }
```

There is another challenge. The loop-carried dependence may exist beyond just the traversal statement. For example, it is possible for the LDS to contain a cycle, that is there is a path from the current node to its successors until it reaches the current node again. This is common in a graph, but can also occur in linked

lists such as in a circular list. When a node is revisited in a traversal, the code may read from the node's data that was changed in an earlier iteration, such as in the statement "`p->data = p->data + x;`", creating an additional loop-carried dependence. In general, this situation is created by design by the programmer as he/she precisely knows how the LDS is organized. However, it may not be immediately apparent to performance tuners or the compiler.

Another challenge is that some LDS use recursive functions for its traversal. Recursive traversal is common in a tree. A recursive traversal is not necessarily a hindrance in exploiting parallelism. For example, in a tree traversal, we can spawn a thread to traverse the left subtree and another to traverse the right subtree. So a recursive tree traversal presents an opportunity for extracting parallelism. However, some parallel constructs such as OpenMP 2.0 directives do not provide a way to specify parallelism in a recursion.

One type of parallelism opportunity that can be exploited is between the traversal of the loop and the processing of each node traversed. For example, the processing in the above example involves reading the value stored in `p->key` and comparing it with `key`. The node processing can be executed in parallel as the traversal continues to the next node. Thus, we can have a master thread that executes the traversal, and multiple slave threads, each of them spawned when a node is visited by the master thread and it executes the node processing code. Unfortunately, in order for such parallelism to give a benefit, the node processing computation body must be large compared to the latency of traversing one node. Otherwise, the benefit from such parallelism is low. Note, however, that such an approach has been demonstrated to yield a useful prefetching effect, due to the master thread that performs the LDS traversal bringing in nodes of the LDS into the cache slightly ahead of the time the slave threads need to access them.

Considering the fundamental limitation of loop-carried dependence in the traversal loop, applying loop-level parallelization in LDS is difficult. However, there are parallelism opportunities at a higher level. If we consider an LDS as a data structure that has a set of standard operations such as searching a node, modifying the data of a node, inserting a node, and deleting a node, we can consider how such operations can be executed in parallel with respect to one another. This will be discussed in the next section.

5.1.2 Approaches to Parallelization of LDS

At the algorithm level, we can view an LDS as a data structure that comes with a set of basic primitives/operations that can be performed on it, such as *node insertion*, *node deletion*, *node search*, and *node modification*. In some LDS, there may be additional primitives such as *tree balancing* in a tree LDS. In this section, we will discuss how to exploit parallelism at such a level.

If we consider allowing various operations to be executed in parallel, we must know in what way the parallel execution is correct or incorrect. It fits our intuition that parallel execution should not produce different results compared to a sequential execution. More formally, such a constraint is defined in a concept called **serializability** [1] which states that "*a parallel execution of a group of operations or primitives is serializable if there is some sequential execution of the operations or primitives that produce an identical result.*" For example, if there is a node insertion operation and a node deletion operation that are executed in parallel, to achieve serializability, the result of the execution must be either equal to the result achieved when we sequentially perform the insertion followed by the deletion, or when we sequentially perform the deletion followed by the insertion.

The key to correct parallelization for LDS is to ensure that parallel execution of LDS operations always produces serializable outcomes. Before discussing how such techniques can be constructed, in the next section we will first discuss how unregulated parallel execution of linked list primitives can produce non-serializable outcomes.

5.1.3 Challenges in LDS Parallelization

Consider the key code portion of the node insertion and deletion for a singly-linked list which are shown in Code 5.2 and Code 5.3. In the node insertion function, the steps involved are the memory allocation of the new node through calling the `IntListNode_Create()` function, finding where to insert the node – required to keep the list sorted, and finally the insertion of the new node either at the head of the list, or between `prev` and `p`. Note that up until the last step, the list is only read and not modified. Only in the last step the list is modified by changing `prev`'s next pointer to point to the new node.

In the node deletion function, the steps involved are finding the node that contains the key value to be deleted as specified by the function argument variable

[1] The concept of serializability is used in processing transactions in parallel in database management systems (DBMS) to ensure that the values in the database remains unaffected by parallel execution.

Code 5.2 A node insertion function.

```
1 void Insert(pIntList pList, int key)
2 {
3   pIntListNode prev, p,
4     newNode = IntListNode_Create(key);
5
6   if (pList->head == NULL) { // first element, insert at head
7     pList->head = newNode;
8     return;
9   }
10
11  /* find where to insert */
12  p = pList->head;
13  prev = NULL;
14  while (p != NULL && p->key < newNode->key) {
15    prev = p;
16    p = p->next;
17  }
18
19  /* insert the node at head or between prev and p */
20  newNode->next = p;
21  if (prev != NULL)
22    prev->next = newNode;
23  else
24    pList->head = newNode;
25 }
```

key, deleting the node from the list, and finally freeing up the deleted node by calling the free() function. Note that the modification to the list occurs when the node is deleted, by changing the prev's next pointer to point to p's next pointer, effectively removing the node to be deleted from the list.

Suppose that there are two insertion operations that are executed in parallel. If handled incorrectly, they may conflict and cause a non-serializable outcome. An example is shown in Figure 5.1, in which two insertion operations conflict, with one that wants to insert a node with a key value of "4", and another that wants to insert a node with a key value of "5". Initially, the list has a node with a key value of "6" as the successor of a node with a key value of "3" (Figure 5.1(a)). Note that the variables p, prev, and newNode are local function variables. Hence, they are all allocated in the respective stack of each thread, and therefore are private to each thread. However, the nodes pointed by these variables are a part of the linked list and hence they are shared by all threads.

Code 5.3 A node deletion function.

```
1 void IntList_Delete(pIntList pList, int key)
2 {
3   pIntListNode prev, p;
4
5   if (pList->head == NULL) /* list is empty */
6     return;
7
8   /* find the node */
9   p = pList->head;
10  prev = NULL;
11  while (p != NULL && p->key != key) {
12    prev = p;
13    p = p->next;
14  }
15
16  /* delete the node if found */
17  if (p == NULL)  /* node not found */
18    return;
19  if (prev == NULL) /* delete the head node */
20    pList->head = p->next;
21  else  /* or delete a non-head node */
22    prev->next = p->next;
23  free(p);
24 }
```

Suppose that the two insertion operations are executed in parallel at almost the same stage. Both thread 0 and thread 1 have allocated the new nodes to insert, and have executed the statement "`newNode->next = p`", which link the new node's next pointers to point to node 6 (Figure 5.1(b)). At this point, suppose that thread 0 is slightly faster and executes "`prev->next = newNode`". This will link node 3's next pointer to point to the new node (node 4) (Figure 5.1(c)). Next, thread 1 also executes the same statement slightly later, and because its `prev` variable still points to node 3, it overwrites node 3's next pointer to point to the new node (node 5). The result is incorrect because node 4 has been accidentally removed from the linked list. In contrast, in a serializable execution, regardless of whether the insertion of node 4 or insertion of node 5 happens first, the final list has both node 4 and node 5.

A similar non-serializable outcome may arise between an insertion and a deletion operations. Suppose that the insertion and the deletion are executed in parallel, as shown in Figure 5.2. The figure shows that thread 0 wants to insert a node

Figure 5.1: Non-serializable outcome from parallel execution of two insertions.

with a key value of "4", while thread 1 wants to delete the node with key value of "5" from the linked list. Initially, the list has node 3, which is linked to node 5, which is linked to node 6 (Figure 5.1(a)). Suppose that both threads have finished the traversal. Thread 0 has discovered that it wants to insert node 4 after node 3, hence its `prev` pointer points to node 3. Thread 1 has found node 5, and hence its `prev` pointers points to node 3 while its `p` pointer points to node 5.

Figure 5.2: Non-serializable outcome from parallel execution of an insertion and a deletion operations.

Assume that thread 0 runs slightly ahead and executes statements that insert node 4. It changes node 4's next pointer to point to node 5, and then changes node 3's next pointer to point to node 4 (Figure 5.1(b)). Thread 1 is oblivious to the fact that the linked list has changed, and its `prev` and `p` pointers still point to node 3 and node 5, respectively. When thread 1 executes the statement that deletes node 5, it overwrites node 3 (pointed by its `prev` variable)'s next pointer to point to node 6 (pointed by its `p`'s next pointer). The result is incorrect because node 4 has now been accidentally removed from the linked list. In contrast, in a serializable execution, regardless of whether the insertion of node 4 or deletion of node 5 happens first, the final list has node 4.

Note that this is not the only possible incorrect outcome involving an insertion and a deletion operations. If the deletion of node 5 is executed first by thread 1,

prior to the insertion of node 4, then the insertion fails and node 4 is missing from the list. If thread 1 attempts to delete node 3 instead of node 5, incorrect outcomes can also result. Overall, any insertion as a successor or predecessor of a node that is to be deleted can yield to a non-serializable outcome.

We have discussed how two insertion operations, and an insertion and a deletion can conflict and produce an incorrect outcome. In a similar manner, two deletion operations can also conflict and produce an incorrect outcome (readers are invited to work out this outcome as an exercise). In general, when two operations modify the linked list simultaneously, involving nodes that are adjacent, then incorrect outcome may arise.

So far we have discussed non-serializable outcomes resulting from parallel execution of two operations that modify some nodes in the list. We have not discussed what outcome may arise when one of the operation modifies the list, while another operation only reads from the list without modifying it. Can incorrect outcome occur as a result of their parallel execution?

To answer the question, Figure 5.3 illustrates the parallel execution of a deletion and a node search operations. The node search code is similar to what is shown in Code 5.1 except that the statement that modifies the node's `data` field is omitted. Suppose that initially, thread 0, which attempts to search the node with the key value of "5", has reached node 3. Thread 1 has found node 5 and is ready to delete it. Its `prev` pointer points to node 3 and its `p` pointer points to node 5. This initial situation is illustrated in Figure 5.3(a).

Suppose that next, thread 1 performs the deletion by changing node 3's next pointer to point to node 6 (Figure 5.3(b)). In this case, when thread 0 continues its traversal by following node 3's next pointer, it will reach node 6, and continues traversal from there. The outcome of the traversal is that node 5 is not found.

Alternatively, suppose that thread 0 is slightly faster than thread 1 and follows node 3's next pointer and visits node 5. Then, thread 1 deletes node 5 by changing the node 3's next pointer to point to node 6 (Figure 5.3(c)). In this case, thread 0 reports that node 5 is found.

The two possible outcomes of thread 0 finding node 5 or not are actually not a problem as they reflect serializable outcomes. If the insertion is performed sequentially after the node search, the node search will find node 5. Alternatively, if the node search is performed sequentially after the insertion, then node search will not find node 5. Therefore, the outcomes are serializable in this case. However, note that in the former case in which node 5 is found, there is still a possible error.

Figure 5.3: Serializable outcome from parallel execution of a deletion and a node search operations.

When node 5 is deleted, and the traversal is pointing to the node, it is possible that thread 1 follows it up directly with freeing the heap memory allocated to node 5 by calling `free()`. The memory deallocation routine in the heap library may then overwrite the content of node 5. Although thread 0 has found node 5, its content may have been corrupted by the heap library, that it cannot rely on whatever it reads from node 5. Therefore, a conflict can occur not only between the basic primitives of the list, but also between them and memory management functions.

Finally, we will look at whether incorrect outcome can occur when parallel execution of a node search and a node insertion can occur. Figure 5.4 illustrates the parallel execution of an insertion and a node search operations. Suppose that initially, thread 0, which attempts to search the node with the key value of "6", has reached node 3. Thread 1 has found a place to insert a new node with a key value of "5" between node 3 and node 6. Its `prev` pointer points to node 3 and

its p pointer points to node 6. This initial situation is illustrated in Figure 5.4(a).

Figure 5.4: Non-serializable outcome from parallel execution of an insertion and a node search operations.

The code for insertion consists of a statement to set the new node's next pointer to node 6 (`newNode->next = p;`), and a statement to link node 3's next pointer to the new node (`prev->next = newNode;`). In a sequential execution, the order of these two statements does not affect the outcome. In parallel execution, however, it affects the outcome. Suppose that thread 1 links the new node to node 6 first. After that, thread 0 continues its traversal by following the next pointer of node 3 and and moves on to node 6 (the current state of the linked list is shown Figure 5.4(b)). The outcome is that thread 0 finds node 6. Finally, thread 1 updates node 3's next pointer to point to the new node (not shown in the figure). The outcome is correct in this case.

However, suppose that thread 1 first links node 3 to the new node first, before

attempting to link the new node's next pointer to node 6 (the current state of the linked list is shown Figure 5.4(c)). Then, thread 0 continues its traversal by following node 3's next pointer. Thread 0's traversal reaches node 5 (the new node), then continues traversal by following node 5's next pointer. Unfortunately, node 5's next pointer does not point to node 6 yet. The pointer may have a NULL or an undefined value. In this case, the outcome of the node search is that node 6 is not found. This is a non-serializable outcome as in sequential execution, node 6 will always be found. Overall, this case illustrates different reordering of statements in the program that does not make any difference in sequential execution can cause a very different outcome in parallel execution.

To conclude the discussion, based on the examples that we have discussed, we can make the following observations:

1. *Parallel execution of two operations that access a common node, in which at least one operation involves writing to the node, can produce conflicts that lead to non-serializable outcome.* We have illustrated this using two parallel insertions, an insertion and a deletion, and an insertion/deletion and a node search. Note that if two operations affect completely different sets of nodes, then they will not lead to a conflict.

2. *Under some circumstances, serializable outcome may still be achieved under a conflict mentioned in point (1).* We have illustrated this point by showing that node deletion and a node search operation can go on in parallel without resulting in a non-serializable outcome. Unfortunately, this is highly circumstantial and is not generally true for all types of LDS, e.g. it cannot be said that a node deletion and search in a tree or graph structure to be always serializable. Thus, exploiting such parallelism opportunity requires deep analysis of a particular LDS and algorithm that is used in the LDS operations.

3. *Conflicts can also occur between LDS operations and memory management functions such as memory deallocation and allocation.* We have illustrated that even when a node search can execute in parallel with a node deletion, we still have to ensure a node that is found is not corrupted by deallocation of the node.

Knowing the basic observations of how non-serializable outcome can arise, in the next section we will discuss parallelization techniques for the singly linked list.

5.2 Parallelization Techniques for Linked Lists

There are several parallelization techniques possible for the linked list LDS. They differ in terms of the amount of parallelism extracted and the programming complexity of the solution. In general, in LDS parallelization, a higher degree of parallelism requires a higher programming complexity. Programmers need to assess their situations carefully before deciding on which parallelization strategy best suits their needs and constraints.

5.2.1 Parallelization Among Readers

From the discussion in the previous section, we have learned that when there are two operations that affect a common node, in which one of them modifies the node, an incorrect outcome may result. If the operations do not modify the list, then they cannot produce incorrect outcome. Therefore, the most straightforward way to exploit parallelism is to allow parallel execution between read-only operations, but not between read-only and read-write operations.

Out of the primitives of the linked list, node insertion, deletion, and node modification, all modify some nodes in the list. Only node search does not modify the list. Hence, we can allow parallel execution of multiple node searches. How much parallelism can be extracted from allowing parallel execution only among readers? It depends on the usage of the LDS. In cases where an LDS structure is modified infrequently, this simple approach can yield a high degree of parallelism.

To implement this approach, we need to ensure mutual exclusion between a read-write operation and a read-only operation, but not between two read-only operations. To achieve that, we can define two types of locks: a **read lock** and a **write lock**. A read only operation must obtain a read lock prior to execution and release it after execution. A read-write operation must obtain a write lock prior to execution and release it after execution. If a read lock is already granted to an operation, a request for another read lock can be granted, but a request for a write lock cannot be granted until all read locks are released. In addition, when a write lock is already granted to an operation, a request for a read lock or another write lock cannot be granted until the current write lock is released. It is helpful to visualize this behavior using a *lock compatibility table* as shown in Table 5.1.

To implement this solution, each operation can be wrapped with a call to ob-

Table 5.1: Lock compatibility table.

Already Granted Lock	Requested Lock: Read Lock	Requested Lock: Write Lock
Read Lock	*Yes*	*No*
Write Lock	*No*	*No*

tain a read or write lock, depending on the operation type, and a release of the lock at the end of the operation. For example, the first statement of the function that executes a node search will be `acq_read_lock()`, and the last statement will be `rel_read_lock()`. Of course, for any early exit out of the function, the lock release function must also be called. The lock itself can be implemented using a regular data structure that keeps track of what lock is currently acquired, and if it is a read lock, how many of them are acquired. Accesses to the data structure can be protected using a regular lock.

■ *Did you know?*

For transaction processing in database management system (DBMS), read and write locks are only a subset of lock types that are used. Another lock type is the upgrade lock, *used for avoiding a deadlock when multiple transactions hold read locks for a single object and both want to upgrade the lock to a write lock. There are also* intention read *and* intention write *locks which are used for nested locking mechanisms, for example, rather than read locking all records in a table one at a time, a transaction can request an intention-read lock on the table itself.*

5.2.2 Global Lock Approach

While the previous approach enables parallelism among read-only operations, a higher degree of parallelism can be obtained if parallelism is also allowed between a read-write operation and other operations. There are at least two approaches for providing parallelism in this case. The finer grain approach is to associate a lock with each node in the list. Individual nodes affected by an operation can be locked separately. A simpler approach uses a global lock, that is, one lock for the

entire LDS. The fine-grain lock approach introduces complexities in managing the locks, such as avoiding or handling deadlocks and livelocks. Using a single lock for the LDS avoids deadlocks and livelocks since no cyclic dependence in lock acquisition can occur when there is only one lock.

The key observation in allowing parallelism between read-write operations is to decompose the operations into components that are only read from the linked list, and components that modify the linked list. Logically, a linked list insertion or deletion involve a traversal to determine where to insert a node or where the node to be deleted is located. Once they are located in a traversal, modifications are made. Since the traversal only reads from the linked list, we can allow multiple operations doing the traversals at the same time. Only when the operation needs to be modify the linked list, it acquires a lock and makes the modification. Note, however, as discussed in Section 5.1.3, incorrect outcome can arise when the linked list is changed (by another thread) between the time a thread finishes traversal and the time when it makes modification to the list. Therefore, when a thread obtains a lock to modify the list, it must ensure that the nodes that will be modified have not changed since the last time they are read. If they have not changed, then the modification can be performed. Otherwise, the operation must either be canceled or repeated.

To be more specific, consider the insertion operation shown in Code 5.2. The traversal portion finds where the new node is to be inserted, i.e., between the node pointed by the `prev` variable and the node pointed by the `p` variable. When the thread that performs the operation obtains the lock, it must ensure that (1) no node has been inserted between the nodes pointed by `prev` and `p`, and (2) the node pointed by `prev` and `p` are still valid, i.e. they have not been deleted. If any of the condition is not met, the traversal must be repeated to rediscover the location in which the new node should be inserted.

Code 5.4 shows the new node insertion code. The linked list node is augmented with an additional field `deleted` that indicates whether a node has been deleted or not. This field is added so that in a node deletion, a node is marked deleted but not deallocated. Deallocation of the node is deferred until it is safe to do so, such as when no threads point to the deleted node.

After the traversal, the thread first acquires a write lock, ensuring that no other threads also attempt to modify the list simultaneously. It then checks that no node has been inserted between `prev` and `p` by checking whether `prev`'s next pointer still points to `p`. It also checks whether any of nodes pointed by `prev` or `p` has

Code 5.4 A parallelizable node insertion function.

```
1 typedef struct tagIntListNode{
2   int key;  /* unique for each node */
3   int data;
4   struct tagIntListNode *next;
5   int deleted;  /* =1 if deleted, =0 otherwise */
6 } IntListNode;
7 typedef IntListNode *pIntListNode;
8 ...
9 void Insert(pIntList head, int key)
10 {
11   ... /* traversal code to find where to insert a new node */
12
13   /* insert the node at head or between prev and p */
14   acq_write_lock();  /* enter critical section */
15   if (prev->next ! = p || prev->deleted || p->deleted) {
16     ... /* repeat traversal */
17   }
18   /* by this point, prev and p are valid. Insert the node */
19   newNode->next = p;
20   if (prev != NULL)
21     prev->next = newNode;
22   else
23     pList->head = newNode;
24   rel_write_lock();
25 }
```

been deleted by checking the value of the `deleted` field of both nodes. If any of the condition is true, the traversal is repeated to rediscover the correct location to insert the node. The repeat of the traversal is performed inside the critical section, so we can be sure that no other threads modify the list while re-traversal is performed. Finally, the node is inserted and the lock is released.

A node deletion can be parallelized using a similar strategy, as shown in Code 5.5. After a traversal to locate the node to be deleted, a write lock is acquired. To ensure that the list has not changed between the time of traversal until this point, we can check to make sure that no node has been inserted between `prev` and `p` by checking whether `prev`'s next pointer still points to `p`. It also checks whether any of nodes pointed by `prev` or `p` has been deallocated. In either case, the traversal is repeated to rediscover the correct predecessor of the node that will be deleted. When finally node pointed by `p` is deleted, it is not directly deallocated to avoid corrupting the node, in case there are other threads that rely

on the values stored in the node. Rather, the node is marked as deleted.

Code 5.5 A parallelizable node deletion function.

```
void IntList_Delete(pIntList pList, int x)
{
  ...  /* code that traverses the list to find the node
       to be deleted */

  /* now, perform the deletion */
  if ... /* code for deleting head node */
  else  { /* delete a non-head node */
    acq_write_lock();
    if (prev->next != p || prev->deleted || p->deleted) {
      ... /* repeat traversal */
    }
    /* by this point, prev and p are valid */
    if (prev == NULL) /* delete head node */
      pList->head = p->next;
    else  /* delete non-head node */
      prev->next = p->next;
  }
  p->deleted = 1;  /* mark it as deleted */
  rel_write_lock();
}
```

When is it safe to deallocate and reclaim the storage of a node? The minimum requirement is that the node is not currently involved in any operation, that is, no threads have any active pointers pointing to the node. However, keeping such information requires keeping a reference count on each node, which is costly. A simpler alternative is to wait until all current pending operations complete, and then schedule a garbage collection phase that deallocates and reclaims all deleted nodes.

In the code for node insertion and deletion, only the write lock is used. In what situations should the read lock be used? The read lock should be used in operations that rely on the values of a node to be unchanged during the operations for correctness. For example, consider an operation that searches for a node and reads the data stored in the node. During traversal to locate the node, no lock is needed. However, once the node is located, it needs to be read locked until the data has been read. This prevents a concurrent operation from write locking the node to change its data. With the lock, the concurrent operation will stall until the read lock has been released by the reading operation.

How much parallelism can we expect from the global lock approach? Suppose that the time taken to locate a node through traversal is T_{trav}, and the time taken to actually modify the node is T_{mod}. As the linked list adds more nodes and increases in size, T_{trav} increases along with the linked list size as well (for random searches). T_{mod}, on the other hand, remains constant as the modifications to the list still involves only one or two nodes. By keeping the traversal parallel, the global lock approach parallelizes a larger fraction of the execution time as the data structure size increases. Therefore, for very large data structures, the parallelism benefit (and potential speedup) from the global lock approach is quite high as long as the time to acquire the global lock remains constant.

Generalizing to types of LDS, the global lock approach is relatively easy to implement regardless of the type of the LDS. This is because we have only one lock and do not need to handle deadlocks and livelocks. What is needed to apply the global lock approach is to write each operation on the LDS in a way that traversal can be separated with the code that modifies the LDS.

5.2.3 Fine-Grain Lock Approach

While the global lock approach allows multiple operations to execute in parallel, it is still limited in the sense that only one thread is allowed to modify the linked list at a time. Even when different threads modify different parts of the linked list, they have to perform the modification sequentially in the critical section. In this section, we will look at how modifications to different parts of the list can be allowed to proceed in parallel.

To achieve that, we need a finer lock granularity. Rather than using a single global lock, we can associate a lock with each node. The basic principle is that when an operation needs to modify a node, it locks the node so that operations that want to write or read from the node cannot do so, but other operations that write or read other nodes can proceed in parallel without a conflict.

The question is now how to figure out what nodes to lock for each operation. To answer the question, we first need to distinguish between nodes that will be modified in an operation, versus nodes that will only be read in an operation but must remain valid in order for the operation to complete correctly. The principle for handling them is that: *nodes that will be modified must be write locked*, while the *nodes that are read and must remain valid must be read locked*. Note that locking both types of nodes too strongly (e.g., using write locks) is largely inconsequential as it only results in reduced concurrency. However, locking them

too weakly (e.g., using read locks) affects the correctness of the operation. Hence, this analysis must be performed carefully.

For the node insertion operation, the node pointed by `prev` will be modified to point to the new node, so it needs to be write locked. The node pointed by `p` will not be modified, but it must remain valid for the operation to be performed correctly, e.g. it cannot be deleted before the operation is completed. Therefore, it needs to be read locked.

For the node deletion operation, the node pointed by `prev` will have its next pointer changed, so it must be write locked. The node pointed by `p` will later be deleted, so it needs to be write locked. Note that the successor node (the node pointed by `p`'s next pointer) must also remain valid for the delete operation to complete correctly. The reason is that if it is deleted, then at the end of the operation, `prev`'s next pointer will point to a deleted node, which is incorrect. According to the rules, the node must be read locked. However, in this special case, it is unnecessary to read lock the successor node because write locking the node pointed by `p` already ensures that the successor node cannot be deleted, as the deletion operation must also write lock the node pointed by `p`, and the lock acquisition attempt will fail.

After all involved nodes are locked appropriately, the next step before modifying them is to retest the validity of the nodes, in the same way as with the global lock approach. This is because in between the time of the traversal and the first lock acquisition, as well as between lock acquisition of various nodes, the linked list may have been changed by another thread.

Code 5.6 and 5.7 show the node insertion and deletion using the fine-grain locking approach, respectively. The code shows a loop that performs node lock acquisition, followed by a test of node validity, and if the test fails, the locks are released and the traversal repeated. When the code gets out of the loop, the locks have been successfully acquired and the nodes are valid.

One caution in implementing the fine grain locking approach is that deadlocks are a possibility if we are not careful with the order of lock acquisition. In the shown implementation, the lock acquisition is always performed from the left-most node first. If, on the other hand, the node insertion acquires locks starting from the left-most node but the node deletion acquires locks starting from the right-most node, then cyclic dependence of lock acquisition between node insertion and deletion can occur and cause a deadlock. An alternative approach to ensure a uniform order of lock acquisitions for all threads is to acquire locks in

the order of increasing or decreasing node addresses. This approach is useful when the data structure itself does not have natural ordering information, such as graphs.

Code 5.6 Parallelization of node insertion using the fine-grain locking approach.

```
void Insert(pIntList head, int x)
{
  int succeed;
  ... /* traversal code to find where to insert */

  /* insert the node at head or between prev and p */
  succeed = 0;
  do {
    acq_write_lock(prev);
    acq_read_lock(p);
    if (prev->next ! = p || prev->deleted || p->deleted) {
      rel_write_lock(prev);
      rel_read_lock(p);
      ... /* here, repeat traversal, return if not found */
    }
    else
      succeed = 1;
  } while (!succeed);

  /* by this point, prev and p are valid, so insert the node */
  newNode->next = p;
  if (prev != NULL)
    prev->next = newNode;
  else
    pList->head = newNode;
  rel_write_lock(prev);
  rel_read_lock(p);
}
```

Comparing the fine-grain lock approach with the global lock approach, for the singly linked list they are not much different in programming complexity. This is because of the regularity of the linked list structure makes it is easy to separate the following three steps: (1) traversal, (2) locking of nodes that will be modified or relied upon as valid, and (3) the modification of the nodes. We can cleanly separate the steps because we can easily identify what nodes will be written or read prior to modifying the nodes. For more complex types of LDS, the fine-grain locking approach will be more complex as well because the three

Code 5.7 Parallelization of node deletion using the fine-grain locking approach.

```
void IntList_Delete(pIntList pList, int x)
{
  int succeed;
  ... /* code that traverses the list to find the node
      to be deleted */

  /* node is found, perform the deletion */
  succeed = 0;
  do {
    acq_write_lock(prev);
    acq_write_lock(p);
    if (prev->next ! = p || prev->deleted || p->deleted) {
      rel_write_lock(prev);
      rel_write_lock(p);
      ... /* here, repeat traversal, return if not found */
    }
    else
      succeed = 1;
  } while (!succeed);

  /* by this point, prev and p are valid */
  if (prev == NULL) { /* delete head node */
    acq_write_lock(pList);
    pList->head = p->next;
    rel_write_lock(pList);
  }
  else  /* delete non-head node */
    prev->next = p->next;

  p->deleted = 1;  /* don't deallocate, mark it deleted */
  rel_write_lock(prev);
  rel_write_lock(p);
}
```

steps may not be as easy to separate. For example, in a tree LDS, the algorithm that performs tree balancing operation hardly knows how many nodes will be written or read until the balancing itself is being carried out. In addition, node modifications and locking may be intermingled unless the algorithm is designed to separate them. A similar issue arises with node insertion and deletion in a balanced tree. Furthermore, determining the ordering of node acquisition that avoids deadlocks is also more challenging because in the natural algorithms of

some operations, lower level nodes may be visited first (thus need to be locked first), while in others, higher level nodes may be visited first (thus need to be locked first). For these cases, the algorithm may have to be split into two phases: a phase in which we figure out which nodes will need to be locked and/or modified, and a phase in which we actually lock the nodes and perform the modification.

Overall, we have shown that parallelization techniques for LDS are very different than parallelization techniques for loop-based scientific applications. Whereas loop-level parallelization is very fruitful in scientific applications, it is not so fruitful in LDS. In addition, loop-level parallelism is easy to express using simple directives. Thus, it lends itself to the ease of exploiting parallelism without many changes to the source code or the algorithms. Unfortunately, exploiting parallelization at the LDS primitives level requires many changes to the algorithms and source code that implement the LDS primitives.

5.3 Exercise Problems

1. (*10 points*) **Non-serializable outcome**.

   In a singly-linked list, show how two deletion operations can produce non-serializable outcome. Be specific in showing all possible outcomes.

2. (*10 points*) **Non-serializable outcome**. In a singly-linked list, suppose that a deletion operation attempts to delete a node from the list (i.e., node x), while an insertion operation attempts to insert a node between node x and its successor node. If they execute in parallel without any locks, what are all the possible outcomes?

3. (*30 points*) **Doubly-linked list**. In a doubly-linked list, each node has two pointer fields: `pred` which points its predecessor node, and `succ` that points to its successor node.

   (a) Show the pseudo code for node insertion and node deletion.

   (b) Specify what nodes need to be write locked and read locked for an insertion operation.

   (c) Specify what nodes need to be write locked and read locked for a deletion operation.

4. **Deadlock Avoidance**. In order to avoid the possibility of a deadlock in a fine-grain lock approach, typically the nodes are globally ordered so that the order of lock acqusition on nodes performed by multiple threads cannot form a cycle. For a graph or tree algorithms, one way to achieve this is to acquire locks in the order of ascending (or descending) node addresses. What constraints does this lock acquisition ordering place on algorithms and programming complexity?

5. **Read vs. Write Locks**. Replacing all read locks with write locks simplifies programming of a fine-grain lock approach, but at the same times reduces concurrency. Estimate the degree of the loss of concurrency on a linked list data structure, assuming that each node has a random but equal probability of being deleted or linked to a new node inserted as its successor. Assume that insertions and deletions balance out each other, keeping the number of nodes in the linked list relatively constant at n nodes.

6. **Unsafe deletion**. In the fine-grain lock approach, explain why the following code snippet for deleting a node is not safe to use:

```
// ... lock acquisition here
// ... assumptions  found to be valid here
prev->next = p-> next;
p->next = NULL;
p->deleted = 1;
```

Propose two solutions of how this problem can be avoided.

7. **ABA problem**. One challenge that the fine grain lock approach faces is referred to as the ABA problem. Suppose we would like to delete a node from a linked list at an address pointed by p, having a key value 5. The deletion thread T_1 has found the node, but before it can perform the deletion, another thread T_2 successfully performs two operations. The two operations are: (1) delete the node pointed by p and free up its memory chunk, and (2) reuse the chunk as a new node having a key value of 10 and insert the new node into the list. Suppose that the new node is inserted by T_2 into the list at exactly the same position as the old node was. When T_1 continues with acquiring locks and testing its assumptions, it finds that the assumptions are valid, i.e., the node pointed by `prev` and `p` are not in a deleted state, and `prev->next == p`. Thus, T_1 deletes the node with a key value of 10, rather than the one it intended to delete. Give three possible solutions to avoid this ABA problem, and list their advantages and disadvantages.

Part II

Designing Parallel Computer Architectures

Chapter 6

Introduction to Memory Hierarchy Organization

Contents

6.1 Motivation for Memory Hierarchy 142
6.2 Basic Architectures of a Cache 144
6.2.1 Placement Policy 144
6.2.2 Replacement Policy 149
6.2.3 Write Policy 151
6.2.4 Inclusion Policy on Multi-Level Caches 153
6.2.5 Split Cache Organization 155
6.2.6 Translation Lookaside Buffer and Cache Addressing . 155
6.2.7 Non-Blocking Cache 158
6.3 Cache Performance 159
6.3.1 The Power Law of Cache Misses 162
6.3.2 Stack Distance Profile 164
6.3.3 Cache Performance Metrics 165
6.4 Prefetching 168
6.4.1 Stride and Sequential Prefetching 170
6.4.2 Prefetching in Multiprocessor Systems 172
6.5 Exercise Problems 174

The objective of this chapter is to give a brief overview of the memory hierarchy used in current computer systems, focusing primarily on PC and server architectures. In these architectures, the memory hierarchy consists of levels of temporary storage managed by hardware, referred to as *caches*. In embedded

systems, the memory hierarchy often consists of software-managed, rather than hardware-managed, storage referred to as *scratchpad memory*. This chapter will focus mostly on caches.

In previous chapters, we have discussed how parallel programs are constructed. To get the most performance from a parallel program, it is necessary to tune it to conform to the machine architecture on which it runs. One of the biggest obstacles in parallel program performance tuning is probably the complex memory hierarchy present in current computer systems. It makes a difference in performance where a thread can get most of its data from: from an upper level cache with an access time of only a few processor clock cycles, from a lower level cache with an access time of tens of clock cycles, from the local main memory with an access time of hundreds of clock cycles, or from a remote memory with an access time of up to thousands of clock cycles. Therefore, a good understanding of how memory hierarchy is organized is crucial for (1) knowing how to tune the performance of parallel programs, and (2) knowing what other hardware support is needed to build a shared memory multiprocessor. It is the purpose of this chapter to give a brief overview of the basic design issues of caches.

6.1 Motivation for Memory Hierarchy

For decades, the increase in CPU speed has been much faster than the decrease in the access latency of the main memory. Up until a few years ago, CPU speed as measured in its clock frequency grew at the rate of 55% annually, while the memory speed grew at the rate of only 7% annually [Hennessy and Patterson, 2003]. This speed gap produced an interesting implication. While in the past, a load instruction could get the needed datum from main memory in one CPU clock cycle, in recent systems it requires hundreds of processor clock cycles to get a datum from the main memory. Dependences between a load instruction (producer) and instructions that use the loaded value (consumers) dictate that the consumer instructions must wait until the load obtains its datum before they can execute. With the latency of loading datum from the main memory in the order of hundreds of cycles, the CPU will stall for much of that time because it runs out of instructions that are not dependent on the load. Hence, it is critical to performance that most data accesses are supplied to the CPU with low latencies. Caches provide such support.

A cache is a relatively small and temporary storage for keeping data that is

likely needed by the requestor. The concept of a cache is universal since it can be used as a software construct or a hardware component. In this chapter, we focus on hardware caches that exist between the processor and the main memory.

Current memory hierarchy looks something like that shown in Figure 6.1. It shows a configuration in which there are two processor cores on a chip. Each core has a private Level 1 (L1) data cache and a Level 1 instruction cache. The cores share a single Level 2 (L2) cache that holds both instructions and data. The typical range of access latencies in 2007 (in terms of CPU clock cycles) of each cache is shown in the figure. Due to the substantial difference between the access latency to the L2 cache and to the main memory, sometimes a Level 3 cache is employed.

Figure 6.1: A memory hierarchy configuration that is typical in 2007.

For an example of the memory hierarchy of a recent processor, the IBM Power6 [Le et al., 2007] has two processor cores on a die. Each core has 2-way **simultaneous multithreading** (SMT), which means that it can execute two threads simultaneously by fetching from two different program counters. Most of the processor core resources such as register files and functional units are shared by the two threads. A Power6 die also has two cores, so there are a total of four threads that can run simultaneously. Each of the core has a 64KB L1 instruction cache and a 64KB L1 data cache. Each core also has a private 4MB L2 cache, so in total the two L2 caches have 8MB of storage. The L3 cache is located outside the chip

but its tag is located on chip. Multiple chips can be interconnected to form a larger shared memory system.

Due to temporal locality behavior naturally found in programs, caches are an effective structure to keep most useful data closer to the processor. By simply keeping more recently used data in the cache while discarding less recently used data, caches achieve that goal. For example, if data access is completely random, the **hit rate** of each cache (the fraction of accesses that find their data in the cache) will be proportional to the ratio of the cache size to the size of the program working set (e.g., miniscule for small caches). However, cache hit rates are often much higher than that, 70-95% hit rates are not uncommon both for the L1 and L2 caches.

6.2 Basic Architectures of a Cache

Figure 6.2 shows a simple analogy in which at a very high level, a cache structure can be thought of as a table, with multiple rows referred to as **sets** or **congruence classes**, and multiple columns referred to as **ways**. To exploit spatial locality and reduce management overheads, multiple bytes are fetched and stored together as a **cache block**, which is equal to a cell in the table analogy. The basic parameters of a table are the number of rows, the number of columns, and the cell size. Similarly, the basic parameters of a cache structure are therefore the number of sets, **associativity** (or the number of ways), and the cache block size. **Cache size** can be computed simply by multiplying the number of sets, associativity, and the block size. Since cache size often has a more direct impact on performance than the number of sets, typically cache parameters are specified in terms of cache size, associativity, and block size.

To manage a cache, several aspects must be considered. The first is **placement policy** that decides where a memory block can be placed in the cache. The second is **replacement policy** that decides which block needs to be evicted to make room for a new block. The third is **write policy** that decides that when parts of a cache block is written, when the new value should be propagated to the lower level memory hierarchy component.

6.2.1 Placement Policy

The placement policy decides where a block in memory can be placed in the cache. This decision affects performance significantly, since a block stored in

Figure 6.2: Analogy of a table and a cache.

the cache must be locatable in future accesses to the block. To simplify the process of searching a block, the placement of a block is restricted to a *single* set, although it can be placed in any way in that set. Therefore, the organization of the cache in terms of number of sets and ways determine the placement of a block. This is illustrated in Figure 6.3. At one extreme, a block in memory can only be stored at a particular block in the cache. Such a policy requires a **direct-mapped** cache organization in which we only have one-way associativity. At the other extreme, a block in memory can be stored at any of the blocks in the cache. Such a policy requires a **fully-associative** cache organization in which we only have one set. A direct-mapped cache easily suffers from conflicts in which multiple blocks in memory map to the same cache block and end up evicting each other. A fully-associative cache does not suffer from conflicts but is expensive to implement due to the need to search all ways when a block needs to be found. A compromise is to limit the associativity using a **set-associative** cache organization (Figure 6.3 shows a 4-way associative cache). In a set-associative cache, a block can be kept at any way in a single set. A set-associative cache organization is much more resistant to conflicts than a direct-mapped cache, and yet is much cheaper to implement than a fully-associative cache.

To locate a block in the cache, a cache employs three steps:

1. Determine the set to which the block can map.

Figure 6.3: Various cache associativities.

2. Access the set to determine if any of the blocks in the set matches the one that is being requested.

3. If the block is found, find the byte or word that is requested, and return it to the processor.

The first task, determining the set to which a block can map, is accomplished through a **cache indexing function**. Typically, the cache indexing function used is simple: the index to the set is obtained by taking the modulo of the block address by the number of sets:

$$setIdx = blkAddr \ \mathbf{mod} \ numSets \tag{6.1}$$

If the number of sets is the power of two, then the modulo operation does not require any computation as the result is simply a few lower bits of the address (the index bits in Figure 6.4). The advantages for using a simple modulo indexing function are that determining the correct set index does not require computation, and contiguous blocks in memory are mapped to different sets – which helps in

distributing memory blocks evenly across sets in the cache. How many index bits are needed depend on the number of sets in the cache. For example, if there are 2048 sets, then the number of bits that are used for locating the correct set is $log_2(2048) = 11$ bits.

Figure 6.4: How different parts of an address are used for locating a block in a cache.

In the second step, after the set is located, we need to know whether the block being accessed actually resides in the cache or not. There are many blocks that can map to a particular set because the main memory is much bigger than a cache. Therefore, when a block is stored in the cache, the cache must remember which blocks are stored so that it has the ability to check whether the requested block is one of the blocks stored in a cache set. Multiple blocks that map to the same set will have the same values in the index bits, so for that purpose, when a cache block is stored, the address bits that are not part of the block offset or index bits must be stored along with the block. These bits are referred to as the **tag** bits (Figure 6.4).

Finally, if the block is found in the cache, then the addressed byte or word needs to be selected from the cache block and returned to the processor. For this purpose, the lowest few bits are used for locating a particular byte/word in a cache block. For example, if the block size is 64 bytes, to locate any byte in the cache block, we need $log_2(64) = 6$ lowest-order bits.

The process of locating a block is further illustrated in Figure 6.5. The figure shows a two-way associative cache bank. A cache bank contains an address decoder for mapping an address to the set index, tag and data arrays, and sense amplifiers that temporarily store a block that has been read from the data array. A large cache may contain more than one cache bank. Three steps are involved in locating data: accessing the set that the block may map to (circle 1) in the tag array. The tag array stores tag bits for blocks that are currently cached. The two tags in the set are pulled down for comparison with the tag bits of the requested addressed (circle 2). If any of them matches, then we have a cache hit, else we

have a cache miss. When we have a cache hit, the data block corresponding to the way in which the hit was found is chosen, and the requested bytes or words are selected using the block offset bits of the address, and returned to the processor (circle 3). In a high-performance cache, these steps are pipelined, so that multiple accesses can be served (at different stages) by the cache simultaneously.

Figure 6.5: Locating a block in the cache. A two-way associative cache bank is shown.

Note that the steps of accessing the tag array and the data array can be performed sequentially, or both in parallel. The advantage of accessing them sequentially is that after a cache hit to a way is determined, the cache only accesses the way of the data array in which the hit occurs. So this approach is more power efficient. The alternative to it is to access both the tag array and data array in parallel. Such an approach is faster since by the time hit or miss is determined,

data blocks from all ways have been read and the remaining thing to do is to select the correct block and the correct byte or word from the block. However, it is less power efficient because all cache ways of the corresponding set in the data array are always activated and read, even when the access results in cache misses. Such a power-performance trade-off is common in cache design.

6.2.2 Replacement Policy

Suppose that we have a set-associative cache. When a new block is read from the lower level hierarchy and needs to be placed in the cache, it needs to evict a block from among blocks currently cached in the set. The criteria for an optimum replacement [Belady, 1966] is that *the block that will be accessed the farthest in the future should be the one that is evicted.* While the optimum replacement is theoretically guaranteed to produce the least number of cache misses, it is not implementable because it requires information from the future. Hence, most cache implementations use a **Least Recently Used** (LRU) replacement or its approximations. In the LRU policy, the cache block that was accessed the farthest back in the past is selected to be evicted to make room for a new block. The LRU replacement exploits temporal locality in code because recently accessed data tend to be accessed again in the near future. It has been shown that LRU performs very well overall, although it trails the performance of optimum replacement, sometimes by a large margin. There is a case in which LRU performance is pathological: when a program accesses more blocks in a circular pattern than what the cache can hold, LRU replacement performs terribly. For example, suppose that the program accesses block A, B, C, A, B, C, A, and so on. If all the blocks map to a single cache set, and the cache has a 2-way associativity, then every block access incurs a cache miss.

LRU can be implemented cheaply for lowly-associative caches, but is expensive to implement for highly-associative caches, because the order of recent accesses must be maintained. As a result, designs for highly-associative caches use approximations to LRU instead of true LRU. One such approximation is the **first in first out** (FIFO) policy, which evicts a cache block that was brought into the cache the earliest, regardless of whether the block was recently used or not.

Figure 6.6: Illustration of Pseudo-LRU replacement on a 4-way set associative cache: the initial state (a), the updated tree after an access to block B (b), and the block chosen for replacement (c).

> ■ *Did you know?*
>
> *For more discussion on various alternative cache indexing functions and their impact on performance, readers are referred to the study by Kharbutli and Solihin [Kharbutli et al., 2004].*
>
> *The performance of LRU is not* graceful, *i.e. when the working set of a program becomes larger than the cache, the performance may degrade abruptly rather than gradually. Replacement policies that rely more on randomness often exhibit more gracefulness, although they perform slightly worse than LRU on many cases. Readers are referred to a study by Guo and Solihin [Guo and Solihin, 2006] for more discussion.*

Another possible approximation to LRU is by keeping track of only a few most recently used blocks, and on replacement, a block from among blocks that are not the few most recently used ones is selected, either randomly or based on a cer-

tain algorithm. The simplest example of this approach is a scheme that tracks the most recently used (MRU) block, and on a replacement, pseudo-randomly selects a block from among non-MRU blocks. Another example that belongs to this approach is the tree-based pseudo LRU replacement policy [Al-Zoubi et al., 2004], illustrated in Figure 6.6.

In the pseudo-LRU replacement policy, for an A-way associative cache, $A-1$ bits are maintained and logically organized as a binary tree. The purpose of the tree is to record the direction of recent accesses. During accesses, the direction from the root of the tree to the accessed block is recorded in the nodes of the tree, with '0' indicating the left direction and '1' indicating the right direction. For example, suppose that initially the bits in the tree have the values shown in Figure 6.6(a). If block B is accessed, the path from the root of the tree is going to the *left*, followed by going to the *right*. Since the left direction is represented by a '0', the root node's value is reset to 0. Since the right direction is represented by a '1', the value in the left child node of the root is set to 1 (Figure 6.6(b)). When a replacement needs to be made, the tree is traversed from the root to the leaf, and the leaf node is the block that is selected for replacement. During the traversal, the meaning of the bit values is switched. The value '0' now indicates going to the right child, whereas '1' indicates going to the left child (Figure 6.6(c)). In the example, block D is selected for replacement.

6.2.3 Write Policy

Write Through vs. Write Back

A cache is a temporary storage. The main storage for programs is the main memory. So intuitively, an important question in cache design is that when a processor changes the value of bytes at a cache block, when should that change be propagated to the lower level memory hierarchy? There are two choices for implementation: **write-through** and **write back** policies. In a write through policy, any bytes written by a single write event in the cache are immediately propagated to the lower level memory hierarchy component.

In contrast, in a write-back policy, the change is allowed to occur at the cache block without being propagated to the lower level memory hierarchy component. Only when the block is evicted, is the block "written back", or propagated and it overwrites the stale version at the lower level. In the write back policy, if every evicted block is written back, bandwidth consumption may be high. Hence, an obvious optimization is to write back cache blocks that have been modified while

they are in the cache. To achieve this, each cache block's tag is augmented with a **Dirty** bit. When a new cache block is installed in the cache, its dirty bit is cleared. When there is a write to any byte or word in the block, the dirty bit is set to one. When the cache block is evicted, it is written back if its dirty bit has a value of one, or *silently* discarded if its dirty bit has a value of zero.

Comparing the write through versus write back policies, the write back policy tends to conserve bandwidth usage between the cache and its lower level memory hierarchy component because there are often multiple writes to a cache block during the time the cache block resides in the cache. In the write-through policy, each write propagates its value down so it consumes bandwidth on each write. As a result, the lowest level cache in a processor chip typically uses a write back policy since off-chip pin bandwidth is a limited resource. However, the choice is not as clear for on-chip bandwidth which tends to be much higher. For example, the L1 cache is backed up by a lower level L2 cache, and both of them are on-chip. The bandwidth between the L1 cache and L2 cache is quite high compared to the off-chip bandwidth.

Another factor that determines the choice of write through vs. write back policy is fault tolerance against hardware faults. A bit stored in the cache can flip its value when it is struck by alpha particles or cosmic rays (a phenomenon called **soft errors**). The cache cell is not damaged but it loses its value. In a write-through cache, if this fault is detected, then it is safe to discard the block and re-fetch it from the lower level hierarchy. However, in a write-back cache, detecting the fault is not enough because the only valid copy of the data has been changed (the lower level memory hierarchy component does not have a valid copy yet). Hence, a write-back cache needs to be protected not just by error detection, but also by **error correction**. Redundant bits called **error correcting code** (ECC) are added to enable error correction. However, since ECC computation is more expensive than simple **parity bits** which can detect but not correct errors, ECC can negatively impact cache access latencies. Therefore, in many implementations, the L1 cache uses a write-through policy (so that it can use simpler parity bit protection), while the L2 cache uses a write back policy (so it can conserve off-chip bandwidth) and is protected with ECC.

Write Allocate vs. Write No Allocate

Another aspect of the write policy is whether the block that contains the byte or word to be written needs to be brought into the cache if it is not already in the

cache. In the **write allocate** (WA) policy, upon a write miss, the block is brought into the cache before it is written. In the **write no-allocate** (WNA) policy, upon a write miss, the write is propagated down to the lower level memory hierarchy component without bringing the block into the cache.

The write no-allocate policy is more beneficial when there is only a small probability that adjacent bytes in the block will be read or written in the near future, i.e. there is little spatial locality after a write. It is also more beneficial when there is only a small probability that the written bytes in the block will be read or written again in the near future, i.e. there is little temporal locality after a write. If the written bytes or adjacent bytes in the block will be read or written after a write in the near future, then a write allocate policy works better. Since increasing the cache block size increases the probability that a block will be accessed multiple times, the write allocate policy tends to perform better relative to the write no-allocate policy on larger cache block sizes.

A write through policy may use a write allocate or write no-allocate policy. However, a write no-allocate policy is more natural since the writing through can be performed regardless of whether the block is found in the cache or not. On the other hand, a write-back cache typically uses write allocate policy since without a write allocate policy, a write miss will have to be propagated down similar to a write through cache.

6.2.4 Inclusion Policy on Multi-Level Caches

In a multi-level cache configuration, another issue that is relevant is whether the content of the upper level cache (which is smaller) should be contained in the lower level cache (which is bigger). If a lower level cache includes the content of the upper level cache, then the cache is said to maintain an **inclusion property**. The benefit of maintaining inclusion is that in a multiprocessor system, the party that checks whether a cache keeps a copy of a memory block is not just the processor that has the cache, but also other processors. These processors send a request first to the lower level cache, which must respond if it has a copy of the block. With an inclusion property, if the lower level cache does not have a copy of the block, it is guaranteed that the upper level cache also does not have a copy of the block, allowing faster and simpler checking. When the inclusion property is not enforced, both the lower level cache and upper level caches must be checked to see if any one of them has a copy of the block. The problem with this is that an upper level cache such as the L1 cache is highly *timing sensitive*,

i.e. it must have an access time that is only a few processor clock cycles or the processor performance suffers. Allowing the processor and external requests to check it simultaneously results in contention which may slow down access by the processors. Hence, it may be necessary to provide two tag arrays for the upper level cache with one tag array used by the processor and the other used by external requests. Of course, this solution does not completely remove the contention problem because while simultaneous tag checks can be supported, a change in one tag copy must be propagated to the other tag copy, making both tags unavailable in the mean time.

The drawback of enforcing inclusion, however, is that the total number of cache blocks that can be held is dictated by the size of the lower level cache, rather than by the total size of the lower and upper level caches. If the lower level cache size is relatively small, then the wasted cache capacity can be significant. For example, if we have 32-KB L1 data cache, 32-KB L1 instruction cache, and a 512-KB L2 cache, then the wasted space is 64-KB, which is equivalent to 12.5% of the L2 cache size. But if we have a 2-MB L2 cache, then the wasted space of 64-KB only corresponds to 3.1% of the L2 cache size.

In a multicore architecture that relies on a shared last level cache, the storage overhead of an inclusion property is more pronounced. For example, in the AMD Barcelona quad-core processor, each core has a private 512-KB L2 cache (for a total over four cores of 2-MB), while the shared-by-all-cores L3 cache size is 2-MB. If the inclusion property is enforced, 2-MB of cache space, equivalent to 100% of the L3 cache size, will be wasted!

Enforcing inclusion is typically achieved through a protocol between two caches. For example, when an L2 cache block is evicted, enforcing the inclusion property requires the L2 cache to also remove the copy from the L1 cache if the L1 cache has it.

A related concept to inclusion is what I refer to as **value inclusion**. When inclusion is maintained, and a block is cached in both the upper level and lower level caches, can the block's data values differ in both caches? If value inclusion is maintained, such a difference is not permitted. Otherwise, the difference is allowed. A difference in data value can happen if the upper level cache uses a write back policy, and the block is modified by the processor. The modification is not seen yet by the lower level cache. In a multiprocessor system, this creates a problem because the lower level cache does not know whether the block in the upper level cache is dirty or not. One way to avoid this situation is to use a write-

through policy at the upper level cache, so that value inclusion is maintained, i.e. the lower level cache always has the latest values of all blocks currently cached in the upper level cache. Another way to avoid this situation is not to maintain value inclusion, but on a write to a block in the upper level cache, the lower level cache is informed. The lower level cache can keep a bit or a special state that indicates the block is dirty in the upper level cache. When necessary, the lower level cache can request the upper level cache to write back the dirty block.

6.2.5 Split Cache Organization

The L1 cache is typically organized as a split cache, in which the cache is separated into the instruction cache and the data cache. One major reason for this separation is to reduce the requirements for ports. Each additional port in a cache adds an extra decoder, wires for word lines, wires for bit lines, sense amplifiers, and multiplexers. Hence, a two-ported cache can have a much larger structure than a single-ported cache.

The required number of ports for a cache depends on the level of the cache. For a L1 cache, in each cycle, the cache is accessed to provide instructions to the processor. In the same cycle, the data cache may be read or written by multiple instructions (especially in a superscalar core). Hence, if the L1 cache is not split, it needs to have at least three ports, which is complex and is likely slow. By separating the instruction cache from data cache, each of them can have fewer ports and consequently lower access time.

Lower level caches such as the L2 cache and L3 cache, on the other hand, are accessed infrequently (the L2 cache is only accessed on L1 cache misses), hence it is usually sufficient to have them single ported.

6.2.6 Translation Lookaside Buffer and Cache Addressing

An address presented to the cache that is used for indexing and tag comparison can be **virtual address** (addresses the program sees) or **physical address** (addresses of the physical memory). Let us first consider that physical address is used for indexing a cache and for the tags that are stored in the tag array of the cache.

The Operating System (OS) manages the translation of virtual address to physical address at the granularity of pages (typically 4 KBytes, but can be larger). The translation of a virtual page address to the physical page address is maintained in a software data structure called the **page table**. Since processes in the system have their own views of the virtual address spaces, each process has its own page table.

The page table is a structure stored in the main memory, so accessing it requires a high latency. Hence, requiring page table access for each memory access (a load or store instruction) in order to get the physical address to supply to the cache will cause the cache access time to be very high. Hence, most processors employ a **Translation Lookaside Buffer** (TLB), which is a cache that stores most recently used page table entries.

Figure 6.7 illustrates three options in cache addressing. The first option is to use **physical addressing**, that is, to index the cache using the index bits of the physical address, and store tag bits of the physical address in the cache's tag array. The drawback of this approach is that the TLB access time is directly added to the cache access time. A TLB is typically designed to be small so that it can be accessed in a few processor cycles. Still, adding a few cycles to each L1 cache access can significantly degrade performance.

Figure 6.7: Various cache addressing options.

The second option is to use **virtual addressing**, that is, to index the cache using the index bits of the virtual address, and store tag bits of the virtual address in the cache's tag array. Using virtual addressing, the TLB and the L1 cache can be accessed in parallel, so the latency of accessing the TLB is completely hidden.

However, the drawback of this approach is that since each process has its own page table, the mapping between the same virtual page address (of different processes) to physical page address is different in different processes. Thus, a virtually-addressed cache cannot be shared by multiple processors, either simultaneously in a multicore or multithreading manner, or in a time-shared manner. For example, upon a context switch, a different process is loaded into the processor, and the cache content is invalid for that process. Hence, the cache must be flushed on a context switch. The TLB also needs to be flushed if its entries are not tagged with the **address space identifiers** which identify different processes. The cost of L1 cache flushing, which includes the latency to invalidate the entire cache and the loss of data, may not be acceptable if context switches are frequent.

The third option is to use a mixed addressing referred to as virtual indexing and physical tagging. The key observation in this approach is that the page offset bits (bits used to index a particular byte or word location in a page) are the same in a virtual and physical address, so these bits can be used validly prior to accessing the TLB. If we ensure that the cache index bits are a subset of these bits, then the cache index is the same regardless of whether we use the virtual or physical address. For example, for a 4KB page, we need the lowest $log_2(4096) = 12$ bits as the page offset. For a 32-byte cache block size, we use the lowest $log_2(32) = 5$ bits as the block offset. Hence, we can use up to $12 - 5 = 7$ bits for cache indexing, which means that we can have $2^7 = 128$ sets in the cache. For a direct mapped cache, we can support an L1 cache size of $128 \times 1 \times 32 = 4096$ bytes. For a four-way associative cache, we can support an L1 cache size of $128 \times 4 \times 32$ = 16KB. More generally, the relationship between the maximum cache size that can be used is:

$$\begin{aligned} MaxCacheSize &= \frac{PageSize}{CacheBlockSize} \times Associativity \times CacheBlockSize \\ &= PageSize \times Associativity \end{aligned} \tag{6.2}$$

With virtual indexing and physical tagging, the cache can be accessed immediately and in parallel with TLB access (Figure 6.7). When the TLB access is complete, the physical address tag can be compared with the tag stored in the cache to determine whether we have a cache hit or a cache miss. The drawback of this addressing is that the L1 cache size is limited. However, this is not a serious drawback because the L1 cache size is already limited for performance purpose. In addition, the L1 cache size can be increased beyond the maximum size if we

are willing to increase the cache associativity. Therefore, many cache implementations choose this mixed addressing approach.

6.2.7 Non-Blocking Cache

A high performance processor may generate memory accesses even as an earlier memory access already suffers from a cache miss. In early cache designs, these memory accesses will stall until the access that misses in the cache is satisfied. This can obviously hurt performance. Modern cache architectures overcome this limitation by introducing a structure referred to as **miss status handling registers** (MSHRs). When a cache access suffers a cache miss, an MSHR is allocated for it to keep track of the status of the miss. This frees up the cache to service new accesses. Such a cache is referred to as a **non-blocking** or **lockup-free cache**. After an outstanding miss is satisfied, the MSHR that it corresponds to is deallocated.

The role of the MSHR depends on what type of access is kept track there. For a store instruction that misses in a write-back cache, the new value of the byte or word to be stored must be held in the MSHR until the data block is fetched from the lower level memory hierarchy. When the block arrives, the new value from the store must be merged into the block. For a load instruction that misses in the cache, the MSHR must link up with the processor register that is the destination of the load instruction. When the block arrives, the MSHR extracts the corresponding byte or word and passes that to the register for which the load instruction stalls.

The number of cache misses that can be outstanding at any given time is limited by the number of MSHRs. Hence, the number of MSHRs determines the degree of memory-level parallelism. The number of MSHRs must be designed by taking into account the number of instructions that the processor can execute in a cycle, the latency and bandwidth to access the lower level memory hierarchy, and the configuration of the upper level memory hierarchy that provides access filtering.

Note that non-blocking caches are required for implementing certain architecture features, among them are software prefetching (to be discussed in Section 6.4) and relaxed memory consistency models (to be discussed in Chapter 10). This is because without memory-level parallelism provided by a non-blocking cache, they cannot bring the performance benefit that they are designed for.

6.3 Cache Performance

The performance of the processor due the cache hierarchy depends on the number of accesses to the cache that find the blocks in the cache (**cache hits**) versus those which do not find the blocks in the cache (**cache misses**). There are various basic types of cache misses, known as the 3 Cs of cache misses:

1. **Compulsory misses** are misses that are required to bring blocks into the cache for the first time. They are also referred to as **cold misses** because they occur when the cache is cold (empty).

2. **Conflict misses** are misses that occur in the cache due to limited cache associativities.

3. **Capacity misses** are misses that occur in the cache due to limited cache size.

Distinguishing compulsory misses from other types of misses is easy: the first miss to a memory block is a compulsory miss, and subsequent misses are not. However, it is not always easy to distinguish between conflict and capacity misses. Typically, conflict misses are measured by subtracting the number of misses of a limited associativity cache by the number of misses of a fully-associative cache of the same size and cache block size [1]. The remaining misses after subtracting the compulsory misses and conflict misses can be categorized as capacity misses.

There are other types of cache misses that are mentioned less frequently in literature:

1. **Coherence misses** are misses that occur due to the need to keep the cache coherent in shared memory multiprocessor systems. This will be discussed in more details in later chapters.

2. **System-related misses** are misses that occur due to system activities such as system calls, interrupts, and context switches. When a process is suspended due to an interrupt, system call, or context switch, its cache state is perturbed by an interfering entity (another thread/process or the OS). When

[1] However, this approach sometimes fails. It is well known that in certain uncommon pathological cases, the number of cache misses on a fully-associative cache can be higher than in a set associative or a direct mapped cache. Using this approach will yield a negative number of conflict cache misses!

the process resumes execution, it will suffer from new cache misses to restore cache state that has been perturbed. These misses are system-related misses.

Coherence misses matter a lot in a multiprocessor system. They are one of the sources of bottlenecks in scalability of parallel programs. System-related misses also matter overall, and we need to take them into account as they tend to increase when the cache size increases, precisely when the number of conflict and capacity misses decreases.

Figure 6.8 shows the breakdown of the L2 cache misses for twelve SPEC2006 benchmarks when each benchmark is co-scheduled with one other interfering benchmark for various cache sizes. The figure is taken from the study by Liu et al. [Liu et al., 2008]. Since there are eleven other benchmarks, there are eleven possible co-schedules. The number of L2 cache misses for each benchmark is the average over eleven co-schedules the benchmark can have, normalized to the case in which 512-KB cache is used. The context switch misses are divided into two types: *replaced* misses that occur due to misses to blocks that were replaced during the cache perturbation caused by a context switch, and *reordered* misses that occur due to misses to blocks that were not replaced but were reordered in the recency positions in the cache during the cache perturbation. A time slice of 5ms is used for both benchmarks in a co-schedule. The Linux kernel version 2.6 assigns time quanta between 5 ms to 100 ms based on the process/thread priority level, so 5ms corresponds to the lowest priority level.

The figure shows that as the cache size increases, the number of "natural misses" (the sum of cold, capacity, and conflict misses) decreases, the total number of context switch misses may decline (in namd, perlbench), increase (in mcf, soplex), or increase then decline (in astar, bzip2, gcc, hmmer, omnetpp and sjeng). The reason for this behavior is because when the cache is small, there are not many cache blocks that can be displaced by a cache perturbation that occurs in a context switch, so the number of context switch misses is low. As the cache becomes larger, a cache perturbation may displace more cache blocks, causing more context switch misses when the process resumes execution. However, when the cache becomes large enough to hold the total working sets of both benchmarks, cache perturbation only displaces very few blocks, so the number of context switch misses declines. Rather than being displaced, most blocks are now reordered, but reordered blocks do not incur context switch misses if there are no other misses to cause them to be displaced when the thread resumes execution. Hence, the abso-

Figure 6.8: Breakdown of the types of L2 cache misses suffered by SPEC2006 applications with various cache sizes.

lute number of reordered misses also declines as the cache becomes large enough. Overall, system-related misses should not be ignored when context switches occur frequently and when the cache is large but is still smaller than the combined working set size of processes that time-share a processor.

Cache parameters affect the different types of misses. Intuitively, increasing the cache size reduces the number of capacity misses, while increasing cache associativity reduces the number of conflict misses. However, capacity misses and conflict misses are sometimes intermingled. For example, increasing the cache size while keeping cache associativity constant (hence increasing the number of sets) changes the way blocks in memory map to sets in the cache. Such a change in mapping often influences the number of conflict misses, either increasing or decreasing it. Increasing the cache size while keeping the number of sets constant (hence increasing the associativity) does not change how blocks in memory map to cache sets, but since associativity is increased, both conflict and capacity misses may also be reduced.

Although compulsory misses are "compulsory", they are measured by the number of times new blocks are fetched into the cache, hence they are affected by the cache block size. Larger cache block sizes reduce the number of compulsory misses when the program has good spatial locality. The number of capacity and conflict misses may also decrease because more bytes are fetched upon each

miss, thus fewer misses are required to load the same number of bytes. However, when there is little or no spatial locality, the number of capacity misses may increase. This is because increasing block sizes while keeping the cache size constant reduces the number of blocks that can be cached.

To summarize, Table 6.1 shows the impact of various cache parameters on various types of cache misses.

Table 6.1: Impact of various cache parameters on cache misses.

Parameters	Compulsory	Conflict	Capacity
Larger cache size	unchanged	unchanged	reduced
Larger block size	reduced	reduced/ increased	reduced/ increased
Larger associativity	unchanged	reduced	unchanged

6.3.1 The Power Law of Cache Misses

Researchers have long made an interesting empirical observation with regard to how a sequential application's number of cache capacity misses is affected by the cache size. In a single core/processor system, the capacity miss rate of an application tends to follow a certain trend known as the *power law* [Hartstein et al., 2008]. Mathematically, the power law states that if m_0 is the miss rate of a workload for a baseline cache size C_0, the miss rate (m) for a new cache size C can be expressed as:

$$m = m_0 \cdot \left(\frac{C}{C_0}\right)^{-\alpha} \tag{6.3}$$

where α is a measure of how sensitive the workload is to changes in cache size. Hartstein et al. in [Hartstein et al., 2008] validated the power law on a range of real-world benchmarks, and found that α ranges from 0.3 to 0.7, with an average of 0.5. When $\alpha = 0.5$, Equation 6.3 can be rewritten as:

$$\frac{m}{m_0} = \sqrt{\frac{C_0}{C}} \tag{6.4}$$

Let us suppose that $C = 4 \times C_0$, then $m = \frac{m_0}{2}$, which means that for an average workload, Equation 6.3 states that in order to decrease the capacity miss rate by a factor of two, the cache size must be increased by a factor of four. Hence,

there is a diminishing number of cache misses that can be eliminated for each constant increase in the cache capacity.

How good are applications in reality tracking the power law? Figure 6.9 plots the cache miss rate of each application we evaluate normalized to the smallest cache size as a function of cache size for a single level of cache. Note that both axes are shown in logarithmic scale; the miss rate curve will follow a straight line if it obeys the power law, because $log\ m = log\ m_0 + \alpha\ log\ C_0 - \alpha\ log\ C$. In the figure, the base cache size is 8KB, and each subsequent data point doubles the size until it reaches 2MB.

Figure 6.9: Normalized cache miss rate as a function of cache size.

The figure shows that these applications tend to follow the power law of cache miss rate quite closely. Also, note the four bold lines in the figure correspond to the power law fit for all commercial applications, for all SPEC 2006 benchmarks, for the commercial application with the smallest α (OLTP-2), and for the commercial application with the largest α (OLTP-4). The larger the α value, the steeper the negative slope of the line. The curve-fitted average α for the commercial workloads is 0.48, close to 0.5 in Hartstein et al.'s study [Hartstein et al., 2008], while the minimum and maximum α for the individual commercial applications are 0.36 and 0.62, respectively. The smallest α (SPEC 2006) has a value of 0.25.

While the power law is a highly useful tool for coming up with a back-of-envelope projection of cache performance, there are several aspects that need to be taken into account before using it. First, note that the power law has a finite range of cache sizes for which it applies. This is because at some large enough cache size, the number of capacity misses will drop to zero, whereas the power

law, if extrapolated beyond this range, will predict a negative number of cache capacity misses. It is important to apply the power law only in the range before the number of capacity misses flattens. In addition, there are applications which have their miss rate figures exhibiting shapes that are stepwise linear. Typically, this is because the application has a discrete number of loops, where at a certain cache size, the entire working set of a loop fits, resulting in a large drop in miss rate. Then, there is no improvement in miss rate until the cache size is large enough for another loop's working set fits. For this reason, individual SPEC2006 applications exhibit more stepwise linear miss rate figure compared to commercial applications which exhibit a much more linear figure.

6.3.2 Stack Distance Profile

The power law captures a rough approximation of an application's miss rate as affected by the cache size. The power law enables us to represent the average behavior of a group of applications, and in some cases, individual applications. However, there are situations in which the power law does not fit the behavior of individual applications. To come up with a better representation of the behavior of an application's miss rate as affected by the cache size, researchers have demonstrated a useful profiling information referred to as the **stack distance profile**.

A stack distance profile captures the temporal reuse behavior of an application in a fully- or set-associative cache [Mattson et al., 1970]. To collect the stack distance profile of an application on an A-way associative cache with LRU replacement algorithm, $A + 1$ counters are maintained: $C_1, C_2, \ldots, C_A, C_{>A}$. On each cache access, one of the counters is incremented. If it is a cache access to a line in the ith position in the LRU stack of the set, C_i is incremented. Note that our first line in the stack is the most recently used line in the set, and the last line in the stack is the least recently used line in the set. If it is a cache miss, the line is not found in the LRU stack, resulting in incrementing the miss counter $C_{>A}$. A stack distance profile can be estimated by the compiler [Cascaval et al., 2000], by simulation, or directly on hardware with an appropriate hardware support [Suh et al., 2002].

Figure 6.10 shows an example of a stack distance profile. Applications with regular temporal reuse behavior usually access more-recently-used data more frequently than less-recently-used data. Therefore, typically, the stack distance profile shows decreasing values as we go to the right, as shown in Figure 6.10. Using the stack distance profile, the number of cache misses for a smaller cache can be

Figure 6.10: Illustration of a stack distance profile

easily computed. For example, for a smaller cache that has A' associativity, where $A' < A$, the new number of misses can be computed as:

$$miss = C_{>A} + \sum_{i=A'+1}^{A} C_i \tag{6.5}$$

Hence, for a cache that employs the LRU replacement policy, the miss rate as a function of the cache capacity can be directly derived from the stack distance profile of an application.

6.3.3 Cache Performance Metrics

In many sequential program execution cases, the number of cache misses correlates almost linearly with program throughput (instruction per cycle or IPC). However, how much each cache miss contributes to the reduction in IPC depends on the characteristics of the program, such as how much of computation can be overlapped with cache misses (due to **Instruction-Level Parallelism** or ILP) and how much of a cache miss can be overlapped with other cache misses (due to **Memory-Level Parallelism** or MLP). If one ignores the impact of ILP and MLP on performance, then one of meaningful metrics to evaluate cache performance is the **average access time** (AAT), which is defined as the average time a memory access takes, and is computed by taking into account hit rates at various cache levels. For example, for a system with two levels of caches, AAT can be computed as follows:

$$AAT = L1HitTime + L1MissRate \cdot L2AccessTime + L1MissRate \cdot L2MissRate \cdot L2MissTime \quad (6.6)$$

A more meaningful metrics (but more difficult to measure) that takes into account the impact of ILP and MLP is CPI (cycle per instruction) and how it is related with miss rates. Suppose a program has n non-overlapping types of instruction: type 1, 2, ... , n. Each type of instruction has a cycle per instruction ratio (cpi_i), and the fraction of instruction of type i is $frac_i$. Assume that the fraction or CPI of each type of instruction does not affect the CPI of other types of instructions. Then, the cycle per instruction (cpi) of the program can be expressed as:

$$cpi = frac_1 \cdot cpi_1 + frac_2 \cdot cpi_2 + ... + frac_n \cdot cpi_n \quad (6.7)$$

Using the formula, we can separate instructions that result in cache misses and instructions that do not result in cache misses. Let us denote $h2$ as fraction of instructions that access the L2 data cache, and $t2$ be the CPI of that type of instruction. Let hm denotes fraction of instructions that miss in L2 data cache (access into memory), and tm be the CPI of that type of instruction. $t2$ can be thought of as the processing time/latency of instructions that access the L2 cache but not beyond it, and tm as the latency of instructions that access the memory. Furthermore, let us express the CPI of all other instructions as CPI_0. We can express the CPI of the program as the weighted sum of its components:

$$CPI = (1 - h2 - hm) \cdot CPI_0 + h2 \cdot t2 + hm \cdot tm \quad (6.8)$$

$h2$ and hm can be computed from the number of misses in the L1 cache ($L1miss$) and the L2 cache ($L2miss$), which are easily obtained from hardware counters that are available in many processors. Given the total number of instructions executed, $h2$ and hm can be computed as:

$$h2 = \frac{L1miss - L2miss}{NumInstructions} \quad (6.9)$$

$$hm = \frac{L2miss}{NumInstructions} \quad (6.10)$$

On the other hand, CPI_0, $t2$, and tm are not easily computed because they already embed the impact of ILP and MLP. On a processor simulation, they can be

readily obtained by embedding statistics collection as the simulation progresses. On a real machine, however, most hardware counters may not have the ability to collect them. In such a case, their values can be inferred statistically using the least square estimation method. To illustrate the process, suppose that CPI_0 is known, and all we need to do is to estimate $t2$ and tm. Suppose that we have the ability to run a program n times with different input sizes that change the hit rates and miss rates of various cache levels, but preserve the overall degree of ILP and MLP of the program. Since there are n runs, we obtain n data points which include a set of cpi, $h2$, and hm values:

$$cpi^1 = (1 - h2^1 - hm^1)cpi_0 + h2^1 \cdot t2 + hm^1 \cdot tm \tag{6.11}$$

$$cpi^2 = (1 - h2^2 - hm^2)cpi_0 + h2^2 \cdot t2 + hm^2 \cdot tm \tag{6.12}$$

$$\cdots$$

$$cpi^n = (1 - h2^n - hm^n)cpi_0 + h2^n \cdot t2 + hm^n \cdot tm \tag{6.13}$$

They can be expressed conveniently as a matrix multiplication $Y = A \times X$, where Y is a $n \times 1$ matrix, A is an $n \times 2$ matrix, and X is a 2×1 matrix. The values of the matrices are:

$$Y = \begin{pmatrix} cpi^1 - (1 - h2^1 - hm^1)cpi_0 \\ cpi^2 - (1 - h2^2 - hm^2)cpi_0 \\ \cdots \\ cpi^n - (1 - h2^n - hm^n)cpi_0 \end{pmatrix} A = \begin{pmatrix} h2^1 & hm^1 \\ h2^2 & hm^2 \\ \cdots & \cdots \\ h2^n & hm^n \end{pmatrix} X = \begin{pmatrix} t2 \\ tm \end{pmatrix}$$

To make the system solvable by linear equation set algorithm such as Gaussian elimination, both sides are multiplied by the transpose of matrix A ($A^T_{2\times n}$):

$$(A^T \cdot Y)_{2\times 1} = (A^T \cdot A)_{2\times 2} \cdot X_{2\times 1} \tag{6.14}$$

Since now matrix $(A^T \cdot A)$ has 2 rows and 2 columns, the equation 6.14 can be directly solved using Gaussian elimination. The solution to the equation is the least square error approximation to $t2$ and tm.

We have discussed how various cache performance metrics can be collected and how they can be related to CPI or IPC for sequential execution. For sequential execution, CPI multiplied by the number of instructions of the program directly yields the execution time of the program. For parallel programs, however, the relationship between cache performance metrics such as miss rates and execution

time is not as straightforward due to the relative timing of execution progress of different threads. For example, a thread that arrives at a barrier early will either spin in a loop or block waiting for all other threads to arrive at the barrier. While spinning, it executes a series of reads and compares on a memory location in the cache. Such spinning code executes a lot of instructions and may produce a lot of cache accesses and cache hits with very few cache misses. This leads to a high IPC and low miss rates. Unfortunately, none of the high IPCs and low miss rates of spinning contributes to shorter execution time since it is due to useless work. Another example is the number of coherence misses which highly depends on the relative timing of execution of different threads. To summarize, regular cache performance metrics cannot be blindly used for diagnosing parallel program performance.

6.4 Prefetching

Modern cache designs often employ **prefetching**, a technique which attempts to bring data into the cache before the program accesses it. Prefetching reduces the time a program waits for data to be fetched, so it can speed up program execution. However, it comes at the expense of extra bandwidth usage since it employs prediction of what data may be needed in the future by the processor. Such prediction can never be perfect, hence some blocks that are prefetched may never be used by the processor.

Many processors provide a special instruction for specifying an address to prefetch. While such prefetching instruction is supported in hardware, it is up to the software (i.e. the compiler) to use it, and where to insert it. Therefore, using prefetch instructions for performing prefetching is referred to as **software prefetching**. The alternative to software prefetching is to make hardware structures that dynamically observe the program's behavior and generate prefetch requests, transparently to software. Such an approach is referred to as **hardware prefetching**. More recently, the boundary between hardware and software prefetching has become less firm, for example prefetch instructions may be inserted dynamically by a run-time optimizer, whereas some hardware prefetching techniques may be implemented in software.

There are three metrics that are traditionally used to characterize the effectiveness of a prefetching technique: coverage, accuracy, and timeliness. **Coverage** is defined as the fraction of original cache misses that are prefetched, and therefore,

the cache misses become cache hits or partial cache misses. **Accuracy** is defined as the fraction of prefetches that are useful, i.e. they result in cache hits. And finally, **timeliness** measures how early the prefetches arrive, which determines whether the full cache miss latency is hidden, or only a part of the miss latency is hidden. An ideal prefetching technique should exhibit high coverage so that it eliminates most of the cache misses, high accuracy so that it does not increase memory bandwidth consumption, and timeliness so that most of the prefetches hide the full cache miss latency. Achieving this ideal is a challenge. By aggressively issuing many prefetches, a prefetching technique may achieve a high coverage but low accuracy. By conservatively issuing only prefetches that are highly predictable, a prefetching technique may achieve a high accuracy but a low coverage. Finally, timeliness is also important. If a prefetch is initiated too early, it may pollute the cache, and may be replaced from the cache or prefetch buffer before it is used by the processor. If it is initiated too late, it may not hide the full cache miss latency.

The three metrics (coverage, accuracy, and timeliness) are complete metrics if prefetched blocks are stored in a special storage (called **prefetch buffers**). If on the other hand they are placed directly in the cache, they can cause an extra problem, which requires different metrics to be collected. For example, if a prefetch causes a cache block to be evicted and there is a miss that occurs to that block due to the eviction, then the prefetch is potentially harmful. If the prefetch itself results in a future cache hit, then its harmfulness is offset. Therefore, collecting the statistics on the number of useful vs. useless but harmless vs. useless and harmful prefetching can give good insights into the effectiveness of a prefetching scheme.

In addition to evaluating prefetching schemes based on their performance metrics, they must also be evaluated based on other important measures, such as the cost and complexity of a hardware implementation and whether they require the code to be recompiled.

We will now focus our discussion on basic hardware prefetching techniques since they are ones that are directly related to the cache design. Based on the type of data access patterns that can be handled, **sequential prefetching** detects and prefetches for accesses to contiguous locations. **Stride prefetching** detects and prefetches accesses that are s-cache block apart between consecutive accesses, where s is the amount of the stride. Therefore, an s-strided accesses would produce address trace of $A, A+s, A+2s, \ldots$. Note that sequential prefetching is a

stride prefetching where $s \leq 1$. With large cache block sizes in current processor architectures, sequential prefetching is often sufficient in detecting most strided access patterns.

There are different places where prefetching can be initiated and the destination where the prefetched data can be placed. Prefetching can be initiated in the L1 cache level, L2 cache level, L3 cache level, the memory controller, or even in the memory chips to preload DRAM row buffers. Prefetched data is usually stored at the level at which the prefetch is initiated, such as in the prefetch-initiating cache, or in a separate prefetch buffer to avoid polluting the cache with prefetched data.

6.4.1 Stride and Sequential Prefetching

Early stride and sequential prefetching studies include the Reference Prediction Table by Chen and Baer [Chen and L.Baer, 1992], and also stream buffers by Jouppi [Jouppi, 1990]. This section will describe stream buffers scheme, which is attractive due to its simplicity. A stream buffer allows prefetching sequential accesses. A stream buffer is a first in first out (FIFO) buffer, where each entry contains a cache-block worth of data, an address (or tag) of the block, and an "available" bit. To prefetch for multiple streams in parallel, more than one stream buffers can be used, in which each buffer prefetches from one stream.

On a cache access, the cache is checked for a match. The head entries of stream buffers are also checked in parallel to that (or alternatively, after the cache access is found to be suffer a cache miss). If the requested block is found in the cache, no action on the stream buffers is performed. If the block is not found in the cache, but found at the head of a stream buffer, the block is moved into the cache. The next entry in the stream buffer now becomes the head entry, and the freed up slot triggers a prefetching request to an address sequential to address in the last entry of the buffer. If the block is not found in both the cache and all the stream buffers, a new stream buffer is allocated. The block's successors are prefetched to fill the stream buffers.

Figure 6.11 illustrates the stream buffer operation with four stream buffers. Suppose that the addresses from access streams are $A, B, A+2, B+1, A+4, B+2, \ldots$ and that the cache and stream buffers do not contain any of the blocks initially. The access to A results in a cache miss, which results in allocating a stream buffer for A's successors, which are $A+1, A+2, A+3, A+4$. The figure shows all A's successors have been prefetched into the stream buffer. Similarly, the access to B results in a cache miss, and a stream buffer is allocated for $B's$

Figure 6.11: Illustration for stream buffer operation.

successors $B+1, B+2, B+3, B+4$. The figure shows that currently, only blocks $B+1$ and $B+2$ have been stored in the stream buffer, while $B+3$ and $B+4$ are still being prefetched. Next, for an access to $A+3$, the cache and the head entries of the stream buffers are checked for a match. Since there is no match, a new stream buffer is allocated for $A+3$'s successors $A+4, A+5, A+6, A+7$. However, an access to $B+1$ will find a match in the stream buffer, and $B+1$ will be moved into the L1 cache, and the freed entry is used to prefetch the successor of the tail entry.

Note that now we have two stream buffers allocated for A's successors and $A+3$'s successors, and some entries overlap in two stream buffers ($A+4$). This is caused by the inability to identify a stride of 2 for A's stream. In fact, stream buffers cannot handle any non-sequential accesses efficiently, including non-unit strides ($s > 1$) or even a negative sequential accesses ($s = -1$). For non-unit stride accesses, each access could result in a stream buffer miss because the address is only compared to the head entries of the buffers. Not only does this produce overlapping entries, a single stream may also occupy several stream buffers. To overcome that, Palacharla and Kessler [Palacharla and Kessler, 1994]

suggested two techniques to enhance the stream buffers: allocation filters and a non-unit stride detection mechanism. The filter waits until there are two consecutive misses for the same stream before allocating a stream buffer, making sure that a stream buffer is only allocated for strided accesses.

Overall, prefetching for sequential and strided accesses can be achieved with relatively simple hardware. Several recent machines, such as Intel Pentium 4 and IBM Power4 architectures implement a hardware prefetcher that is similar to stream buffers at the L2 cache level [Hinton et al., 2001, IBM, 2002]. For example, the L2 cache in the Power4 can detect up to eight sequential streams. The L2 cache block is 128 bytes, and therefore, a sequential stream detector can catch most strided accesses. On a cache miss to a block in the L2 cache, the fifth successor block is prefetched. This is similar to a 4-entry stream buffers. The L3 cache block is four times larger than the L2 (512 bytes), and therefore it only prefetches the next successive block on an L3 miss. Since the prefetching is based on physical addresses, prefetching of a stream is terminated when a page boundary is encountered. However, for applications with long sequential accesses, continuity in the prefetching can be supported if large pages are used.

6.4.2 Prefetching in Multiprocessor Systems

In a multiprocessor system, ensuring that prefetching does not cause unintended negative performance impact is harder. A block that is prefetched by a processor too early may be useless since it may be stolen by another processor before the processor has a chance to access the block. In the mean time, the prefetched block has already evicted another block that may be more useful. Therefore, in multiprocessor systems, prefetching must ensure that a block is prefetched not only sufficiently early to hide the latency of future misses, but also not too early that it ends up stolen by other processors. In a single processor environment, prefetching that is too early is risky if the cache capacity is limited because the prefetched block may evict a more useful block (this does not happen if the cache is large enough to keep both blocks). In multiprocessor systems, even when the cache is really large, prefetching too early can still hurt performance by unnecessarily stealing a cache block from another processor, which may have to steal it back.

■ *Did you know?*

In embedded systems which are used for processing tasks with hard real-time constraints, predictability in the worst-case execution time is paramount as missing the deadline of a task processing will result in catastrophic consequences. Such systems often strip out components that contribute to execution time variation that is hard to predict, such as virtual memory, branch prediction, and also caches. Since the number of cache misses and the additional execution time due to cache misses are hard to bound, these systems replace them with software-managed storage called scratchpad memory. *Various policies that are managed by hardware in caches are managed by software, including placement policy, replacement policy, allocation policy, and write back policy. Transfer to and from the scratchpad memory often involves an on-chip DMA engine. Since the software knows exactly which data access will be to the scratchpad memory as opposed to the main memory, it can bound the worst case execution time much more tightly. However, the complexity in managing the scratchpad memory is quite high, and increases when the applications grow in complexity. Therefore, some embedded systems include both caches and scratchpad memory.*

6.5 Exercise Problems

1. (25 points) **Cache Performance**. Suppose you have a load-store in-order processor running at 2 GHz and has a CPI of 0.7 (excluding memory accesses). The average workload on the processor consists of 20% loads and 5% stores. The processor has L1 data cache and L1 instruction cache, L2 cache that holds both data and instructions, and the main memory. The organization of the caches are shown in the following table:

	L1-Instruction	L1-Data	L2
Size	32KB	32KB	1MB
Block Size	32B	32B	64B
Associativity	2	2	8
Access time	1ns (hit)	1ns (hit)	9ns (hit), 90ns (miss)

The average workload suffers miss rates of 10% at the L1 caches and 20% at the L2 cache. The L2 miss rate is computed as the number of L2 cache misses divided by the number of L2 cache accesses.

Compute the following (ignoring write through or write back traffic):

(a) What is the average memory access time (in processor cycles) for instruction accesses?

(b) What is the average memory access time (in processor cycles) for data reads?

(c) What is the average memory access time (in processor cycles) for data writes?

(d) What is the overall CPI including memory accesses?

(e) What is the speedup when we change the CPU to a 4 GHz one, everything else (including CPI) unchanged?

2. (30 points) **Cache Performance**. Suppose you have a load-store in-order processor running at 2 GHz and has a CPI of 0.7 (excluding memory accesses). The average workload on the processor consists of 20% loads and 5% stores. The processor has L1 data cache and L1 instruction cache, L2 cache that holds both data and instructions, and the main memory. The organization of the caches are shown in the following table:

	L1-Instruction	L1-Data	L2
Size	32KB	32KB	2MB
Block Size	64B	64B	128B
Associativity	2	2	8
Access time	1ns (hit)	1ns (hit)	9ns (hit), 90ns (miss)
Write policy	Write-Through	Write-Through	Write-Back

The average workload suffers miss rates of 10% at the L1 caches and 20% at the L2 cache. The L2 miss rate is computed as the number of L2 cache misses divided by the number of L2 cache accesses. The bus between the L1 and L2 cache is 256-bit wide and runs at 1GHz. The bus between the L2 cache and main memory is 64-bit wide and runs at 500MHz. In the L2 cache, 50% of all blocks that are replaced are dirty (thus need to be written back).

Compute the following (taking into account data transfer between caches, between the L2 cache and memory, and transfer due to write through and write back activities):

(a) What is the average memory access time (in processor cycles) for instruction accesses?

(b) What is the average memory access time (in processor cycles) for data reads?

(c) What is the average memory access time (in processor cycles) for data writes?

(d) What is the overall CPI including memory accesses?

(e) Suppose that we enable the critical-word early-restart optimization at the L2 cache. What is the new overall CPI with this optimization?

(f) Suppose that we use Itanium-like optimization in which on each memory access, we check the L1 cache tag and L2 cache tag in parallel. The benefit of this scheme is we can start transferring data from L2 to L1 or from memory to L2 sooner. What is the new overall CPI with this optimization?

3. (25 points) **Cache Replacement Policy**. Suppose that you have a 4-way associative cache. In one of the sets of the cache, you see a stream of accesses to the following block addresses: A B C D E A G H F G G C D F G E B A. Determine whether each of the access would result in a cache hit, a

cache miss, or a cache miss and replacement. Show what block is replaced for each of the following replacement policies: LRU, OPT, FIFO, Pseudo-LRU, and Inverse MRU. Inverse MRU is an approximation to NMRU1 that keeps track of which line is the most recently used within each set (say, line 010 from an 8-way associative cache). On replacement, replace the line that is the inverse of the MRU (101 in the above example). Assume that initially you fill the cache starting from way 00, 01, 10, and then 11.

For each cell, use the following notation: "H" for hit, "M" for miss, "MR-x" for miss and replace block x.

Accessed Blk	LRU	Pseudo-LRU	FIFO	Inverse MRU	OPT
A					
B					
C					
D					
E					
A					
G					
H					
F					
G					
G					
C					
D					
F					
G					
E					
B					
A					

4. (15 points) **Cache Replacement and Indexing**.

   (a) Is it possible for a direct-mapped cache to achieve a lower miss rate than a fully-associative cache with LRU replacement policy? Provide a proof for your answer.

   (b) Suppose that you have a cache with n sets, and the block size is b. Now suppose you run a program that shows striding accesses with a

stride of s starting from address 0. That is, the addresses generated would be $0, s, 2s, 3s, 4s, \ldots$. If $n = 32$, $b = 8$ bytes, and $s = 32$ bytes, identify which sets of the cache would be more heavily utilized than others?

(c) Suppose that instead of using number of sets that is a power of two, we switch to a number of sets that is prime, i.e. $n = 31$. Can you identify which sets of the cache would be more heavily utilized than others? Explain your observation.

5. **Power Law**. How much bigger a cache size needs to be in order to reduce the miss rate of a workload by 50%, if the workload has a cache size sensitivity (α) value of 0.3, 0.5, 0.7, or 0.9?

6. **Power Law and Stack Distance Profile**. Suppose a workload with $\alpha = 0.5$ suffers a miss rate of 10% on an 8-way associative cache with size C_0. Calculate the stack distance counters $C_1, C_2, \ldots, C_8$ in terms of the fraction of accesses to various stack positions. Repeat the exercise for $\alpha = 0.3$ and $\alpha = 0.8$.

7. **Stack Distance Profile**. Suppose a program running on a processor that has a 8-way associative cache has the following fractions of cache accesses to different stack positions:

Stack position	App1's fraction of accesses	App2's fraction of accesses
1 (MRU)	60%	50%
2	10%	10%
3	5%	8%
4	3%	7%
5	2%	5%
6	0%	1%
7	8%	1%
8 (LRU)	7%	1%
9 and higher	5%	17%

(a) Draw the miss rate curve (the miss rate as a function of cache size/associativity) for both applications.

(b) Suppose that we employ a hardware prefetching mechanism that is able to accurately identify 100% of all cache miss block addresses. The prefetching is timely, in that it places blocks in the cache early

enough that three blocks will be accessed in each set before the prefetched block is accessed. What are the new cache miss rates after prefetching is applied? Is prefetching helpful or harmful to performance for App1 and App2?

8. **Non-Uniform Cache Architecture (NUCA).** Suppose a program running on a single processor has the following fractions of cache accesses to different stack positions:

Stack position	App1's fraction of accesses	App2's fraction of accesses
1	60%	50%
2	10%	10%
3	5%	8%
4	3%	7%
5	2%	5%
6	0%	1%
7	8%	1%
8	7%	1%
9 and higher	5%	17%

Suppose that the system implements multiple cache banks with various latencies. The closest cache bank has an access latency of 3 cycles and holds blocks that are the first two stack positions (most recently and next most recently used blocks). The next farther cache bank has an access latency of 6 cycles and holds blocks in the next two stack positions. The next farther cache bank has an access latency of 9 cycles and holds blocks in the next two stack positions. The farthest cache bank has an access latency of 12 cycles and holds blocks in the last two stack positions.

(a) What is the average cache access time for App1?

(b) What is the average cache access time for App2?

Chapter 7

Introduction to Shared Memory Multiprocessors

Contents

7.1 The Cache Coherence Problem 181
7.2 Memory Consistency Problem 184
7.3 Synchronization Problem 186
7.4 Exercise Problems 192

In Chapter 6, we have discussed memory hierarchy organization for a single processor system. In what way does the memory hierarchy design for a multiprocessor system differ from a single processor system? To answer such a question, it is necessary to ask what factors affect program correctness if a multiprocessor system is built from just interconnecting multiple single processors together.

The objective of this chapter is to answer the question of what hardware support is needed in order to guarantee the correctness of executing shared memory parallel programs on a multiprocessor system. This chapter highlights that there are three main types of support needed: a (1) *cache coherence protocol*, which ensures a coherent view of cached values as seen by multiple processors, a (2) *memory consistency model*, which ensures consistent view of the ordering of memory operations, and (3) hardware *synchronization* support, which enables a simple, correct, and efficient primitive on which programmers can build other synchronization primitives.

As we have mentioned in earlier chapters, we focus the discussion on multiprocessor systems that support the abstraction of shared memory. The abstraction

of shared memory does not come automatically; it requires special hardware support. This is in contrast with systems that do not provide the shared memory abstraction and rely on explicit message communication for processors to communicate with one another. Such systems only require a message passing library to ease sending and receiving messages.

There are several advantages of organizing a multiprocessor system as a shared memory system. First, programs that were created assuming a shared memory system, such as shared memory parallel programs, and multi-threaded programs created for a single processor system, will work automatically on a shared memory multiprocessor. They need to be significantly modified to run on systems that do not provide shared memory. Secondly, there is only one OS image running on the system, which simplifies maintenance and co-ordinated scheduling of threads. Thirdly, the total memory size of a shared memory multiprocessor is the multiplication of the number of nodes with the size of memory of each node. This makes shared memory multiprocessor systems attractive for applications with large memory footprint sizes. For example, an anecdotal evidence on the profile of applications that ran on the 128-processor shared memory Origin 2000 system indicated that an important fraction of applications were non-parallel applications that required a huge memory to perform well without thrashing. These programs would cause an excessive number of page faults on systems that do not provide shared memory. Fourthly, fine grain sharing is naturally supported in shared memory multiprocessors. Communicating a few bytes or a cache block takes a few hundred to a few thousand cycles, whereas in a system without shared memory, the overheads of message bundling, sending, receiving, and unbundling can be orders of magnitude higher. Therefore, algorithms in a program must be structured in such a way that a large collection of data items are sent together using few large messages rather than many small messages, in order to make message sending and receiving infrequent events. Finally, common data structures that are read by all threads do not need to be replicated in a shared memory system but they need to be replicated in systems without shared memory, incurring storage overheads.

Providing the shared memory abstraction requires hardware support that can be expensive. Such support is relatively cheap on small-scale systems such as multicore processors and small bus-based multiprocessors (with up to 8-16 processors). In fact, on multicore processors with cores sharing a cache, shared memory can be automatically provided. However, the cost of providing shared memory grows super-linearly as the number of processors increases, whereas the cost

of non shared-memory clusters can grow relatively linearly with the number of processors if the demand for high bandwidth and low latency is not too strong (allowing cheap interconnection to be used). Shared memory multiprocessors are prohibitively expensive to support for very large systems consisting of thousands of nodes. For example, the SGI Altix 3300 [Silicon Graphics, Inc., 2004], perhaps the largest shared memory system today, can only support 512 processors. To provide larger systems, a cluster of shared memory nodes can be interconnected together, with the shared memory abstraction provided within nodes but not across nodes.

7.1 The Cache Coherence Problem

Let us assume that there are multiple chips, each with one processor core and a cache, which we want to use to form a shared memory multiprocessor system. To achieve a shared memory multiprocessor, is it only a matter of interconnecting such chips with some kind of interconnect network such as a bus (Figure 7.1)? Can the abstraction of a *single* shared memory be automatically achieved? The answer is no for several reasons that will be discussed further. It is important to understand what problems occur, so that we can have an intuition into what hardware support is required to provide the shared memory abstraction.

Figure 7.1: A simple bus-based multiprocessor system with four processors.

For example, consider the following code (Code 7.1), which has two threads executing on two processors accumulating the values of $a[0]$ and $a[1]$ to the vari-

able sum.

Code 7.1 A simple code to accumulate two values to a sum.

```
sum = 0;
#pragma omp parallel for
for (i=0; i<2; i++) {
  #pragma omp critical {
    sum = sum + a[i];
  }
}
print sum;
```

Let us assume that $a[0] = 3$ while $a[1] = 7$. The correct outcome of the computation is that the sum reflects the addition of $a[0]$ and $a[1]$ at the end of the computation, which corresponds to a value of 10. In a system without a cache, thread 0 reads the initial value of sum from memory, accumulates a value of 3 to it, and stores it back to memory. Thread 1 reads a value of sum (which is now 3) from memory, accumulates 7 to it, and stores back a value of 10 to memory. When thread 0 reads sum from memory, it will read 10 and print it, which is correct.

Now imagine that each processor has a cache (Figure 7.1), more specifically, a write-back cache. Suppose that initially, the value of sum in memory is 0. The sequence of operations that occurs is shown in Table 7.1. Thread 0 starts out by reading (with a load instruction) from the memory location where the variable sum is located, into a register. This results in the memory block that contains sum to be cached by processor 0. Then, thread 0 performs an addition instruction between sum and $a[0]$. The result of the addition, currently still in a register, is then written to the memory location of sum using a store instruction. Since the block containing sum is cached, the cache block is modified and the dirty bit is set. In the mean time, the main memory still contains the stale copy of the block, in which the value of the sum is 0. When thread 1 performs a read to sum from the main memory, it will see the value of sum as 0. It adds $a[1]$ and sum, and stores the result to the cache. The value of sum in its cached copy is 7. Finally, when thread 0 prints the value of sum, it reads from the memory location of sum, and it will read it from the cache because it has a valid copy in its cache (a cache hit). It will print "3", which is incorrect.

At this point, readers may wonder whether the problem is resolved when write-through caches are employed instead of write-back caches. The answer is no. In the sequence shown in Table 7.1, the problem is partially mitigated, because when

Table 7.1: Illustration of the cache coherence problem.

Action	P0's Cache	P1's Cache	Memory
Initially	–	–	$sum = 0$
P0 reads sum	$sum = 0$	–	$sum = 0$
P0 adds sum with a[0]	$sum = 3$	–	$sum = 0$
P1 reads sum	$sum = 3$	$sum = 0$	$sum = 0$
P1 adds sum with a[1]	$sum = 3$	$sum = 7$	$sum = 0$

thread 0 updates the value of `sum`, it is propagated to the main memory. So when thread 1 reads the value of `sum`, it reads a correct value of 3. It then adds 7 to the `sum`, so the `sum` becomes 10, and when it writes to `sum`, it is propagated to the main memory. Therefore, the main memory has a value of 10. However, when thread 0 prints the value of `sum`, it finds a valid copy in the processor 0's cache (a cache hit) and it prints a value of "3", which is also incorrect (despite the fact that the main memory has an up-to-date value of `sum`, which is 10).

Overall, the write policy of caches (either write through or write-back) only dictates how a change of value in a cached copy should be *propagated to the lower level* (main memory), but does not dictate how a change of value in a cached copy should be *propagated to other cached copies*. The problem, of course, does not exist in systems that do not have caches. Since the root of the problem is non-coherent view of values of a single data item in different caches, it is referred to as the *cache coherence* problem.

To enable a coherent view of data values in multiple caches, we need to support cache coherence. At the very least, there has to be a mechanism to propagate changes in one cache to other caches. We refer to this requirement as the *write propagation* requirement. There is another (subtler) requirement for supporting cache coherence which is referred to as *write serialization*. Write serialization is essentially a requirement that multiple changes to a single memory location are seen in the same order by all processors.

To illustrate the need for write serialization, assume that we have four processors: P0, P1, P2, and P3. P0 writes a value of 5 to a certain location/address in its cache and then P1 writes a value of 9 to the same address in its cache. If we only guarantee write propagation but not write serialization, P2 and P3 may see a different order of changes. P2 may see that P0's write occurs before P1's write, therefore it sees a final value of 9. P3 may see that P1's write occurs before

P0's write, therefore it sees a final value of 5, which is incoherent with what P2 sees. Therefore, write serialization is required in order for all processors to have a coherent view of cached values.

Cache coherence is supported using a mechanism called the *cache coherence protocol*. The requirement for the correctness of a cache coherence protocol is that it must ensure write propagation and write serialization.

Cache coherence protocols will be discussed in more details in Chapter 8.

7.2 Memory Consistency Problem

There is another problem that multiprocessor systems must address, which is more subtle than that of the cache coherence problem. This is known as the *memory consistency* problem. Whereas cache coherence determines the requirement for propagating a change of value in a single address in one cache to other caches and the serialization of those changes, memory consistency deals with the *ordering* of *all* memory operations (loads and stores) to *different* memory locations. The problem is referred to as *memory* consistency because unlike *cache* coherence problem that only occurs for systems with caches, memory consistency problem exists even on systems without caches, although caches may exacerbate it.

To illustrate the problem, consider a pair of signal-wait synchronization. Readers may recall that such synchronization is needed for DOACROSS and DOPIPE parallelism (See Chapter 3), among others. Suppose that thread 0 produces a datum that is to be consumed by thread 1. Thread 0 executes the *post* in order to let thread 1 know that the datum is ready. Thread 1 executes a *wait* which makes it block until the corresponding post is executed. One straightforward way to implement a pair of signal-wait is to use a shared variable initialized to 0. The post operation sets it to 1, while the wait operation waits until the value of the variable is set to 1. This is illustrated in Code 7.2. In the illustration, the shared variable that implements the signal-wait pair is `datumIsReady`. Processor 1 (executing thread 1) waits in a loop until `datumIsReady` is set to 1, which indicates that `datum` has been produced.

Code 7.2 A code containing signal-wait synchronization.

P0:	**P1:**
`S1: datum = 5;`	`S3: while (!datumIsReady) {};`
`S2: datumIsReady = 1;`	`S4: print datum;`

Let us consider whether Code 7.2 works as intended on a single processor system, and then whether it works on a multiprocessor system. In a single processor system, the correctness of execution of the code depends on the requirement that statement S1 is executed before statement S2. If for some reasons S2 is executed before S1, then `datumIsReady` is set to 1 before the value of `datum` is produced. Similarly, if S4 is executed before S3, then it may print the value of `datum` that is not yet produced. Hence, the correctness of execution of the code depends on the proper execution ordering of S1 and S2, and S3 and S4. It is crucial that the binary code corresponding to program source code preserves the ordering of the statements. We refer to the order of instructions as they occur in the program source code as the *program order*.

Let us first consider how the program ordering of S3 and S4 is preserved in a single processor. A typical processor today implements out-of-order execution which may reorder instruction execution to exploit instruction-level parallelism, but preserve dependences (data and control flow) and commit instructions in program order. The execution of S4 is control flow dependent on whether the loop condition in S3 is satisfied or not, hence S3 has a control flow dependence with S4. Since the processor obeys control flow dependence, we know that S3 and S4 will be executed in program order.

Let us now consider S1 and S2. The store instructions corresponding to S1 and S2 have neither data dependence (true, anti, or output) nor control flow dependence. During execution, S2 may be performed before S1, but since a processor implements in-order instruction commit, it is guaranteed that instructions corresponding to S1 are committed before S2. Hence, the code will execute correctly unless the program binary itself does not preserve the program order. There needs to be a way to tell the compiler to preserve program order when generating the binary code. To tell the compiler to preserve the program order, the typical solution is to provide a language construct that the compiler understands. One example is that in C/C++, if `datumIsReady` is declared as `volatile`, the compiler knows that any load or store to the variable must not be eliminated or reordered with other instructions that precede or follow it. While such a solution may be overly restrictive (not all loads and stores to a volatile variable need strict ordering), it is very simple to use. Programmers only need to remember which variables are used for synchronization purposes and declare them volatile.

Overall, in a single processor system, for the code to execute correctly, one only needs to ensure that the compiler preserves the program order when there are

accesses to synchronization variables. How about in a multiprocessor system? In a multiprocessor system, thread 0 and thread 1 are executed on different processors, so an extra problem arises. First, even though with respect to processor 0, the instructions of S1 and S2 are committed in program order, they may not appear so to processor 1. For example, the write to `datumIsReady` may be propagated sooner to processor 1 (e.g., due to a cache hit), while the write to `datum` may be propagated later to processor 1 (e.g., due to a cache miss). Hence, preserving program order on a single processor is insufficient for ensuring the correctness of the execution of the code. In a multiprocessor system, therefore, there need to be mechanisms to ensure that acesses of one processor appear to execute in program order to all other processors, at least partially. However, as one may expect, ensuring full program order across processors is expensive in terms of performance, hence some processors only enforce partial program ordering. What types of program ordering are enforced, and what are not, are specified in a *memory consistency model* of the processor.

Basically, system programmers must be aware of the memory consistency model that the multiprocessor system guarantees, and write their code accordingly. Various memory consistency models trade off performance with the ease of programming. Since the optimum point between performance and ease of programming is unclear, various processors today support different memory consistency models. It is critical for programmers to know what model the system they use provides, know how to produce correct code for it, and also realize its performance characteristics.

Memory consistency models will be discussed in more details in Chapter 10.

7.3 Synchronization Problem

Even if cache coherence is supported, the example in Code 7.1 assumes that synchronization primitive required by the `#pragma omp critical` clause is implemented in the system. It leaves the question of how mutual exclusion required by the critical section is enforced. *Mutual exclusion* requires that when there are multiple threads, only one thread is allowed in the critical section at any time.

Mutual exclusion is not a problem that is unique to multiprocessor systems. In a single processor system, for example, multiple threads may time-share a processor, and variables that are potentially modified by more than one thread must be protected in a critical section. A brute-force way to achieve critical section is by

disabling interrupts when mutual exclusion is needed for a specific code segment. Disabling all interrupts ensure that the thread that is executing will not be context switched or interrupted by the OS. Such an approach is costly to use in a single processor system. In a multiprocessor system, besides of being costly, disabling interrupts does not achieve mutual exclusion because other threads in different processors still execute simultaneously.

In order to provide mutual exclusion, suppose that a primitive `lock (lockvar)` that acquires a lock variable `lockvar`, and `unlock (lockvar)` that releases the lock variable `lockvar`, are provided. Also suppose that `lockvar` has a value of 1 if it is acquired by a thread, otherwise its value will be 0. To enter a critical section, a naive implementation of these functions is shown in the following code (Code 7.3).

The code shows that to acquire a lock, a thread will go in a loop waiting for the value of `lockvar` to become 0, indicating a released lock. Then, the thread acquires the lock by setting `lockvar` to 1. This prevents other threads from entering the critical section once the lock is successfully acquired. Finally, when a thread exits the critical section, it sets the `lockvar` back to 0.

Code 7.3 An incorrect implementation of lock/unlock functions.

The high-level language implementation:

```
void lock(int *lockvar) {
  while (*lockvar == 1) {} ;  // wait until released
  *lockvar = 1;  // acquire lock
}

void unlock(int *lockvar) {
  *lockvar = 0;  // release lock
}
```

The corresponding assembly language version:

```
lock: ld   R1, &lockvar     // R1 = lockvar
      bnz  R1, lock         // jump to lock if R1 != 0
      st   &lockvar, #1     // lockvar = 1
      ret                   // return to caller

unlock: st   &lockvar, #0   // lockvar = 0
        ret                 // return to caller
```

The problem with the naive implementation is that the sequence of instruc-

tions that reads the value of `lockvar` and writes to it is not atomic, as evident in the corresponding assembly language version. The load that reads the value of `lockvar` and the store to it are distinct instructions, and there are other instructions between them such as the compare and the branch instructions. The sequence of instructions is not atomic since it can be interrupted in the middle, and can overlap with the same sequence at other processors. This non-atomicity causes a correctness problem as it allows multiple threads to read the same value of `lockvar`, say 0, and acquire it simultaneously. For example, consider the sequence of events shown in Figure 7.2. Say two threads running in processor 0 and processor 1 execute the `lock()` code at about the same time (the example shows thread 1 slightly lagging). Both read `lockvar` and discover its value to be 0. Both compare the value with zero, and the branch is not taken because the value is found to be zero by both processors. Then, thread 0 acquires the lock and writes a value of 1 to `lockvar`. It then enters the critical section. Thread 1 does not know that the lock has been acquired, and it also writes a value of 1 to `lockvar` and enters the critical section. Now there are more than one thread in the critical section at the same time, which is incorrect.

Figure 7.2: Incorrect execution of the naive lock implementation.

Note that the incorrect outcome is unaffected even when the system correctly implements a cache coherence protocol. Cache coherence protocol ensures that a new value written to a location will be propagated to other cached copies. In this example thread 1's `lockvar` copy may get invalidated as a result of thread 0's write to `lockvar`. However, since thread 1 does not attempt to re-read the value of `lockvar` that it has read earlier, it will not see the new value. Instead, thread 1 attempts to write (or overwrite) `lockvar` with a value of 1.

One way to fix the problem is to introduce a support for an **atomic instruction** that performs a sequence of load, modify, and store as a single, unbreakable unit of execution. The word atomic here implies two meanings. First, it implies that either *the whole sequence is executed entirely*, or none of it appears to have exe-

cuted. Second, it also implies that *only one atomic instruction from any processor can be executed at any given time.* Multiple such instructions from several processors will have to be serialized. How can an atomic instruction be supported? Certainly, software support cannot provide any atomicity of the execution of several instructions. Thus, hardware support is required, and will be discussed in more details in Chapter 9.

In this chapter, we will look at the need for hardware support from a different angle, that is, is there an alternative that does not require hardware support? If there is, is it efficient? To answer the questions, consider a software solution for guaranteeing mutual exclusion, such as the Peterson's algorithm [Tanenbaum, 1992]. Peterson's algorithm guarantees mutual exclusion for two threads through the following code (Code 7.4), To enter a critical section, a thread calls the `void lock(..)` function, while to exit a critical section, the thread calls the `void unlock(..)` function.

Code 7.4 Acquiring and releasing a lock in Peterson's algorithm

```
int turn;
int interested[n];  // initialized to 0

void lock (int process) {      // process is 0 or 1
  int other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process && interested[other] == TRUE) {} ;
}
// Post: turn != process or interested[other] == FALSE

void unlock (int process) {
  interested[process] = FALSE;
}
```

Let us see how the Peterson's algorithm works. First of all, notice in `lock(..)` that the condition in which the code exits the while loop and successfully acquires the lock is either because of `turn` is not equal to `process`, or `interested[other]` has a FALSE value. When there is only one thread attempting to acquire the lock, it will successfully acquire the lock because `interested[other]` is FALSE. When a thread exits the critical section, all it needs to do is to set the `interested[process]` to FALSE, allowing the other thread to enter the critical section.

tions that reads the value of `lockvar` and writes to it is not atomic, as evident in the corresponding assembly language version. The load that reads the value of `lockvar` and the store to it are distinct instructions, and there are other instructions between them such as the compare and the branch instructions. The sequence of instructions is not atomic since it can be interrupted in the middle, and can overlap with the same sequence at other processors. This non-atomicity causes a correctness problem as it allows multiple threads to read the same value of `lockvar`, say 0, and acquire it simultaneously. For example, consider the sequence of events shown in Figure 7.2. Say two threads running in processor 0 and processor 1 execute the `lock()` code at about the same time (the example shows thread 1 slightly lagging). Both read `lockvar` and discover its value to be 0. Both compare the value with zero, and the branch is not taken because the value is found to be zero by both processors. Then, thread 0 acquires the lock and writes a value of 1 to `lockvar`. It then enters the critical section. Thread 1 does not know that the lock has been acquired, and it also writes a value of 1 to `lockvar` and enters the critical section. Now there are more than one thread in the critical section at the same time, which is incorrect.

Figure 7.2: Incorrect execution of the naive lock implementation.

Note that the incorrect outcome is unaffected even when the system correctly implements a cache coherence protocol. Cache coherence protocol ensures that a new value written to a location will be propagated to other cached copies. In this example thread 1's `lockvar` copy may get invalidated as a result of thread 0's write to `lockvar`. However, since thread 1 does not attempt to re-read the value of `lockvar` that it has read earlier, it will not see the new value. Instead, thread 1 attempts to write (or overwrite) `lockvar` with a value of 1.

One way to fix the problem is to introduce a support for an **atomic instruction** that performs a sequence of load, modify, and store as a single, unbreakable unit of execution. The word atomic here implies two meanings. First, it implies that either *the whole sequence is executed entirely*, or none of it appears to have exe-

cuted. Second, it also implies that *only one atomic instruction from any processor can be executed at any given time*. Multiple such instructions from several processors will have to be serialized. How can an atomic instruction be supported? Certainly, software support cannot provide any atomicity of the execution of several instructions. Thus, hardware support is required, and will be discussed in more details in Chapter 9.

In this chapter, we will look at the need for hardware support from a different angle, that is, is there an alternative that does not require hardware support? If there is, is it efficient? To answer the questions, consider a software solution for guaranteeing mutual exclusion, such as the Peterson's algorithm [Tanenbaum, 1992]. Peterson's algorithm guarantees mutual exclusion for two threads through the following code (Code 7.4), To enter a critical section, a thread calls the `void lock(..)` function, while to exit a critical section, the thread calls the `void unlock(..)` function.

Code 7.4 Acquiring and releasing a lock in Peterson's algorithm

```
int turn;
int interested[n];  // initialized to 0

void lock (int process) {      // process is 0 or 1
  int other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (turn == process && interested[other] == TRUE) {} ;
}
// Post: turn != process or interested[other] == FALSE

void unlock (int process) {
  interested[process] = FALSE;
}
```

Let us see how the Peterson's algorithm works. First of all, notice in `lock(..)` that the condition in which the code exits the while loop and successfully acquires the lock is either because of `turn` is not equal to `process`, or `interested[other]` has a FALSE value. When there is only one thread attempting to acquire the lock, it will successfully acquire the lock because `interested[other]` is FALSE. When a thread exits the critical section, all it needs to do is to set the `interested[process]` to FALSE, allowing the other thread to enter the critical section.

Figure 7.3 shows how the algorithm ensures mutual exclusion when there is no race in acquiring the lock. The figure illustrates thread 0 which successfully acquires the lock, and while it is in the critical section, thread 1 attempts to acquire the lock. Because thread 0 is in the critical section, `interested[0]` is TRUE and therefore, thread 1 waits in the while loop. When thread 0 exits the critical section, it sets `interested[0]` to FALSE. That allows thread 1 to exit its while loop and enter the critical section. Mutual exclusion is ensured because thread 1 cannot enter the critical section while thread 0 is still in it.

Figure 7.3: Execution of the Peterson's algorithm without races.

Figure 7.4 shows how the algorithm ensures mutual exclusion when two threads try to simultaneously acquire the lock. Suppose that both threads assign a value TRUE to `interested[process]` simultaneously. Then, they try to write their own thread ID to `turn`. Since `turn` is a shared variable, one thread will overwrite the value written by the other. The figure assumes that thread 0 is slightly faster and writes 0 to `turn`, while thread 1 which is slightly late overwrites `turn` with a value of 1. Then, both threads execute the while loop. Because the value of `turn` is 1, thread 0 exits the loop, successfully acquires the lock, and enters the critical section. Thread 1 must wait in the loop because `turn` is 1, and `interested[0]` is TRUE. Only after thread 0 exits the critical section and sets `interested[0]` to FALSE, thread 1 can enter the critical section.

While the software solution (the Peterson's algorithm) can also achieve mutual exclusion, it suffers from a lack of scalability. Consider the condition for exiting the while loop in Peterson's algorithm when the algorithm is modified to work for

Figure 7.4: Execution of the Peterson's algorithm with a race in acquiring the lock.

n threads rather than two threads. If there are n threads, then there are n elements in the `interested[]` array. In order for a thread to acquire the lock, it must test all elements in the array `interested` (except the one that corresponds to the thread itself) to check whether any of them has a TRUE value. Therefore, the lock acquisition algorithm complexity is $\Theta(n)$. Compare this against Code 7.3, which only tests one variable `lockvar` in order to acquire a lock. Of course, Code 7.3 is incorrect unless it is used in conjunction with an atomic instruction supported by the hardware. This points out that it is important to have hardware support for synchronization in order to reduce synchronization overheads and enable scalable synchronization.

Finally, it is important to note that the correctness of a synchronization primitive is related to the memory consistency model provided by the processor. By nature, synchronization operation is used to impose ordering between memory accesses performed by different threads. However, the ordering of the memory accesses performed by a synchronization operation and those performed outside the synchronization cannot be guaranteed unless the hardware provides either a strict memory consistency model, or an explicit mechanism that can order memory accesses. This will be discussed further in Chapter 9.

7.4 Exercise Problems

1. (*10 Points*) **Register Allocation**. Suppose that you are writing a compiler algorithm that decides what variable can be allocated in a register. A register allocation allows the variable to be accessed in the register without involving a load or store to read from or write to it. In order not to create cache coherence problem, which type of variable can be register allocated vs. cannot be register allocated?

Variable type	Can be register allocated?
Read-only	
R/W Non-Conflicting	
R/W Conflicting	

If there are variable types that you think cannot be safely allocated in registers, give explanation of why it is so (see Section 3.4 for what each variable type refers to).

2. (*10 Points*) **Cache Coherence**. Assume that we have 2-processor system with each processor having a single data cache. Also assume that we have an identical access sequence when computing sum, as follows:

   P1: prefetch the block containing sum from &sum (i.e., the location of sum)
   P2: read the value of sum from &sum
   P2: write sum+7 into &sum
   P1: read the value of sum from &sum
   P1: write sum+3 into &sum
   P1: read sum to print its value

   Each access may hit or miss in the cache, depending on the cache configuration that will be given.

   For each item below, determine if the cache design or protocol given has correctness problems. Also give the final values in all caches, and in the main memory.

   (a) Explain whether there is a correctness problem in a shared cache architecture, where there are two processors, which share a single level cache (the L1 cache). Assuming a line is not replaced after it is brought into the cache, give the final values of sum at the cache and in the main memory. Assume the cache uses a write back policy.

(b) Explain whether there is a correctness problem, in a configuration where each processor has a single-level write-through cache. Give the final values of sum at both processors' caches and in the main memory. Does your answer depend on whether the cache have a write allocate or write no-allocate policy?

(c) Explain whether there is a correctness problem, in a configuration where each processor's L1 cache uses a policy of evict-on-write, i.e. a block is replaced immediately after it is written to.

(d) If each processor has a cache, explain whether there is a correctness problem if we do not allow sum to be cached.

(e) Explain whether there is a correctness problem in a partially-shared cache architecture, where each processor has a separate write-through L1 cache, but the L2 cache and all lower level memory hierarchy components are shared by all processors. Give the final values of sum in all caches and in the main memory.

(f) Explain whether there is a correctness problem in an architecture with several processors, each with its own cache, in which a line in memory can only be cached in at most one cache. If the line is cached at P_i, and P_j requests the line on a read or a write, it is removed from P_i's cache and placed in P_j. Assuming a line is not replaced after it is brought into the cache except when it is moved to another processor's cache, give the final value of sum at both processors' caches and in the main memory.

(g) Suppose we run a sequential program on a multiprocessor system (it will only occupy one processor). Each processor has a private cache but shares the memory with others. To achieve balanced processor utilization, the operating system frequently migrates one process from one processor to another. Explain whether there is a correctness problem in this scenario.

3. **Peterson's algorithm.** Write Peterson's algorithm for lock acquisition and release involving 3 threads.

Chapter 8

Bus-Based Coherent Multiprocessors

Contents

8.1 Overview **196**
8.1.1 Basic Support for Bus-Based Multiprocessors 199
8.2 Cache Coherence in Bus-Based Multiprocessors **203**
8.2.1 Coherence Protocol for Write-Through Caches 203
8.2.2 MSI Protocol with Write Back Caches 206
8.2.3 MESI Protocol with Write Back Caches 215
8.2.4 MOESI Protocol with Write Back Caches 222
8.2.5 Update-Based Protocol with Write Back Caches . . . 229
8.3 Impact of Cache Design on Cache Coherence Performance **235**
8.4 Performance and Other Practical Issues **236**
8.4.1 Prefetching and Coherence Misses 236
8.4.2 Multi-Level Caches 237
8.4.3 Snoop Filtering 238
8.5 Exercise Problems **240**

As we have discussed in Chapter 7, hardware support for building a shared memory multiprocessor system that guarantees correct and efficient execution of parallel programs must address cache coherence, memory consistency, and support for synchronization primitives.

The objective of this chapter is to discuss the design of cache coherence protocols. Despite having seemingly simple objectives of ensuring a consistent view

of cached values through write propagation and write serialization, there is a wide range of design choices for implementing a cache coherence protocol. At the most basic levels, we must choose how write propagation is performed: either through directly updating all cached values upon a write by a processor, or through invalidating all other cached values upon a write by a processor – forcing other caches to reload the new values when the processors access them. The former strategy is referred to as *write update*, while the latter is referred to as *write invalidate*. They differ in characteristics and suit different program behavior. The write invalidate strategy is advantageous when a write to a cache block tends to be followed by subsequent writes to the same block. In such a case, the invalidation occurs only once and subsequent writes do not generate any more traffic. The write update strategy is advantageous when a write to a cache block tends to be followed by reads by other processors. Updating other cached values allow other processors to immediately obtain the new value by reading them from the caches. In such a case, the write invalidate strategy is disadvantageous because when copies of a heavily shared data block are invalidated, each processor will suffer a cache miss to reload the block. Such "coherence misses" do not occur in the write update strategy. However, the write update strategy sometimes keep unnecessary data fresh for too long in other caches, hence there are fewer cache blocks available for more useful data, leading to an increase in other types of misses (capacity and conflict).

Most implementations today use the write invalidate strategy for various reasons. We will start describing the write invalidate strategy, and discuss different protocols that rely on this strategy. The protocols are described in increasing sophistication, and one builds upon another. More sophisticated protocols use less bandwidth but are more complex to implement. Readers are advised to read the protocols sequentially. We will then discuss one write update coherence protocol. Finally, we will discuss other related issues: performance, impact on various aspects of cache design such as prefetching, multi-level caches, and snoop filtering.

8.1 Overview

Figure 8.1 shows how multiple processors can be interconnected. They differ in the level of the memory hierarchy they are interconnected at, and what interconnection is used. In a shared cache organization, the cache is shared by all processors. Each processor can access any part of the cache directly. The advantage

of this organization is that cache coherence support is not needed because there is only a single cache, and any block can be cached at only one location. The disadvantages of this organization are that it requires the processors and caches to be in a very close proximity to one another, otherwise the time to access the cache will be very high. In addition, when there are multiple cache banks, the interconnection between processors and cache banks is all-to-all, which is expensive to support. Therefore, such an organization is not scalable.

Figure 8.1: Various ways to organize a shared memory multiprocessor system.

The interconnection can be moved to a lower level between the caches and the main memory. In this approach, two organizations are possible. First, the interconnection may be a bus, in which case we have a bus-based multiprocessor. Bus-based multiprocessors are often referred to as **Symmetric Multi-Processors** (SMPs), because the memory latency is the same for all processors. Note, however, that SMP is an over-used term that sometimes refers to multiprocessor systems in which the memory latencies of different processors do not differ by *much*, with "much" being ambiguous. A more general category that includes a bus-based multiprocessor is that in which point-to-point interconnection is used in a

"dance-hall" configuration, where processors and caches reside in one side of the interconnection while main memory resides in another side of the interconnection. The advantage of this dance-hall organization over the shared cache organization is that each cache is local to a processor, allowing a fast cache access time to the local cache. The disadvantage is that while it can scale to a higher number of processors compared to the shared cache organization, it is still limited in that when the number of processors is very large, the access to memory will be slow. In the bus-based multiprocesor, the scalability restriction is reached earlier because of bus bandwidth saturation. A bus is a medium shared by all processors, and the more processors are added, the longer and slower the bus becomes.

Another organization is *distributed shared memory* (DSM), in which memories are interconnected. This organization is the most scalable. With a large number of processors, the interconnection does increase the average access time to memory, although access to the local memory is still fast. However, a DSM is more scalable than the dance-hall organization because the access time to main memory is already high to begin with (in the order of tens to hundreds of nanoseconds), so adding extra few tens or hundreds of nanoseconds in latency to access remote memories only increase the overall latency by less than an order of magnitude. In contrast, adding tens to hundreds of nanoseconds to the access time of some segments of the cache that are remote will significantly reduce the overall cache performance. In addition, caches filter out most memory accesses, so the main memory is accessed a lot less frequently compared to caches. Hence, an increase in the access time to main memory makes a smaller impact on the overall performance than the same amount of increase in the access time to caches.

Note that the organizations are not exclusive. In reality, various levels may be interconnected with different organizations. For example, in a multicore chip with 2 to 4 cores, because of the physical proximity of the processors, it is feasible to have the shared cache organization at the L2 cache level. Various multicore chips may be interconnected together to form a larger system, and they may use a bus-based, dance-hall, or for a very large system, a DSM organization.

The attractiveness of the bus-based multiprocessors, which is the focus of this chapter, is that it only requires a few modifications to a single processor system, which already relies on a bus for the connection to the main memory. In addition, as will be discussed later, the support needed for cache coherence also includes a *finite-state machine* (FSM) of cache state that is an extension to what are already used in a single processor system. For example, a write-back cache already as-

sociates each cache block with a state, in which the state values include *invalid*, *valid clean*, and *dirty*. Finally, the cache coherence protocol for a bus-based multiprocessor is simpler than for other organizations, so we first use a bus-based multiprocessor as a starting point to introduce basic concepts and techniques used in cache coherence protocols.

8.1.1 Basic Support for Bus-Based Multiprocessors

Bus is perhaps the simplest interconnection fabric that can be used to connect multiple processors into a single multiprocessor system. At the high level, a multiprocessor bus is a set of wires shared by multiple processors. In order for a processor to communicate with another, it must post its command, address, and data on the bus, and other processors must listen to the bus to check if the bus contains any command, address, or data that are relevant to them.

A bus logically has three different types of lines: command lines, address lines, and data lines. The command lines are used to place a bus command such as a read or a write command. In a write back cache, a read miss will generate a bus read command, and a write miss will also generate a bus read command because of the write-allocate policy. A bus write command is generated when a block needs to be written back to the memory, such as when a dirty block is evicted. The address lines are used to place the address that the command is to be applied to. The data lines are used to place the data that is returned by the memory on a bus read, or data that is sent to the memory on a bus write. In some buses, the lines are *physically* separated, while in others, they are shared and multiplexed (e.g. Hypertransport [Hypertransport Consortium, 2004]). When lines are multiplexed, messages sent on the lines must be distinguished from one another. This can be done by encapsulating each message into a series of packets, with the head packet containing information to identify the following packets.

A bus can be **synchronous** or **asynchronous**. In a synchronous bus, all devices connecting to the bus are clocked with the same clock signal, transmitted through the control lines. In an asynchronous bus, the devices do not share a common clock signal. Transmitting data through a synchronous bus is simple because the devices can agree on the timing of when a particular message (command, address, or data) will be sent. Each device can follow a finite state machine to know what message will be posted next and at exactly which bus cycle. In contrast, in an asynchronous bus, the device must establish communication through a series of requests and replies in a handshake protocol. Such a protocol makes the device

more complex and communication slower. Since a processor-memory bus or a multiprocessor bus are co-designed with the devices that will connect to it, the number of such devices are fixed, and the bus is relatively short, a synchronous design is usually used. I/O buses, on the other hand, must be designed with a wide range of devices and a variable number of devices in mind. I/O buses also tend to be relatively long, hence a common clock signal in a synchronous design cannot provide a high frequency due to the high clock skew.

Depending on the design, a bus can transmit data only once per cycle, or can also transmit data at both the rising and falling edges of the clock. The latter is known as a **double rate** bus.

A bus transaction goes on three phases. The first phase is *arbitration* which selects the requester that is allowed to use the bus. Arbitration is important in a multiprocessor system to prevent requests from different processors to collide. A bus grant signal is given by the arbiter to the requestor that wins. In a bus-based multiprocessor system, the requestors are the processors (more accurately, the coherence controller below the lowest level on-chip cache), and the memory controller at the memory side. When the requestor receives a bus grant signal, it places the block address on the address lines, and for a bus read, it waits for data to be returned by the memory. In an **atomic** bus, the bus is reserved for the entire duration of the transaction (until data is returned by the memory). In a **split transaction** bus, the bus lines are released as soon as the transfer of address or data is completed. Obviously, with the access time to the DRAM main memory taking 30-70 ns, reserving the bus for the entire duration of a transaction significantly reduces the available bus bandwidth. Hence, many recent systems rely on a split transaction bus. These phases (arbitration, command, and data transfer), can be pipelined with respect to one another to improved throughput.

Each type of bus in a multiprocessor system may need a separate arbiter. Arbitration logic tries to give a higher priority to a more important device but still ensure fairness. For example, requests from caches may be given a higher priority compared to requests from I/O devices.

Bus-based multiprocessors extend the existing mechanisms of the bus (bus transactions), and the cache (cache states). Figure 8.2 illustrates the components added or modified. The cache tag array is augmented with extra bits for encoding new cache states (shown in shaded regions). Each cache block is associated with a state value, which means that coherence is maintained at the granularity of cache blocks.

Figure 8.2: Support for a bus-based multiprocessor system at each node.

A new component called a **coherence controller** is added to the processor side, and also to the memory side. The **outstanding transaction table** keeps track of bus transactions that have not completed. This is a structure required in the split transaction bus organization. In the split transaction bus, multiple requests to different addresses can be placed on the bus even when the oldest request has not obtained its data. Therefore, the controller must be able to keep track of multiple outstanding bus transactions. Using the table, when the data block is returned, the transaction id is checked against entries in the table to determine which transaction it corresponds to. The table needs to be sized appropriately based on speed of the processor, cache miss rates, and the latency to the off-chip memory, in order to prevent the processor from stalling when its request finds the table already full.

Each coherence controller has a component called a **bus snooper**. The role of the snooper is to snoop each bus transaction. For each snooped bus transaction, the coherence controller checks the cache tag array to see if it has the block that is involved in the transaction, checks the current state of the block (if the block is found), and reacts accordingly by responding with data or by changing the state of the block. To determine what new state a block should transition into upon an event, a **Finite State Machine** (FSM) implementing the cache coherence protocol

is employed. Data that is sent out is placed in a queue called the **write-back buffer**. The write-back buffer keeps a cache block that needs to be written back (or **flushed**) either due to natural cache eviction of a dirty block, or due to the need to respond to snooped bus transaction.

At the memory side, the memory controller must also have a snooper to monitor all bus transactions. Depending on the type of the bus transaction it sees, it must choose whether to respond by supplying or accepting data, or ignore it. However, the memory controller does not keep states for blocks in memory, so it operates differently than the coherence controllers at the processor side.

Note that we have been implicitly assuming that the coherence unit is a cache block, which is common in implementations. The reason for why this assumption is reasonable is because other cache management functions (placement, eviction, and state bookkeeping) are all performed at the granularity of cache blocks. Thus, it is natural and simple to use the same granularity for the purpose of keeping caches coherent. Later, we will see how this assumption may introduce a problem in *false sharing*, a special type of cache misses that occur due to keeping coherence at the granularity of the cache block size.

Recall that the requirements for supporting cache coherence are write propagation and serialization. There are two major strategies for ensuring write propagation. In **write update** protocols, when a write is snooped on the bus by a snooper, the snooper reacts by updating the local cached copy, in effect propagating the new value to the local cache. In **write invalidate** protocols, when a write is snooped on the bus by a snooper, the snooper reacts by making the local cached copy invalid. So the new value is not propagated right away. Only later when a processor accesses the block that was invalidated, it suffers a cache miss that requires the new block to be brought into the cache. At that time, the processor sees the new value.

For write serialization, the bus is a shared medium that already introduces serialization of all reads and writes. In fact, this is stronger than the write serialization requirement that states that writes to a *single* block address have to be seen in the same order by all processors (but writes to different addresses do not need to be serialized). Hence, the bus introduces a natural mechanism to serialize writes, so as long as all coherence controllers obey the bus ordering, then a correct coherence protocol can be supported in a straightforward manner. For example, say a processor wants to write to a block address. It tries to get access to the bus in order for the write to be serialized. In the mean time, the coherence controller snoops

a bus write request to the same block address. To preserve write serialization, the controller must obey the bus order. Since its write request has not been granted access to the bus, it has lost the competition. It has to respond to the bus write transaction it snoops from the bus, by updating or invalidating the corresponding cache block first. It also has to cancel its bus write request, and retry the write request on the cache block (which may now generate a new bus request that is different than the canceled one). Overall, as long as the write order on the bus is obeyed by all cache controllers by responding to transactions in bus order, then write serialization can be guaranteed.

8.2 Cache Coherence in Bus-Based Multiprocessors

8.2.1 Coherence Protocol for Write-Through Caches

The simplest cache coherence protocol is the one that is built over write-through caches. Let us assume a single level cache, which may get requests from the processor side, as well as from the bus side as snooped by the snooper. Processor requests to the cache include:

1. **PrRd**: processor-side request to read to a cache block.
2. **PrWr**: processor-side request to write to a cache block.

Snooped requests to the cache include:

1. **BusRd**: snooped request that indicates there is a read request to a block made by another processor.
2. **BusWr**: snooped request that indicates there is a write request to a block made by another processor. In the case of a write-through cache, the BusWr is a write through to the main memory performed by another processor.

Each cache block has an associated state which can have one of the following values:

1. **Valid** (V): the cache block is valid and *clean*, meaning that the cached value is the same with that in the lower level memory component (in this case the main memory).
2. **Invalid** (I): the cache block is invalid. Accesses to this cache block will generate cache misses.

There is no dirty state in the write-through cache since all writes are written through to the lower level, all cached values are clean. The write through mechanism is designed to propagate writes to the lower level of the memory hierarchy. We will see how cache coherence controllers in other processors can snoop these bus transactions and use them for propagating values to caches, not just to the lower level memory hierarchy. We assume that the caches use *write no-allocate* and *write invalidate* cache coherence policy.

The finite state machine corresponding to the coherence protocol for write-through caches is shown in Figure 8.3. In the figure, the response to processor-side requests is shown on the left part, while the response to the snooper-side requests is shown on the right part. They are separated for ease of illustration purpose only; the actual implementation uses a single finite state machine that can respond to both processor-side and snooper-side requests. The figure shows the transition from one state to another, and the label of the edge shows the triggering event and the action or response taken as a result of the event.

Figure 8.3: State transition diagram for the cache coherence protocol for write-through caches.

The key to a write invalidation protocol is that a processor can modify a cache block after invalidating other copies (in other caches). Hence, the cache block that is being modified resides in only one cache, the one that belongs to the processor that writes to it. By obtaining an **exclusive ownership** of the block, the processor has ensured that no other processors are also writing to the block.

In Figure 8.3, the *I* state represents two cases: a case in which the block is not cached, or when the block is cached but its state is invalid. Note that no state is actually maintained for blocks that are not cached. However, they are treated in the same way as blocks that are cached and in an invalid state. Let us consider the left part of the figure that shows a reaction to a processor read or write request. First, consider the case when the cache block is in "I" (invalid) state. When there is a processor read request, the processor suffers a cache miss. To load the datum into the cache, a BusRd is posted on the bus, and the memory controller responds to the BusRd by fetching the block from the main memory. When the block is fetched, it is placed in the cache, and the state is set to "V" or valid. When there is a processor write request, since the cache uses a write no-allocate policy, the write is propagated down to the lower level using the BusWr transaction without fetching the block into the cache. Therefore, the state remains invalid.

Let us consider the case in which the cache already has the block in the valid state. On a processor read, the block is found in the cache and data is returned to the processor. This does not incur a bus transaction since it is a cache hit, and the state remains unchanged. On the other hand, on a processor write, the block is updated, and the change is propagated down to the main memory through the BusWr transaction. The state remains valid (unchanged).

Now let us look at how the finite state machine reacts to snooped bus transactions. If the cache does not have the block or the block is in Invalid state, all snooped BusRd or BusWr should not affect it, so they are ignored. If the cache block is in a valid state, when a BusRd transaction is snooped, that means another processor suffers a read miss and tries to fetch the block from the main memory. Therefore, the state remains valid, and no action is taken. However, if a BusWr is snooped, it indicates that another processor wants to write to the block and attempts to obtain an exclusive ownership of the block. Hence, in this case, the state must be invalidated by transitioning from the valid to Invalid state.

Write propagation is ensured through two mechanisms. First, by invalidating other copies on a write to a cache block, other caches are forced to suffer a cache miss to reload the block. Secondly, by relying on the write-through mechanism,

the reload can be supplied by the main memory which has the latest value of the block.

The drawback of using a write-through cache is that it requires a high amount of bandwidth. This is because there tends to be temporal and spatial locality of writes to a cache block. Each write in the write-through cache incurs a BusWr transaction that takes up bus bandwidth. In contrast, in a write back cache, if a byte/word in a cache block is written multiple times, or if there are multiple writes to different bytes/words in the same cache block, it only uses the bus bandwidth once to invalidate other caches. Therefore, using write-through caches in a bandwidth-restricted architecture prevents scalability of the system since bandwidth gets saturated more quickly.

8.2.2 MSI Protocol with Write Back Caches

Using write back caches can save bandwidth consumption significantly over using write through caches. In this section, we will discuss how a cache coherence protocol can be built over write back caches. The simplest of such protocols is the MSI protocol. Note that a write back cache has a dirty state that indicates whether any part of the block has changed since it was loaded into the cache. The same concept of cache states is extended to build a coherence protocol.

In the MSI protocol, processor requests to the cache include:

1. **PrRd**: processor-side request to read to a cache block.

2. **PrWr**: processor-side request to write to a cache block.

Bus-side requests include:

1. **BusRd**: snooped request that indicates there is a read request to a cache block made by another processor.

2. **BusRdX**: snooped request that indicates there is a *read exclusive* (write) request to a cache block made by another processor.

3. **Flush**: snooped request that indicates that an entire cache block is written back to the main memory by another processor.

Each cache block has an associated state which can have one of the following values:

1. **Modified** (M): the cache block is valid in only one cache, and the value is (likely) different than the one in the main memory. This state extends the meaning of the dirty state in a write back cache for a single processor system, except that now it also implies exclusive ownership. Whereas dirty means the cached value is different than the value in the main memory, modified means both the cached value is different than the value in the main memory, and it is cached only in one location.

2. **Shared** (S): the cache block is valid, potentially *shared* by multiple processors, and is clean (the value is the same as the one in the main memory). The shared state is similar to the valid state in the coherence protocol for write through caches.

3. **Invalid** (I): the cache block is invalid.

We will also define several terms that will be used throughout. The first is that the cache block state is analogous with a permission to perform read/write on the block without having to go to the bus. The Invalid state is analogous to a no-read/no-write permission because accessing it is Invalid and generates a cache miss that goes to the bus. The shared state is analogous to a read/no-write permission because reading the block does not generate a bus transaction but writing will generate a bus transaction to invalidate other caches. Finally, the Modified state is analogous to a read/write permission because reading from or writing to the block is permitted without generating a bus transaction; we know that there are no other cached copies. Using this analogy, changing the state from less restrictive permission to more restrictive permission (e.g., from M to S) can be referred to as a **downgrade**, while changing the state from more restrictive permission to less restrictive permission (e.g., from S to M) can be referred to as an **upgrade**.

A downgrade request in which the final state is "invalid" is referred to as **invalidation**. A downgrade request in which the final state is "shared" is referred to as **intervention**.

We assume that the caches use *write allocate* and *write invalidate* cache coherence policy. The finite state machine corresponding to the MSI coherence protocol for write-back caches is shown in Figure 8.4. In the figure, the response to processor-side requests is shown on the left part, while the response to the snooper-side requests is shown on the right part. They are separated for ease of illustration purpose only; the actual implementation uses a single finite state machine that can respond to both processor-side and snooper-side requests.

Figure 8.4: State transition diagram for the MSI cache coherence protocol for write back caches.

As before, the invalid (I) state represents two cases: a case in which the block is not cached, or when the block is cached but its state is invalid. Let us consider the left part of the figure that shows a reaction to a processor read or write request. First, consider when the cache block is in "I" (invalid) state. When there is a processor read request, a cache miss occurs. To load the data into the cache, a BusRd is posted on the bus, and the memory controller responds to the BusRd by fetching the block from the main memory. When the block is fetched, it is placed in the cache, and the state is set to "S" or shared. The shared state does not distinguish between the case in which there is only one cached copy, and the case in which there are multiple copies in different caches. When there is a processor write request, the cache must allocate a valid copy of the cache block, and to do that, it posts a BusRdX request on the bus. Other caches will respond by invalidating their cached copies (to be discussed later), and the memory responds by supplying the requested block. When the requestor gets the block, it is placed in the cache in the "M" or Modified state. Then, the processor can write to the cache block.

Suppose now the cache already has the block in the shared state. On a processor read, the block is found in the cache and data is returned to the processor. This does not incur a bus transaction since it is a cache hit, and the state remains unchanged. On the other hand, on a processor write, there may be other cached copies that need to be invalidated, so a BusRdX is posted on the bus, and the state

transitions to modified.

If the cache block is present in the cache in the Modified state, reads or writes by the processor do not change the state, and no bus transaction is generated since we are sure that no other cached copies exist; an earlier invalidation has invalidated all other cached copies.

Now let us look at how the finite state machine reacts to snooped bus transactions. If the cache does not have the block or the block is in an Invalid state, any snooped BusRd or BusRdX should not affect it, so they are ignored.

If the cache block is in a shared state, when a BusRd transaction is snooped, that means another processor suffered a read miss and is trying to fetch the block. Since the memory supplies the data, no action is taken by the cache and the cache block state remains shared. However, if a BusRdX is snooped, it indicates that another processor wants to write to the block, and therefore the local cached copy is invalidated (by transitioning into the Invalid state).

If the cache block is in the Modified state, the copy in the cache is the only valid copy in the entire system (no other cached copies exist and the value in the main memory is stale). Therefore, when a BusRd transaction is snooped, indicating that a processor suffered a read miss and trying to fetch the block, the processor that has the modified copy of the block must respond by flushing the block on the bus. After flushing, the resulting state is the shared state. The cache that issues the BusRd request must pick up the flushed block as a reply to its request and place it in its cache in a shared state. In addition, the main memory must also snoop the flushed block and update the corresponding block in the main memory. This is due to a requirement called **clean sharing** in which if a block is shared in multiple caches, its value has to be clean, i.e. the same with the one in the main memory. Other protocols that allow **dirty sharing** will be discussed in later sections.

If a BusRdX transaction is snooped, the processor that has the modified copy of the block must respond by flushing the block on the bus, and then transitioning into the Invalid state. This flush may sound redundant since the processor that issues a BusRdX is going to overwrite the cache block. So why flush the block that is going to be overwritten? The answer to this is that a typical cache block is large, containing multiple bytes or words. The bytes that have been modified are not necessarily the same bytes that another processor wants to overwrite or modify. Another processor may want to write to other bytes, but read from the bytes that have been modified. Therefore, flushing the entire block is a correctness

requirement, and the processor that issues the BusRdX must pick the block up and place it in the cache prior to writing to it.

Write propagation is ensured through two mechanisms. First, by invalidating other copies on a write to a cache block, other caches are forced to suffer a cache miss to reload the block. Secondly, by flushing a modified block upon snooping a BusRd or BusRdX request, the flushed block can be picked up by the requesting processor/cache, ensuring that it gets the latest value of the block.

Compared to the write-through cache, the MSI protocol requires a much smaller amount of write bandwidth. If a block is written multiple times (due to temporal or spatial locality), only one invalidation is generated, and the rest of the writes occur locally in the cache since the cache block is in the Modified state.

However, there is a serious drawback with the MSI protocol. Consider a processor that wants to read blocks and then write to them, without other processors sharing the blocks. In this case, for each read-then-write sequence, two bus transactions are involved: a BusRd to fetch the block into the shared state, and a BusRdX to invalidate other cached copies. The BusRdX is useless since there are no other cached copies, but the coherence controller has no way to know that there are no other cached copies. This problem penalizes the system performance in two cases. The first case is when a sequential application is running. No other threads run and share the data with the application, yet the application uses more bandwidth compared to when it runs on a single processor system. In addition, the snoopers and cache coherence controllers' occupancy are increased due to having to react to more bus requests. The second case is for parallel programs that have been heavily tuned to reduce data sharing across threads. Such programs incur very little sharing, so most cache blocks are only stored in one cache. Hence, most BusRdX is unnecessary since it does not result in invalidations. Unfortunately, they are penalized in the same way as programs that have heavy sharing.

To remove the drawback of the MSI protocol, a state can be added in order to distinguish between a block that is clean and exclusively cached versus one that is stored at multiple caches. Such a protocol will be detailed in the next section when we discuss the MESI protocol.

To illustrate how MSI works, consider the example in Table 8.1. The table shows requests made by various processors in the following format: Rx or Wx, where R stands for a read request, W stands for a write request, and x stands for the ID of the processor making the request. The bus request generated as a result of the processor request is also shown. The new states at the different

Table 8.1: Illustration of MSI protocol operations.

	Request	P1	P2	P3	Bus Request	Data Supplier
0	Initially	–	–	–	–	–
1	R1	S	–	–	BusRd	Mem
2	W1	M	–	–	BusRdX	Mem
3	R3	S	–	S	BusRd	P1's cache
4	W3	I	–	M	BusRdX	Mem
5	R1	S	–	S	BusRd	P3's cache
6	R3	S	–	S	–	–
7	R2	S	S	S	BusRd	Mem

caches *resulting* from the request, are shown in the table. The last column shows where data for the request is supplied to the local cache from: the main memory or another cache. In the example, assume that initially the cache is empty.

On a read from Processor 1 (the first request), a BusRd is posted, the main memory responds with the block; Processor 1 gets the block and stores it in its cache with a shared state.

On a write from Processor 1, it does not know if other caches have the same block or not, so a BusRdX is posted on the bus to invalidate the same block in other caches. The main memory snoops the BusRdX and responds with the block. Note that here the memory controller does not know whether supplying the data is necessary or not, since it cannot distinguish between the case in which Processor 1 already has the block but needs to upgrade its state to modified, or it does not have a block and needs to fetch it from memory. After posting the BusRdX on the bus, effectively, the write has been serialized, so the cache state can safely transition to modified, and the block can be written to. Note that at some time later, the reply from memory comes. The processor can either ignore the reply, or put it in its cache in the Modified state while delaying its write until it has finished doing so.

When Processor 3 makes a read request, a BusRd is posted on the bus. Processor 1's snooper picks it up, checks its cache tag, and finds that it has the block in modified state. This means that Processor 1 has the latest (and the only valid) copy, and flushes it in response to the snooped request. The block state transitions

to shared after that. In the mean time, the memory controller also attempts to fetch the block from the main memory because it does not know if eventually a cache will supply the data or not. Processor 3 snoops the flush, and by matching the address of the block being flushed with its outstanding read transaction, it knows that the flushed block should be treated as the reply to its read request. So the block is picked up and stored in its cache in the shared state. The main memory which has been trying to fetch the block from memory will also snoop the flushed block, pick it up, cancel its memory fetch, and overwrite the stale copy of the block in memory. Supporting this mechanism means that the memory controller must also have a table keeping its outstanding fetches to the main memory. If a flush is detected, it must cancel its outstanding fetch to the same address, perhaps by marking the outstanding entry so that when the fetched block arrives, it can be discarded. It must also create a new entry that writes to the block with the flushed block.

At this point, readers may wonder whether there is a correctness problem if the main memory has supplied a stale version of the cache block before the **owner** (the processor that has the block in Modified state) has a chance to flush its block. The answer to this is that this cannot be allowed to occur. One possible solution is to give enough "fixed amount" of time for all processors to finish snooping and responding to a bus request, before the memory controller replies with a block. For example, even if the memory controller has fetched a block from main memory, the block must be kept in a table until it is verified that no cache will supply the block through a flush. Another possible solution is to have a special "SNOOP-DONE" bus line that is asserted by all coherence controllers as they finish snooping the transaction and check their cache tags. The line logically implements an AND operation, which means that it is asserted only when *all* processors have responded. Only when the SNOOP-DONE is set and no flush has been posted, can the memory controller safely supply the block.

Next, Processor 3 has a write request. It posts a BusRdX on the bus to invalidate copies in other caches. Processor 1's coherence controller responds by invalidating its copy. Processor 3's cache block state transitions to modified.

When Processor 1 attempts to read the block, it suffers a cache miss as a result of the earlier invalidation that it received. This type of miss is referred to as a **coherence miss** because it arises due to coherence actions (invalidations). So Processor 1 posts a BusRd and Processor 3 responds by flushing its cache block and transitions its state to shared. The flushed block also updates the main memory

copy; as a result, the cache blocks are now clean.

When Processor 3 reads the block, it finds it in the shared state in its cache. Since it has a valid copy, it has a cache hit and does not generate a bus request.

Finally, Processor 2 attempts to read the block and posts a BusRd on the bus. The memory controller fetches the block from the memory and the value in the main memory is valid because it has been updated by the earlier flush. Note that in this case, actually both Processor 1 and 3 have the same valid copy of the block. Should Processor 1 or Processor 3 flush the block to supply the block? Although not required for correctness, they can definitely do that, and performance-wise it makes sense if they can supply the block faster than the memory can read from DRAM and supply the same block. In fact, in the MESI protocol, **cache-to-cache transfer** is employed that allows this to occur.

Earlier we have pointed that on a BusRdX, the memory controller has no way to distinguish between the case in which the requesting cache already has a copy of the block but needs to upgrade its state, or it does not have a copy in the cache block and needs to fetch the block from the main memory. In the former case, the memory controller performs a useless task fetching the block from the main memory and supplies it on the bus, only to be discarded by the requesting cache. To avoid this waste, we can distinguish between the two cases by introducing a new bus request called a bus upgrade (**BusUpgr**). If a cache already has a valid copy of the block and only needs to upgrade its permission, it posts a BusUpgr instead of BusRdX. On the other hand, if it does not have the block in the cache and needs the memory or another cache to supply it, it posts a BusRdX. The memory controller responds differently in these two cases. It ignores the BusUpgr, but fetches the block when it snoops a BusRdX. With this small modification, the state transition diagram becomes one shown in Figure 8.5 with the BusUpgr shown in bold fonts. The affected operation in the example in Table 8.2 are also shown in bold fonts.

Another choice in the coherence protocol is the state a cache block should transition into when it receives an intervention request, i.e. when the snooper snoops a BusRd to a locally cached block that is in Modified state. In the MSI protocol shown earlier, it transitions into a shared state. An alternative is to transition into an Invalid state. The rationale for this is that if a read from another processor is often followed immediately by a write, then if a cache block immediately transitions to invalid, then the requesting processor only needs to incur one bus transition. Does this optimization always improve performance? The answer de-

Figure 8.5: State transition diagram for the MSI cache coherence protocol with BusUpgr.

Table 8.2: Illustration of MSI protocol operations with BusUpgr.

	Request	P1	P2	P3	Bus Request	Data Supplier
0	Initially	–	–	–	–	–
1	R1	S	–	–	BusRd	Mem
2	W1	M	–	–	**BusUpgr**	–
3	R3	S	–	S	BusRd	P1's cache
4	W3	I	–	M	**BusUpgr**	–
5	R1	S	–	S	BusRd	P3's cache
6	R3	S	–	S	–	–
7	R2	S	S	S	BusRd	Mem

pends on how often we have a situation in which all processors have a read-write sequence versus read-read sequence. If there are many read-write sequences, then we have saved one bus transaction for each of this sequence. However, if there are mostly read-read sequence, then we have unnecessarily invalidate cached copies which may result in more cache misses when a processor attempts to read from the cached copy that was just invalidated. Many synchronization operations have

a read-write sequence, especially atomic instructions, however this is not always the case, especially when more optimized synchronization operations are used. This will be discussed in more details in Chapter 9.

8.2.3 MESI Protocol with Write Back Caches

As mentioned earlier, there is a serious drawback with the MSI protocol in that each read-write sequence incurs two bus transactions regardless of whether the block is stored in only one cache or not. This penalizes programs that have little data sharing, such as sequential program (no data sharing), and highly-optimized parallel programs (little data sharing). Clearly, this is not acceptable. Hence, to remove such problem, the MESI protocol adds a state to distinguish between a block that is clean and is exclusively cached versus a block that is clean and stored at multiple caches.

In the MESI protocol, the same as the MSI protocol, processor requests to the cache include:

1. **PrRd**: processor-side request to read to a cache block
2. **PrWr**: processor-side request to write to a cache block

Bus-side requests, the same as the MSI protocol, include:

1. **BusRd**: snooped request that indicates there is a read request to a cache block made by another processor.
2. **BusRdX**: snooped request that indicates there is a *read exclusive* (write) request to a cache block made by another processor which does not already have the block.
3. **BusUpgr**: snooped request that indicates that there is a write request to a cache block that another processor already has in its cache.
4. **Flush**: snooped request that indicates that an entire cache block is written back to the main memory by another processor.
5. **FlushOpt**: snooped request that indicates that an entire cache block is posted on the bus in order to supply it to another processor. We distinguish FlushOpt from Flush because while Flush is needed for write propagation, FlushOpt is not required for correctness. It is implemented as a performance

enhancing feature that can be removed without impacting correctness. We refer to such an optional block flush as **cache-to-cache transfer**.

Each cache block has an associated state which can have one of the following values:

1. **Modified** (M): the cache block is valid in only one cache, and the value is (likely) different than the one in the main memory. This state has the same meaning as the dirty state in a write back cache for a single processor system, except that now it also implies exclusive ownership.

2. **Exclusive** (E): the cache block is valid, clean, and *only resides in one cache*.

3. **Shared** (S): the cache block is valid, clean, but may reside in multiple caches.

4. **Invalid** (I): the cache block is invalid.

One challenge in implementing the MESI protocol is that on a read miss, how does the coherence controller know whether to load the block into the cache in the exclusive or shared state? It depends on whether there are other cached copies. If there are no other copies, the block should start out in the exclusive state. Otherwise, it should start out in the shared state. How can we check whether there are existing copies in other caches? To achieve that, a new bus line can be added, which we will refer to as COPIES-EXIST or "C" bus line. The bus line has a high value when there is at least one cache that asserts it, and thus has a low value only when there are no cached copies.

We assume that the caches use *write allocate* and *write invalidate* cache coherence policy. The finite state machine corresponding to the MESI coherence protocol for write-back caches is shown in Figure 8.6. In the figure, the response to processor-side requests is shown on the top part, while the response to the snooper-side requests is shown on the bottom part.

As before, the *I* state represents two cases: a case in which the block is not cached, or when the block is cached but its state is invalid. Let us consider the top part of the figure that shows a reaction to a processor read or write request. First, consider when the cache block is in "I" (invalid) state. When there is a processor read request, it suffers a cache miss. To load the data into the cache, a BusRd is posted on the bus, and the memory controller responds to the BusRd by fetching the block from the main memory. Other snoopers will snoop the request

Figure 8.6: State transition diagram for the MESI coherence protocol.

and check their caches to determine if any of them has a copy. If a copy is found, the cache asserts the COPIES-EXIST bus line (indicated as "C" in the figure). In that case, the fetched block is placed in the requestor's cache in the shared state. If, on the other hand, the COPIES-EXIST bus line is not asserted (indicated as the " !C" in the figure), the fetched block is placed in the requestor's cache in the exclusive state. When there is a processor write request, the cache must allocate a

→ Block fetched from memory only.

valid copy of the cache block, and to do that, it posts a BusRdX request on the bus. Other caches will respond by invalidating their cached copies, while the memory responds by supplying the requested block. When the requestor gets the block, it is placed in the cache in the "M" or Modified state.

Suppose now the cache already has the block in the exclusive state. Any read to the block is a cache hit and proceeds without generating a bus transaction. A write to the block, in contrast to the MSI protocol which has no exclusive state, does not generate a bus transaction because a block in the exclusive state implies that it is the only cached copy in the system. So the write can proceed after the state transitions to modified.

Suppose now the cache already has the block in the shared state. On a processor read, the block is found in the cache and data is returned to the processor. This does not incur a bus transaction since it is a cache hit, and the state remains unchanged. On the other hand, on a processor write, there may be other cached copies that need to be invalidated, so a BusUpgr is posted on the bus, and the state transitions to modified. The memory controller ignores a BusUpgr because it does not need to fetch the block from the main memory and supplies it to the requesting processor.

If the cache block is present in the cache in the Modified state, reads or writes by the processor do not change the state, and no bus transaction is generated since it can be sure that no other cached copies exist.

Now let us look at how the finite state machine reacts to snooped bus transactions. If the cache does not have the block or the block is in Invalid state, any snooped BusRd or BusRdX/BusUpgr should not affect it, so they are ignored.

If the cache block is in an exclusive state, when a BusRd transaction is snooped, that means another processor suffered a read miss and is trying to fetch the block. Since the result is that the block is stored in more than one cache, the state must transition into the shared state. Furthermore, the Illinois MESI protocol proposed a **cache-to-cache transfer** [Papamarcos and Patel, 1984], in which a valid and clean block can be supplied by a cache that has a copy rather than by just the main memory. Such a cache-to-cache transfer can shorten the read miss latency if the time to transfer the block from another cache is faster than the time for the memory controller to fetch the block from the main memory. This is in general true in a bus-based system in which the coherence is maintained at the lowest level on chip cache. However, this assumption cannot be assumed true in all scenarios. For example, in a multicore architecture in which coherence is maintained at the level

of L2 caches, and there is an on-chip shared L3 cache, it may be faster to fetch the missed block from the L3 cache rather than from another L2 cache.

If a BusRdX is snooped, it indicates that another processor wants to write to the block. In this case, the local cached copy is flushed and then invalidated. The cache-to-cache transfers labeled as FlushOpt are optional, and are performed as a performance enhancement rather than for satisfying correctness requirements.

If the cache block is in a shared state, when a BusRd transaction is snooped, that means another processor suffers a read miss and tries to fetch the block. Therefore, the state remains shared, and one of the caches supplies the block through FlushOpt. If a BusRdX is snooped, one of the copy is flushed to the bus through the FlushOpt, and the local cached copy is invalidated. Note that the main memory does not need to be updated when a FlushOpt is seen on the bus. This is because the cache block involved in the FlushOpt is clean, i.e., the copy in the main memory is also fresh and valid.

If the cache block is in the Modified state, the copy in the cache is the only valid copy in the entire system (no other cached copies exist and the value in the main memory is stale). Therefore, when a BusRd transaction is snooped, the block must be flushed for ensuring write propagation, and the state transitions to shared. The cache that puts the BusRd request must pick up the flushed block as the reply to its request and place it in its cache in a shared state. In addition, the main memory must also snoop the flush and update the main memory.

Write propagation is ensured through two mechanisms. First, by invalidating other copies on a write to a cache block, other caches are forced to suffer a cache miss to reload the block. Secondly, by flushing a modified block upon snooping a BusRd or BusRdX request, the flushed block can be picked up by the requesting processor/cache, ensuring that it gets the latest value of the block.

Compared to the MSI protocol, the MESI protocol eliminates the need to incur two bus transactions on a read-then-write sequence coming from a processor. This avoids penalizing sequential programs and highly-tuned parallel programs. However, the MESI protocol incurs additional complexity compared to the MSI protocol. First, there is the addition of the COPIES-EXIST bus line and its associated logic. Secondly, the cache-to-cache transfer in the form of FlushOpt incurs additional complexity in the coherence controller. This is especially the case when the FlushOpt is performed when a cache block has a shared state. There are potentially multiple caches that have the block in the shared state and all of them read the cache block from their caches and attempt to perform the FlushOpt. One of

Table 8.3: Illustration of MESI protocol operations.

	Request	P1	P2	P3	Bus Request	Data Supplier
0	Initially	–	–	–	–	–
1	R1	**E**	–	–	BusRd	Mem
2	W1	M	–	–	–	–
3	R3	S	–	S	BusRd	P1's cache
4	W3	I	–	M	BusUpgr	–
5	R1	S	–	S	BusRd	P3's cache
6	R3	S	–	S	–	–
7	R2	S	S	S	BusRd	**P1/P3's cache**

them will get to the bus faster than others and flushes its block on the bus. Other coherence controllers must snoop this, realize that somebody else has flushed the same block, and cancel their plans to flush the block. The cost of this mechanism is that potentially many cache controllers have performed useless work (and waste power) by reading the cache, try to obtain the bus access, and cancel it when someone supplies the block earlier than they can.

To illustrate how MESI works, consider the example in Table 8.3 which shows the same sequence of requests made by various processors as in Table 8.1. The example in the table assumes that initially the cache is empty. The difference between MESI and MSI are highlighted in bold fonts. In the example, we assume that BusUpgr is used when a processor write request is made to a block that is already cached, whereas BusRdX is used when the block is not yet cached.

On a read from Processor 1, a BusRd is posted, the main memory responds with the block, and Processor 1 gets the block, stores it in its cache with an exclusive state. On a write from Processor 1, in contrast to MSI, the cache state transitions to modified without incurring a bus transaction, since we know that in the exclusive state, no other cached copies exist.

When Processor 3 makes a read request, a BusRd is posted on the bus. Processor 1's snooper picks it up, checks its cache tag, and finds that it has the block in a modified state. This means that Processor 1 has the latest (and the only valid) copy, and flushes it in response to the snooped request. The block state transitions

to shared after that. In the mean time, the memory controller also attempts to fetch the block from the main memory because it does not know if eventually a cache will supply the data or not. Processor 3 snoops the flush, and by matching the address of the block being flushed with its outstanding read transaction, it knows that the flushed block should be treated as the reply to its read request. So the block is picked up, stored in its cache in the shared state. The main memory which has been trying to fetch the block from memory will also snoop the flushed block, pick it up, cancel its memory fetch, and overwrite the stale copy of the block in memory.

Next, Processor 3 has a write request. It posts a BusUpgr on the bus to invalidate copies in other caches. Processor 1's coherence controller responds by invalidating its copy. Processor 3's cache block state transitions to modified.

When Processor 1 attempts to read the block, it suffers a cache miss as a result of the earlier invalidation that it received. Processor 1 posts a BusRd and Processor 3 responds by flushing its cache block and transitions its state to shared. The flushed block also updates the main memory copy, so as a result the cache blocks are now clean.

When Processor 3 reads the block, it finds it in state shared in its cache. Since it has a valid copy, it has a cache hit and does not generate a bus request.

Finally, Processor 2 attempts to read the block and posts a BusRd on the bus. Unlike MSI that does not employ cache-to-cache transfer, in MESI both Processor 1 and Processor 3's cache controllers attempt to supply the block by flushing it on the cache through FlushOpt. One of them wins and supplies the block, and Processor 2 will pick up the block. The memory controller cancels its fetch upon snooping the FlushOpt block.

While the MESI protocol has improved various performance aspects of the MSI protocol, it still suffers from a remaining problem that is potentially quite serious. When a cache block is read and written successively by multiple processors, each of the read incurs an intervention that requires the owner to flush the cache block. While the flushed block must be picked up by the requestor as a means to ensure write propagation, the flushed block updating the copy in the main memory is not a correctness requirement for write propagation. Unfortunately, the definition of the shared state is that the cache block is clean, in that the value of the block is the same with that in the main memory. Therefore, in order to preserve the meaning of the shared block, the main memory has no choice but to update its copy. This is referred to as **clean sharing** in that when a block is

shared by multiple caches, it has to be clean. Note also that clean sharing implies that evicting a shared cache block can be performed silently, i.e. the block is simply discarded. Unfortunately, by keeping clean sharing, the main memory is updated too many times. In some systems, the bandwidth to the main memory is already restricted, so updating the main memory on each cache flush uses excessive amount of bandwidth. For example, if multiple cores in a multicore architecture maintain coherence at the L2 cache level, the L2 caches can communicate with each other using an on-chip interconnection, but updating the main memory must be performed by going off-chip. Off-chip bandwidth is severely restricted in a multicore architecture because of the limited availability of pins and slow off-chip interconnection. Thus, it will be nice if a cache flush does not need to update the main memory, by allowing dirty block to be shared by multiple caches. Supporting **dirty sharing** can be provided with an additional state in the MOESI protocol which is described in the next section.

8.2.4 MOESI Protocol with Write Back Caches

As mentioned earlier, the bandwidth to the main memory can be reduced by allowing dirty sharing. The MOESI protocol allows dirty sharing. The MESI protocol is used by Intel processors such as the Xeon processor while the MOESI protocol is used by processors such as the AMD Opteron [AMD, 2005]. In the MOESI protocol, the same as the MSI protocol, processor requests to the cache include:

1. **PrRd**: processor-side request to read to a cache block
2. **PrWr**: processor-side request to write to a cache block

Bus-side requests include:

1. **BusRd**: snooped request that indicates there is a read request to a cache block made by another processor.
2. **BusRdX**: snooped request that indicates there is a *read exclusive* (write) request to a cache block made by another processor which does not already have the block.
3. **BusUpgr**: snooped request that indicates that there is a write request to a cache block that another processor already has in its cache.
4. **Flush**: snooped request that indicates that an entire cache block is placed on the bus by a processor to facilitate a transfer to another processor's cache.

5. **FlushOpt**: snooped request that indicates that an entire cache block is posted on the bus in order to supply it to another processor. We refer to it as FlushOpt because unlike Flush which is needed for write propagation correctness, FlushOpt is implemented as a performance enhancing feature that can be removed without impacting correctness.

6. **FlushWB**: snooped request that indicates that an entire cache block is written back to the main memory by another processor, and it is not meant as a transfer from one cache to another.

Each cache block has an associated state which can have one of the following values:

1. **Modified** (M): the cache block is valid in only one cache, and the value is (likely) different than the one in the main memory. This state has the same meaning as the dirty state in a write back cache for a single system, except that now it also implies exclusive ownership.

2. **Exclusive** (E): the cache block is valid, clean, and *only resides in one cache*.

3. **Owned** (O): the cache block is valid, possibly dirty, and may reside in multiple caches. However, when there are multiple cached copies, there can only be one cache that has the block in owned state, other caches should have the block in state shared.

4. **Shared** (S): the cache block is valid, possibly dirty, but may reside in multiple caches.

5. **Invalid** (I): the cache block is invalid.

The idea behind the owned state is that when a cache block is shared across caches, its value is allowed to differ from that in the main memory. One cache is assigned as the owner and caches the block in state "O" or owned, while others cache it in the shared state. The existence of the owner simplifies how data is supplied in a cache-to-cache transfer. For example, when a BusRd is snooped, we can let the owner to provide data through FlushOpt, while other controllers take no action. The main memory does not need to pick up a Flush or FlushOpt to update the block in main memory. In addition, we can also assign the owner to be responsible for writing back the block to the main memory when the block is evicted. Hence, when a cache block in the shared state is evicted, regardless of

whether it is clean or dirty, it can be discarded. Only when the cache block that is evicted is in the owned state, it is written back to the memory to update it. To indicate that a block in the owned state is evicted and needs to update the main memory, a different bus request type is needed, which we refer to as FlushWB.

Who should be the owner of a cache block? To answer this question, consider that when there is dirty sharing, a block in the shared state can be replaced silently, but a block in the owned state must be written back to the main memory. Bus bandwidth can be conserved if the frequency of write backs is minimized. To reduce the frequency of write backs, the cache that will hold the block the longest should be selected as the owner. Although predicting which cache will hold a particular shared block the longest is difficult, good heuristics can often help. Since applications tend to exhibit temporal locality, a good heuristic for predicting such a cache is selecting the cache that last wrote to or read from the block as the owner. However, reads to valid blocks do not incur any bus transactions, so it is inconvenient (and expensive) to change the ownership when a processor reads from a shared block in the cache. Thus, one heuristic that can be used (implemented in AMD Opteron systems) is to select the last cache that wrote to the block as the owner. More specifically, the cache that has the block in the Modified state, when it receives an intervention request, downgrades the block state to owned – in effect becoming the owner of the block.

We assume that the caches use *write allocate* and *write invalidate* cache coherence policy. The finite state machine corresponding to the MOESI coherence protocol for write-back caches is shown in Figure 8.7. In the figure, the response to processor-side requests is shown on the top part, while the response to the snooper-side requests is shown on the bottom part.

As before, the *I* state represents two cases: a case in which the block is not cached, or when the block is cached but its state is invalid. Let us consider the top part of the figure that shows a reaction to a processor read or write request. First, consider when the cache block is in "I" (invalid) state. When there is a processor read request, it suffers a cache miss. To load the data into the cache, a BusRd is posted on the bus, and the memory controller responds to the BusRd by fetching the block from the main memory. Other snoopers will snoop the request and check their caches to determine if any of them has a copy. If a copy is found, the cache asserts the COPIES-EXIST bus line (indicated as "C" in the figure). In that case, the fetched block is placed in the requestor's cache in the shared state. If, on the other hand, the COPIES-EXIST bus line is not asserted (indicated as

Figure 8.7: State transition diagram for the MOESI coherence protocol.

the " !C" in the figure), the fetched block is placed in the requestor's cache in the exclusive state. When there is a processor write request, the cache must allocate a valid copy of the cache block, and to do that, it posts a BusRdX request on the bus. Other caches will respond by invalidating their cached copies, while the memory responds by supplying the requested block. When the requestor gets the block, it is placed in the cache in the "M" or Modified state.

Suppose now the cache already has the block in the exclusive state. Any read to the block is a cache hit and proceeds without generating a bus transaction. A write to the block, as in the MESI protocol, does not generate a bus transaction because a block in the exclusive state implies that it is the only cached copy in the system. So the write can proceed after the state transitions to modified.

Suppose now the cache already has the block in the shared state. On a processor read, the block is found in the cache and data is returned to the processor. This does not incur a bus transaction since it is a cache hit, and the state remains unchanged. On the other hand, on a processor write, there may be other cached copies that need to be invalidated, so a BusUpgr is posted on the bus, and the state transitions to modified.

If the cache block is present in the cache in the Modified state, reads or writes by the processor do not change the state, and no bus transaction is generated since it can be sure that no other cached copies exist, since through an earlier invalidation, we have made sure that only one modified copy can exist in the system.

If the cache block is present in the cache in an owned state, this means that it is dirty and the block is shared in other caches. A processor read can just fetch the value from the cache. A processor write must invalidate other cached copies by posting a BusUpgr transaction.

Now let us look at how the finite state machine reacts to snooped bus transactions. If the cache does not have the block or the block is in Invalid state, any snooped BusRd or BusRdX/BusUpgr should not affect it, so they are ignored.

If the cache block is in an exclusive state, when a BusRd request is snooped, the block is flushed to the cache using FlushOpt, and the state transitions to shared. When a BusRdX request is snooped, the block is flushed using FlushOpt and the state transitions to invalid.

If the cache block is in a shared state, when a BusRd transaction is snooped, that means another processor suffered a read miss and is trying to fetch the block. Therefore, the state remains shared, and since only the owner is responsible for flushing the block, the local cache does not flush the block. Note that there may be an owner (in case of dirty sharing) or there may not be an owner (clean sharing). In the case that there is no owner (clean sharing), the main memory can supply the block, although using a MESI-like FlushOpt is also possible here. If a BusRdX or BusUpgr is snooped, the block's state transitions to invalid. Again, the owner (if there is one) is responsible for flushing the block, so, in contrast to MESI, the non-owner caches do not need to flush their block copies.

If the cache block is in the Modified state, the copy in the cache is the only valid copy in the entire system (no other cached copies exist and the value in the main memory is stale). Therefore, when a BusRd transaction is snooped, the block must be flushed for ensuring write propagation, and the state transitions to owned. The reason for transitioning to the owned state is based on a heuristic that has been discussed earlier. Note that by transitioning to the owned state, the local cache has become the supplier of the block and is responsible for flushing it when required.

If the cache block is in the owned state, it indicates that there is dirty sharing, with the local cache being responsible as the supplier of the block. Hence, when a BusRd is snooped, it flushes the block and remains in the owned state (i.e., remains the owner). If a BusRdX is snooped, it supplies the block by flushing it, and transitions into the Invalid state. If a BusUpgr is snooped, it transitions into the Invalid state without flushing the block. Note that because or dirty sharing, the flushes from this state is a correctness requirement, rather than a performance enhancement because potentially, nobody else in the system has a valid copy of the block (some sharers which obtain the block from the owner may have a valid copy, but there is no guarantee that there are sharers).

When the block in the owned state is evicted from the cache, the ownership disappears since other caches have the copy in the shared state. Hence, at this point dirty sharing must be *converted* into clean sharing. The owner is responsible for flushing the block to the memory so that the memory can update its copy. This is achieved by posting a FlushWB request on the bus. In contrast to Flush or FlushOpt requests that are ignored by the memory controller, a FlushWB request is picked up by the memory controller to update the value in the main memory.

Similar to MSI and MESI, write propagation is ensured through two mechanisms. First, by invalidating other copies on a write to a cache block, other caches are forced to suffer a cache miss to reload the block. Secondly, with dirty sharing, a cache always acts as an owner, and flushes the block when it snoops a BusRd or BusRdX, ensuring that the latest value of the block is passed on. With clean sharing, the memory supplies the latest value of a block.

Compared to the MESI protocol, the MOESI protocol does not reduce the bandwidth usage on the bus, but it does reduce the bandwidth use to the main memory. There are cases in which this introduces benefits. For example, in the AMD K7, two processor chips are connected to a common memory controller, and the memory controller connects to DRAMs. The interconnection between

Table 8.4: Illustration of MOESI protocol operations. * indicates dirty sharing, in which the main memory is not updated when a block is flushed on the bus.

	Request	P1	P2	P3	Bus Request	Data Supplier
0	Initially	–	–	–	–	–
1	R1	E	–	–	BusRd	Mem
2	W1	M	–	–	–	–
3	R3	**O**	–	S	BusRd	P1's cache*
4	W3	I	–	M	BusUpgr	–
5	R1	S	–	**O**	BusRd	P3's cache*
6	R3	S	–	**O**	–	–
7	R2	S	S	**O**	BusRd	**P3's cache**

one processor and the memory controller is approximately equal to the bandwidth available between the memory controller and the main memory. However, when two processors are interconnected to the main memory controller, the combined bandwidth between processors is higher than the bandwidth to the main memory. Hence, MOESI was selected for the obvious benefit that it reduces the bandwidth requirement to the main memory. Another example in which reduced memory bandwidth may be useful is in a multicore architecture in which each core has a private last level cache, and the private caches are kept coherent. The caches can communicate using on-chip bandwidth, but the bandwidth to the main memory is limited by off-chip pin bandwidth. Since on-chip bandwidth is more abundant than off-chip bandwidth to the main memory, MOESI can reduce the off-chip bandwidth requirement.

To illustrate how MOESI works, consider the example in Table 8.4 which shows the same sequence of requests made by various processors as in Table 8.1. The example in the table assumes that initially the cache is empty. The difference between MOESI and MESI are highlighted in bold fonts.

On a read from Processor 1, a BusRd is posted, the main memory responds with the block, Processor 1 gets the block and stores it in its cache with an exclusive state. On a write from Processor 1, the cache state transitions to modified

without incurring a bus transaction, since no other cached copies exist.

When Processor 3 makes a read request, a BusRd is posted on the bus. Processor 1's snooper picks it up, checks its cache tag, and finds that it has the block in a modified state. Processor 1 downgrades its cache state to owned, supplies the block through Flush on the bus. Processor 3 picks up the flushed block and places it its cache in the shared state. In the mean time, the memory controller ignores the flush that it snoops on the bus. Only if an owned block is written back, the memory controller needs to pick it up to update the main memory. Note that to distinguish an owner's flush due to natural block replacement versus that due to supplying the block to a remote requestor, another bus transaction needs to be used.

Next, Processor 3 has a write request. It posts a BusUpgr on the bus to invalidate copies in other caches. Processor 1's coherence controller responds by invalidating its copy. Processor 3's cache block state transitions to modified.

When Processor 1 attempts to read the block, it suffers a cache miss as a result of the earlier invalidation that it received. Processor 1 posts a BusRd and Processor 3 responds by flushing its cache block and transitions its state to owned. The flushed block does not update the main memory, allowing dirty sharing.

When Processor 3 reads the block, it finds it in state shared in its cache. Since it has a valid copy, it has a cache hit and does not generate a bus request.

Finally, Processor 2 attempts to read the block and posts a BusRd on the bus. Unlike MESI in which both Processor 1 and 3 try to flush the block, in MOESI the owner (Processor 3) flushes the block, while others (Processor 1) knows that it does not need to react.

8.2.5 Update-Based Protocol with Write Back Caches

One of the drawback of an invalidate-based protocol is that it incurs a high number of coherence misses. Each read to a block that has been invalidated incurs a cache miss, and the latency to serve the miss can be quite high. In this section, we will look at an update-based protocol that relies on directly updating cached values to achieve write propagation compared to using invalidation and subsequent misses to ensure write propagation. The update protocol that will be discussed is called the Dragon protocol [McCreight, 1984]. The protocol assumes the following requests from the processor and bus transactions.

1. **PrRd**: processor-side request to read to a cache block that already resides in the cache.

2. **PrRdMiss**: processor-side request to read to a cache block that *does not* already reside in the cache.

3. **PrWr**: processor-side request to write to a cache block that already resides in the cache.

4. **PrWrMiss**: processor-side request to write to a cache block that *does not* already reside in the cache.

Bus-side requests include:

1. **BusRd**: snooped request that indicates there is a read request to a cache block made by another processor.

2. **Flush**: snooped request that indicates that an entire cache block is placed on the bus by another processor.

3. **BusUpd**: snooped request that indicates a write to a word results in propagating the written value on the bus. Only a word of data is posted on the bus, rather than an entire cache block.

Each cache block has an associated state which can have one of the following values:

1. **Modified** (M): the cache block is valid in only one cache, and the value is (likely) different than the one in the main memory. This state implies exclusive ownership.

2. **Exclusive** (E): the cache block is valid, clean, and *only resides in one cache*.

3. **Shared Modified** (Sm): the cache block is valid, possibly dirty, and may reside in multiple caches. However, when there are multiple cached copies, there can only be one cache that has the block in shared modified state, other caches should have the block in state shared clean. This state is similar to the owned state in MOESI that allows dirty sharing.

4. **Shared Clean** (Sc): the cache block is valid, possibly not clean, but may reside in multiple caches. This state is similar to the shared state in MOESI.

Similar to MOESI, the Dragon protocol allows dirty sharing in which the owner is assigned a state shared-modified, while other caches are assigned the

Figure 8.8: State transition diagram for the Dragon update coherence protocol.

shared-clean state. Note that since a cache block may be updated but never invalidated, there is no Invalid state. If a block is cached, it has a valid value, although the value may be dirty.

We assume that the caches use *write allocate* and *write update* cache coherence policy. The finite state machine corresponding to the Dragon coherence protocol for write-back caches is shown in Figure 8.8. In the figure, the response

to processor-side requests is shown on the top part, while the response to the snooper-side requests is shown on the bottom part. Arrows that do not originate from a state represent the case in which prior to the transition, the block is not present in the cache.

Note that there is no Invalid state. Hence, an arrow that goes from nowhere to a state represents a newly loaded block. Let us consider the top part of the figure that shows a reaction to a processor read or write request. First, consider when a read miss to a block occurs. BusRd is posted on the bus, and if there is no copy of the block in any cache (!C), then the block is placed in the cache in the exclusive state. If there is already a copy in some caches (C), then the block is placed in the cache in the shared-clean state. Now consider when a write miss to a block occurs. If there is no copy of the block in any cache (!C), the block is placed in the cache in the Modified state. If there is already a copy of the block in some caches (C), then the block is placed in the cache in the state shared-modified. After that, the write is performed and BusUpd is posted on the bus in order to update other copies.

If a block is already cached in the exclusive state, a processor read does not change its state and does not incur a bus transaction. A processor write changes it to the Modified state without incurring a bus transaction.

If a block is already cached in the shared-clean, a processor read does not change its state and does not incur a bus transaction. A processor write changes it to the shared-modified because now it is the new owner of the block. In addition, since this is a processor write, it must propagate the written value by posting a BusUpd on the bus. Since it goes to the bus, the COPIES-EXIST line can also be checked, so that if there is no longer another cached copy, the state can transition to modified instead of shared-modified. Note that this is necessary for good performance because in the Modified state, processor writes no longer incur bus update transactions, preserving bus bandwidth.

If a block is already cached in the shared-modified, a processor read does not change its state and does not incur a bus transaction. A processor write either keeps the state unchanged at shared-modified because it remains the owner of the block, or transitions to modified if it becomes the only cache that has a copy. In either case, BusUpd is posted in order to update other cached copies.

If a block is already cached in the Modified state, a processor write or read does not change the state, and no bus transaction is incurred because we know no other cached copies exist.

Now let us look at how the finite state machine reacts to snooped bus transactions. If the cache does not have the block (not shown in any state), any snooped transaction is ignored.

If the cache block is in an exclusive state, when a BusRd request is snooped, the block is supplied from the main memory, so the state transitions to shared-clean without incurring further bus transactions. A BusUpd cannot occur in this state because there are no other caches that have the block; nobody else can update the block.

If the cache block is in a shared-clean state, when a BusRd transaction is snooped, that means another processor suffers a read miss and tries to fetch the block. Since the block is supplied by the main memory or the owner of the block, no action is taken. If there is dirty sharing, the owner is responsible for flushing the block. If a BusUpd is snooped, the update for the word is picked up and used to update the currently cached block.

If the cache block is in the shared-modified state, it means that the local cache is the owner of the block. Hence, if there is a BusRd, the block is flushed on the bus so that it can be picked up by the requestor. If there is a BusUpd, a requestor is trying to write to the block, hence the requestor will be the new owner of the block, so the ownership is relinquished by transitioning into the shared-clean state, and the update is merged into the cache block.

If the cache block is in the Modified state, the copy in the cache is the only valid copy in the entire system. Therefore, when a BusRd transaction is snooped, the block must be flushed for ensuring write propagation, and the state transitions to shared-modified. A BusUpd is impossible to be snooped at this state because no other cached copies exist.

Unlike MSI, MESI, or MOESI, in the Dragon update protocol, write propagation is ensured by directly updating all other copies on a write. The BusUpd can propagate the needed bytes or words, so it may occupy the bus for a shorter time compared to a regular Flush.

To illustrate how the Dragon protocol works, consider the example in Table 8.5 which shows the same sequence of requests made by various processors as in Table 8.1. The example in the table assumes that initially the cache is empty.

On a read from Processor 1, a BusRd is posted, the main memory responds with the block, and Processor 1 gets the block, stores it in its cache with an exclusive state. On a write from Processor 1, the cache state transitions to modified without incurring a bus transaction, since no other cached copies exist.

Table 8.5: Illustration of the Dragon protocol operations. * indicates dirty sharing, in which the main memory is not updated when a block is flushed on the bus.

	Request	P1	P2	P3	Bus Request	Data Supplier
0	Initially	–	–	–	–	–
1	R1	E	–	–	BusRd	Mem
2	W1	M	–	–	–	–
3	R3	Sm	–	Sc	BusRd	P1's cache*
4	W3	Sc	–	Sm	BusUpd	–*
5	R1	Sc	–	Sm	–	–*
6	R3	Sc	–	Sm	–	–*
7	R2	Sc	Sc	Sm	BusRd	P3's cache*

When Processor 3 makes a read request, a BusRd is posted on the bus. Processor 1's snooper picks it up, checks its cache tag, and finds that it has the block in a modified state. Processor 1 downgrades its cache state to shared-modified, supplies the block through Flush on the bus. Processor 3 picks up the flushed block and places it its cache in the shared-clean state. In the mean time, the memory controller ignores the flush that it snoops on the bus, allowing dirty sharing. Only if a shared-modified block is written back, the memory controller needs to pick it up to update the main memory. Note that to distinguish this with a regular flush, a different bus transaction needs to be used.

Next, Processor 3 has a write request. It posts a BusUpd on the bus to update other cached copies. Processor 1's coherence controller responds by transitioning to the shared-clean state, relinquishing the ownership of the block, and updates its own cached copy. Processor 3 becomes the new owner and the state of the block becomes shared-modified.

When Processor 1 attempts to read the block, it has a cache hit since the block is valid because it has been updated earlier. The state is unchanged and the no bus transaction is involved.

When Processor 3 reads the block, it also finds a valid copy in its cache. So it reads the copy and no bus transaction is generated.

Finally, Processor 2 attempts to read the block and posts a BusRd on the bus. Similar to MOESI, the owner (Processor 3) supplies the block through a flush which is picked up by Processor 2, which can then cache the block in the shared-clean state.

There are several low-level protocol choices. One of them is that one can remove dirty sharing by requiring that any BusUpd should be snooped by the memory controller to update the block in the main memory. If this is implemented, then the state shared-modified can be eliminated since we now only have clean sharing. However, clean sharing increases the frequency of updating the lower level memory hierarchy component.

8.3 Impact of Cache Design on Cache Coherence Performance

Let us now revisit different types of cache misses: compulsory, conflict, capacity, coherence, and system-related (Refer to Chapter 6 for more discussion). One of the types of cache misses is coherence misses. Let us define coherence misses more explicitly: *coherence misses are misses that occur due to accessing cache blocks that have been invalidated due to coherence events.*

Clearly, coherence misses only affect write invalidate protocols since an update protocol does not invalidate a cache block, which does not lead to subsequent misses on the block. This is not to say that an update protocol does not incur a performance problem in caches. For example, an update protocol may keep updating a block in another cache that is no longer used. In this case the block stays in the cache longer than needed, and this reduces the number of useful cache blocks that can be stored. Hence, while using an update based protocol does not incur coherence misses, it incurs a hidden cost of increased number of conflict and capacity misses.

There are two types of coherence misses: **true sharing** and **false sharing** misses. A true sharing miss occurs due to multiple threads sharing the same variable (same byte/word), and they invalidate one another. A false sharing miss occurs due to multiple threads sharing different variables in different bytes or words that reside in a single cache block. Because the coherence granularity is a cache block, the block is invalidated by one another while they have no actual sharing.

How do cache parameters affect the number of coherence misses? Table 8.6 shows the relationship. First, larger cache block sizes typically reduce the number

of any type of cache misses if there is spatial locality on the block due to their *prefetching effect*, i.e. it takes fewer misses to fetch the same number of bytes. The exception is the number of false sharing misses. The number of true sharing misses tends to decrease as the prefetching effect of large cache block sizes allows multiple bytes that may have true sharing to be fetched together. However, the number of false sharing misses increases with a larger cache block size because false sharing is more likely to occur as more bytes are grouped in a cache block.

Table 8.6: Impact of various cache parameters on coherence misses.

Parameters	True Sharing	False Sharing
Larger cache size	increased	increased
Larger block size	decreased	increased
Larger associativity	unclear	unclear

The impact of cache size on coherence misses is less subtle. A larger cache size allows more blocks to be cached, and more of them will likely suffer from true sharing and false sharing misses. With a small cache size, these blocks would not be cached, so they incur capacity misses rather than coherence misses. Finally, the impact of associativity on coherence misses is unclear.

8.4 Performance and Other Practical Issues

8.4.1 Prefetching and Coherence Misses

Employing prefetching carries some risks in multiprocessor systems. Prefetching of a block generates an intervention through a BusRd transaction on the bus. If currently there is a cache that holds the block in an exclusive or Modified state, then it must supply the block and downgrade its state to shared. Unfortunately, doing this incurs three risks. First, there is the usual risk that the prefetched block evicts a more immediately useful block in the cache. This risk is inherent in prefetching as long as the prefetched blocks are placed in the cache and not in a separate buffer, and also exists in single processor prefetching. The second risk is that the prefetched block may be stolen before it is used by the processor. Such risk increases when the prefetching acts aggressively in bringing in blocks into the cache early. For example, if an invalidation is received, the prefetched block must

be invalidated and refetched later. When the processor needs to access it, it suffers from a cache miss despite the fact that prefetching was accurate in identifying the block that the processor will be accessing. The third risk occurs when the block is supplied from a cache that held the block in the exclusive or Modified state. If the supplier processor still needs to write to the block, its write will be delayed now that it needs to send an invalidation on the bus. Consequently, prefetching, if done too early, may hurt the performance of the processor that supplies the data as well as the one that performs prefetching. Overall, prefetching in a multiprocessor system is a lot trickier than in a single processor system.

8.4.2 Multi-Level Caches

So far, we have assumed a system in which each processor has a cache. If a processor has multi-level caches, the cache coherence protocol needs to be modified in order to work on them. Consider a system where each processor has L1 caches and an L2 cache, and the coherence is maintained at the L2 cache level.

Ensuring cache coherence protocol correctness requires ensuring write propagation, even in the case of multi-level caches. Write propagation must be performed *downstream*: when a processor writes to its local L1 cache, it must be propagated not just to the L2 cache, and from there to other processors' caches. It must also be performed *upstream*: when another processor's attempt to write is snooped on the bus, the write must not only be propagated to the L2 cache, but also to the L1 cache.

Let us first consider the downstream write propagation. At the very least, a write attempt in the L1 cache must notify the L2 cache so that the L2 cache can generate an appropriate bus transaction to invalidate other processors' caches. This can be accomplished by simple changes to the L1 cache controller. Secondly, when another processor suffers a subsequent miss on the block, the latest value of the block must be supplied to the requestor through a bus flush. One relevant question here is whether the latest value of the block resides in the L1 cache or the L2 cache. If it is in the L2 cache, the L2 cache can supply the block on the bus. If it is in the L1 cache, the L2 cache must not supply the block directly. Instead, it must obtain the block from the L1 cache, before supplying it on the bus.

The write policy of the L1 cache determines how that mechanism should be implemented. If the L1 cache uses a write-through policy, then all writes at the L1 cache are always propagated down to the L2 cache. Thus, the L2 cache always has the latest version of the block and can supply a modified block on the bus

directly. If, on the other hand, the L1 cache uses a write back policy, the L2 cache does not always have the latest version of a block. Dirty blocks may be kept at the L1 cache without the knowledge of the L2 cache. To support write propagation correctly, the L2 cache must (1) know that the L1 cache has a dirty copy, and (2) ask the L1 cache to write back its cache block so that the L2 cache can pick up the latest version of the block. Therefore, the L1 cache controller must be modified to notify the L2 cache upon a write. The L2 cache also needs to keep a special cache state to keep track that a block is dirty in the L1 cache. Finally, when the snooper notifies the L2 cache controller that there is an intervention request to a block that is dirty in the L1 cache, the L2 cache controller must request the block to be written back from the L1 cache, and request the block state to be updated in the L1 cache. The write back updates the copy in the L2 cache as well and modifies the state appropriately. Then, the L2 cache supplies the block on the bus.

Let us consider the upstream write propagation. When an invalidation is received by the L2 cache, it must invalidate its block, but such invalidation must also be propagated to the L1 cache, in case the L1 cache also has a copy of the block. This is required to force a cache miss when the processor wants to access the block later on. Readers may recall that such policy is already enforced if the inclusion property between the L2 and L1 cache is enforced (refer to Section 6.2.4). Therefore, when an inclusive L2 cache is used, the upstream write propagation is automatic. However, if an inclusive L2 cache is not used, each invalidation must still be sent to the L1 cache. Note that sending an invalidation to the L1 cache on each snooped write request is expensive as the L1 cache tag must be accessed to determine if the block is stored at the L1 cache or not. The benefit of the inclusion property is that since it is guaranteed that the blocks that are stored in the L1 caches are also stored in the L2 cache, if a snooped bus transaction has verified that the block involved is not found in the L2 cache, no invalidation or intervention need to be sent to the L1 cache since the block cannot be stored in the L1 cache either.

8.4.3 Snoop Filtering

In well-optimized programs, threads do not share much data. So most of the time, most snooped bus transactions do not find the block in the local cache. However, even in that case, the snooper has incurred unnecessary work: snooped the bus transaction, and checked the cache tag to determine whether the cache has the block. Snooping each bus transaction is unavoidable in a bus-based multiproces-

sor. However, checking the cache tag on each snooped transaction is expensive and often unnecessary. First, if the coherence level is at the L2 cache level, then the L2 cache tags are accessed by both the processor as well as by the snooper. Accesses from the processor and the snooper compete and such competition may increase contention-related delay. The degree of such contention depends on the number of L2 cache ports. Hence, one way to reduce the contention delay in accessing L2 cache port is to add an additional port. However, this comes at the expense of a significant area overhead and power, which reduces the overall cache size. Alternatively, the L2 cache tag can be duplicated to eliminate contention between the processor and the coherence controller. But again, this solution comes at the cost of a significant area overhead.

One promising solution to reduce contention between the processor and the snooper is to introduce a **snoop filter**, which determines whether a snooper needs to check the L2 cache tag or not. By reducing the number snooped transactions that need to check the cache tags, contention and power consumption can be reduced, allowing a single ported L2 cache to perform well.

An example of snoop filter implementations can be found in the IBM BlueGene/P processor [Salapura et al., 2008], which is a 4-core multicore processor. Each core has private L1 and L2 caches, and an L3 cache shared by all cores. Coherence is maintained between the L2 caches. The snoop filter implements several types of filter. The first filter is a *history filter* which records blocks that have been verified not to be in the cache, i.e. recently checked against the cache tags and found to mismatch, or recently invalidated blocks. When there are bus read or read exclusive requests snooped on the bus to these blocks, the requests are ignored and the L2 cache tags are not checked. Another filter is *stream filter* which records recent stream-like cache misses made by the processor. If a snooped request on the bus matches the stream, they are further checked against the L2 cache tags. A third filter is referred to *range filter*. A range filter records address ranges that are not cached, e.g. because they are specified as non-cacheable. Snooped bus transactions to these addresses are ignored.

8.5 Exercise Problems

1. (*20 Points*) **MSI vs. MESI**.

For each of the memory reference streams given in the following, compare the cost of executing it on a bus-based machine that supports (a) MSI, (b) MESI, and (c) MOESI protocols. All the references (Read/Write) in the streams are to the same location, and the digit refers to the processor issuing the reference. Assume that all caches are initially empty, and use the following cost model: a read/write cache hit takes 1 cycle to complete; a read/write cache miss that does not involve a whole block to be transferred on the bus takes 60 cycles to complete; and a read/write cache miss that involves whole block to be transferred on the bus takes 90 cycles to complete. Assume all caches are write-back and write-allocated.

Use the following tables for the answer.

Stream 1 (MSI):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Write1						
Read1						
Write1						
Read2						
Write2						
Read2						
Write2						
Read3						
Write3						
Read3						
Write3						
TOTAL						

Stream 1 (MESI):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Write1						
Read1						
Write1						
Read2						
Write2						
Read2						
Write2						
Read3						
Write3						
Read3						
Write3						
TOTAL						

Stream 1 (MOESI):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Write1						
Read1						
Write1						
Read2						
Write2						
Read2						
Write2						
Read3						
Write3						
Read3						
Write3						
TOTAL						

Stream 2 (MSI):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Read2						
Read3						
Write1						
Write2						
Write3						
Read1						
Read2						
Read3						
Write3						
Write1						
TOTAL						

Stream 2 (MESI):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Read2						
Read3						
Write1						
Write2						
Write3						
Read1						
Read2						
Read3						
Write3						
Write1						
TOTAL						

Stream 2 (MOESI):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Read2						
Read3						
Write1						
Write2						
Write3						
Read1						
Read2						
Read3						
Write3						
Write1						
TOTAL						

Stream 3 (MSI):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Read2						
Read3						
Read3						
Write1						
Write1						
Write1						
Write1						
Write2						
Write3						
TOTAL						

Stream 3 (MESI):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Read2						
Read3						
Read3						
Write1						
Write1						
Write1						
Write1						
Write2						
Write3						
TOTAL						

Stream 3 (MOESI):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Read2						
Read3						
Read3						
Write1						
Write1						
Write1						
Write1						
Write2						
Write3						
TOTAL						

2. (*20 Points*) **Dragon Protocol**

For each of the memory reference streams given in the following, compare the cost of executing it on a bus-based machine that supports Dragon protocol. All the references (Read/Write) in the streams are to the same location, and the digit refers to the processor issuing the reference. Assume that all caches are initially empty, and use the following cost model: a read/write cache hit takes 1 cycle; a read/write cache miss which does not involve transferring a whole block on the bus takes 60 cycles to complete; and a read/write miss which involves a whole block to be transferred on the bus takes 90 cycles. Assume all caches are write-back and write-allocated.

Use the following tables for the answer.

Stream 1 (Dragon):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Write1						
Read1						
Write1						
Read2						
Write2						
Read2						
Write2						
Read3						
Write3						
Read3						
Write3						
TOTAL						

Stream 2 (Dragon):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Read2						
Read3						
Write1						
Write2						
Write3						
Read1						
Read2						
Read3						
Write3						
Write1						
TOTAL						

Stream 3 (Dragon):

Processor	P1	P2	P3	Bus Action	Data From	Cycles
Read1						
Read2						
Read3						
Read3						
Write1						
Write1						
Write1						
Write1						
Write2						
Write3						
TOTAL						

3. **MESI vs. MOESI** Explain two advantages and two disadvantages of the MESI vs. MOESI bus-based cache coherence protocols.

4. **MOESI vs. Dragon** Explain two advantages and two disadvantages of the MOESI vs. Dragon bus-based cache coherence protocols.

5. **Non-Silent Replacement** Suppose that the MESI protocol is modified so that the replacement of a clean block is no longer silent, i.e. it is broadcasted on the bus. This way a cache can detect if it becomes the only cache that holds a copy of the block, and can transition the state of the block from Shared (S) to Exclusive (E). Explain the advantages and drawbacks of this modification. Draw the cache coherence protocol diagram for the modified MESI protocol.

6. **MOESI Problem** In the MOESI protocol, if a block in the Owned (O) is evicted from the cache, the block is written back to the main memory, even though there may be other caches which hold the block in a Shared (S) state. This write back incurs an unnecessary off-chip bandwidth consumption. Propose a solution to avoid having to write back the block, or at least reduce the frequency of such write backs. Draw the cache coherence protocol diagram for your solution.

Chapter 9

Hardware Support for Synchronization

Contents

9.1 Lock Implementations 248
9.1.1 Evaluating the Performance of Lock Implementations 248
9.1.2 The Need for Atomic Instructions 249
9.1.3 Test and Set Lock 252
9.1.4 Test and Test and Set Lock 255
9.1.5 Load Linked and Store Conditional Lock 257
9.1.6 Ticket Lock 262
9.1.7 Array-Based Queuing Lock 265
9.1.8 Qualitative Comparison of Lock Implementations . . 267
9.2 Barrier Implementations 269
9.2.1 Sense-Reversal Centralized Barrier 270
9.2.2 Combining Tree Barrier 274
9.2.3 Hardware Barrier Implementation 274
9.3 Exercise Problems 277

As we have discussed in Chapter 7, hardware support for building a shared memory multiprocessor system that guarantees correct and efficient execution of parallel programs must address cache coherence, memory consistency, and support for synchronization primitives. In the previous chapter, we have discussed cache coherence support for bus-based multiprocessors.

The objective of this chapter is to discuss hardware support for implementing synchronization primitives. From the point of view of software, synchronization

primitives that are widely used are locks, barriers, and point-to-point synchronizations such as signal-wait pairs. Recall for example that locks and barriers are heavily used in DOALL parallelism and in applications with linked data structures, whereas signal-wait synchronization is essential in pipelined (e.g. DOACROSS) parallelism.

There are many ways of supporting synchronization primitives in hardware. A common practice today is to implement the lowest level synchronization primitive in the form of atomic instructions in hardware, and implement all other synchronization primitives on top of that in software. We will discuss the trade-offs of various implementations of locks and barriers. We will show that achieving fast but scalable synchronization is not trivial. Often there are trade-offs in *scalability* (how synchronization latency and bandwidth scale with a larger number of threads) versus *uncontended latency* (the latency when threads do not simultaneously try to perform the synchronization). We will also discuss for very large systems, software barriers implemented on top of an atomic instruction and lock do not give sufficient scalability. For these systems, hardware barrier implementation is common and we will discuss one example.

9.1 Lock Implementations

In this section, we will discuss hardware support for lock implementations, and in the next section, we will discuss hardware support for barrier implementations.

9.1.1 Evaluating the Performance of Lock Implementations

Before discussing various lock implementations, first we will discuss what performance criteria need to be taken into account in a lock implementation. They are:

1. **Uncontended lock-acquisition latency**: the time it takes to acquire a lock when there is no contention between threads.

2. **Traffic**: the amount of traffic generated as a function of number of threads or processors that contend for the lock. Traffic can be subdivided into three: the traffic on *lock acquisition when a lock is free*, traffic on *lock acquisition when a lock is not free*, and traffic on *lock release*.

3. **Fairness**: the degree in which threads are allowed to acquire locks with respect to one another. One criteria related to fairness is whether in the lock

implementation, it is possible for a thread to be **starved**, i.e. it is unable to acquire a lock for a long period of time even when the lock became free during that time.

4. **Storage**: the amount of storage needed as a function of number of threads. Some lock implementations require a constant storage space independent of the number of threads that share the lock, while others require a storage space that grows linearly along with the number of threads that share the lock.

9.1.2 The Need for Atomic Instructions

Recall the discussion in Chapter 7 in which we compare software versus hardware mechanisms for guaranteeing mutual exclusion. We have shown that a software mechanism (i.e. Peterson's algorithm) does not scale because the number of static instructions executed and the number of variables that need to be tested in order for a thread to check the condition for a lock acquisition increases along the number of threads. In contrast, if an **atomic instruction** that can perform a sequence of load, compare or other instructions, and a store, is available, a simple lock implementation that relies on testing only one variable is sufficient.

In current systems, most processors support an atomic instruction as (almost) the lowest level primitive on which other synchronization primitives can be built. An atomic instruction performs a sequence of read, modify, and write in an indivisible manner as one indivisible operation. Consider the lock implementation in the following code (Code 9.1).

Code 9.1 An incorrect implementation of lock/unlock functions.

```
lock: ld  R1, &lockvar    // R1 = lockvar
      bnz R1, lock        // jump to lock if R1 != 0
      st  &lockvar, #1    // lockvar = 1
      ret                 // return to caller

unlock: st  &lockvar, #0  // lockvar = 0
        ret               // return to caller
```

In order for the code to work correctly, the sequence of load, branch, and store instructions must be performed *atomically*. The term atomic implies two things. First, it implies that either *the whole sequence is executed entirely*, or

none of it appears to have executed. Second, it also implies that *only one atomic instruction from any processor can be executed at any given time*. Multiple instruction sequences from several processors will have to be serialized. Certainly, software support cannot provide any atomicity of the execution of several instructions. Thus, most processors support atomic instructions, instructions that are executed atomically with respect to other processors. Different processors may support different types of atomic instructions. Below are some of the commonly used ones:

- `test-and-set Rx, M`: read the value stored in memory location M, test the value against a constant (e.g. 0), and if they match, write the value in register Rx to the memory location M.
- `fetch-and-op M`: read the value stored in memory location M, perform `op` to it (e.g., increment, decrement, addition, subtraction), and then store the new value to the memory location M.
- `exchange Rx, M`: atomically exchange (or swap) the value in memory location M with the value in register Rx.
- `compare-and-swap Rx, Ry, M`: compare the value in memory location M with the value in register Rx. If they match, write the value in register Ry to M, and copy the value in Rx to Ry.

Two natural questions that readers may ask include: (1) how can atomicity be ensured for an atomic instruction? and (2) how can an atomic instruction be used to construct synchronization? We will discuss the answer to the first question first.

An atomic instruction essentially provides a guarantee to programmers that a sequence of operations represented by the instruction will be executed in its entirety. Using this guarantee, programmers can implement a variety of synchronization primitives they need (to be discussed later). The actual mechanism that guarantees atomicity of a sequence of operations itself must be provided correctly by the hardware.

Fortunately, the cache coherence protocol provides a foundation on which atomicity can be guaranteed. For example, when an atomic instruction is encountered, the coherence protocol knows that it must guarantee atomicity. It can do so by first obtaining an exclusive ownership of the memory location M (by invalidating all other cached copies of blocks that contain M). After the exclusive ownership is obtained, the protocol has ensured that only one processor has an

■ *Did you know?*

In the x86 instruction set, in addition to atomic instructions, regular integer instructions can also be made atomic by prefixing it with LOCK. When a memory location is accessed by an instruction that is LOCK-prefixed, the LOCK# bus line is asserted to prevent other processors to read or modify the location while the prefixed instruction executes. For example, the following instructions can be LOCK-prefixed:

- *Bit manipulation instructions such as BT, BTS, BTR, and BTC.*
- *Arithmetic instructions such as ADD, ADC, SUB, and SBB.*
- *Logical instructions such as NOT, AND, OR, and XOR.*
- *Some unary instructions such as NEG, INC, and DEC.*

Atomic instructions are also provided. They start with "X", such as XADD, and XCHG. These atomic instructions do not require the "LOCK" prefix to make them atomic.

To illustrate the use of the prefix, let us say there is a variable `counter` *that we want to increment atomically. To do that, we can prefix the INC instruction with the LOCK prefix, as shown in the following example:*

```
void __fastcall atomic_increment (volatile int* ctr)
{
    __asm {
        lock inc dword ptr [ECX]
        ret
    }
}
```

access to the block. The atomic instruction can then be executed. For the duration of the atomic instruction, the block must not be allowed to be stolen by other processors. For example, if another processor requests to read or write to the block, the block is "stolen" (e.g., it is flushed and its state is downgraded). Exposing the block before the atomic instruction is completed, breaks the atomicity of the instruction, similar to the non-atomic execution of a sequence of instructions in the naive lock implementation case (Code 9.1). Therefore, before the atomic instruction is completed, the block cannot be stolen. One way to prevent stealing of the block (on a bus-based multiprocessor) is to lock or reserve the bus until the instruction is complete. Since the bus is the serializing medium in the system, if it is locked, no other bus transactions can access the bus until it is released. A more general solution (which also applies to non-bus based multiprocessors) is not to

prevent other requests from going to the bus, but for the coherence controller of the processor that executes the atomic instruction to defer responding to all requests to the block until the atomic instruction is completed, or negatively acknowledge the requests so that the requestors can retry the requests in the future. Overall, through exclusive ownership of a block and preventing it from being stolen until the atomic instruction is complete, atomicity is ensured. A simpler solution to provide atomicity will be discussed later in Section 9.1.5.

9.1.3 Test and Set Lock

How is the atomic instruction used to implement a lock? It depends on the type of the atomic instruction. The following code illustrates a lock implementation using a test-and-set instruction (Code 9.2). The first instruction in the lock acquisition attempt is the atomic test-and-set instruction, which performs the following steps atomically: it reads from the memory location where `lockvar` is located (using an exclusive read such as BusRdX or BusUpgr), into register R1, compares R1 with zero. If it finds that R1 is 0, writes "1" to `lockvar` (indicating a successful lock acquisition). If R1 is not 0, it does not write "1" to `lockvar` (indicating a failed lock acquisition). The second instruction branches back to the label `lock` if R1 is not zero, so that the lock acquisition can be retried. If the value of R1 is zero, then we know that when the branch instruction is reached, the test-and-set instruction is successful due to its atomicity. Note that releasing the lock is simply performed by storing a zero to `lockvar` without using an atomic instruction. This works because there is only one thread in the critical section, so there is only one thread that can release the lock. Hence, no races can occur.

Code 9.2 An implementation of test-and-set lock.

```
lock: t&s R1, &lockvar // R1 = lockvar
                       // if (R1==0) lockvar=1
      bnz R1, lock     // jump to lock if R1 != 0
      ret              // return to caller

unlock: st  &lockvar, #0 // lockvar = 0
        ret              // return to caller
```

Let us see how the execution of two threads now allows only one thread to enter the critical section. Figure 9.1 illustrates the case. Let us assume that thread 0 executes the test-and-set instruction slightly earlier than P1. If the memory block

where `lockvar` is stored initially has a value of 0, then after the execution of the test-and-set instruction, it holds a value of 1. Note that the atomicity of test-and-set serializes the execution of other test-and-set instructions. Hence, while test-and-set from thread 0 is ongoing, test-and-set from thread 1 is not allowed to proceed. When the test-and-set instruction of thread 1 executes, it sees the value of 1 in the memory location where `lockvar` is located. Hence, its lock acquisition fails, and thread 1 goes back to retry the test-and-set, on and on until it finally succeeds when the lock is released by thread 0.

Figure 9.1: The execution of test-and-set atomic instructions from multiple processors.

Let us now evaluate the test-and-set lock implementation. The uncontended lock-acquisition latency is low since only one atomic instruction, plus a branch instruction, are needed to successfully obtain a lock. The traffic requirement, however, is very high. Each lock acquisition attempt causes invalidation of all cached copies, regardless of whether the acquisition is successful or not. For example, look at the example in Table 9.1, in which three threads running on three processors all trying to obtain a lock once. Initially, no one holds the lock. Then, P1 executes a test-and-set and successfully obtains the lock. The test-and-set instruction incurs a BusRdX bus transaction since the block is not already cached in P1's cache. Suppose that after P1 obtains the lock (while P1 is in the critical section), P2 and P3 attempt to acquire the lock. P2 executes a test-and-set, followed by an attempt by P3, and then by P2 for the second time. Each of this attempt involves a test-and-set instruction, so it incurs a BusRdX transaction to invalidate other cached copies and transitions the state of the block that contains `lockvar` to modified. Note that despite the fact that the lock is held by P1,

P2 and P3 will keep incurring bus transactions to invalidate each other's copy in attempts to acquire the lock. This is one significant drawback of the test-and-set lock implementation. In fact, there may be many more failed attempts than shown in the table.

When later P1 releases the lock, it writes "0" to the lock variable, causing invalidation through the BusRdX transaction. Then, both P2 and P3 try to acquire the lock, but P2 succeeds and P3 fails. P3 keeps trying and failing to acquire the lock that is held by P2 for as long as P2 is in the critical section (the table shows only two failed attempts, but in reality they can occur much more times). After P2 releases the lock, P3 can acquire the lock successfully. Since no processor is contending for the lock now, when P3 releases the lock, it still has the cache block in modified state and does not incur a BusRdX bus transaction to store "0" to the lock address.

Table 9.1: Illustration of test-and-set lock implementation performance.

	Request	P1	P2	P3	Bus Request	Comments
0	Initially	–	–	–	–	lock is free
1	t&s1	M	–	–	BusRdX	P1 obtains lock
2	t&s2	I	M	–	BusRdX	P2 lock acq fails
3	t&s3	I	I	M	BusRdX	P3 lock acq fails
4	t&s2	I	M	I	BusRdX	P2 lock acq fails
5	unlock1	M	I	I	BusRdX	P1 releases lock
6	t&s2	I	M	I	BusRdX	P2 obtains lock
7	t&s3	I	I	M	BusRdX	P3 lock acq fails
8	t&s3	I	I	M	–	P3 lock acq fails
9	unlock2	I	M	I	BusRdX	P2 releases lock
10	t&s3	I	I	M	BusRdX	P3 obtains lock
11	unlock3	I	I	M	–	P3 releases lock

The table shows that a bus transaction is incurred at every attempt to acquire the lock, regardless of whether the lock is currently held or not. The amount of traffic for a single critical section that appears in the source code is quite high:

$\Theta(p^2)$. This is because there are $\Theta(p)$ processors which will enter the critical section, which means that there are $\Theta(p)$ lock acquisitions and releases. After each lock release, there will be $\Theta(p)$ attempts to acquire the lock, with one processor succeeding in acquiring the lock, and the rest failing in acquiring the lock.

Obviously, the traffic requirement of the test-and-set lock is excessively high. The traffic due to the lock acquisition attempts can slow down all traffic from regular cache misses and other coherence misses. Indeed, the critical section itself may become significantly slower if traffic is saturated by failed lock acquisition attempts, causing the thread that holds the lock to release the lock even later, which further exacerbates the traffic contention.

One way to reduce the traffic requirement is to use a **back-off** strategy, in which after a failed lock acquisition attempt, a thread "waits" (or backs off) before performing another attempt. A delay can be inserted between subsequent retries. The delay in the back-off strategy needs to be carefully tuned: if the delay is too small, there will still be a high amount of traffic; and if it is too large, the thread may miss an opportunity to acquire the lock when the lock becomes available. In practice, an **exponential back-off** strategy, in which the delay starts small but gradually increases exponentially, works quite well.

9.1.4 Test and Test and Set Lock

Another way to reduce the traffic requirement is to have a criteria that tests whether a lock acquisition attempt will likely lead to failure, and if that is the case, we defer the execution of the atomic instruction. Only when a lock acquisition attempt has a good chance of succeeding do we attempt the execution of the atomic instruction. Using this approach, the test-and-set lock can be improved and the improved version is referred to as the **test-and-test-and-set lock** (TTSL) implementation. The code for TTSL is shown in Code 9.3.

The load instruction and the following branch instruction form a tight loop that keeps reading (but not writing) the address where `lockvar` is located until the value in the location turns 0. Therefore, while a lock is being held by a processor, other processors do not execute the test-and-set atomic instructions. This prevents repeated invalidations that are useless as they lead to failed lock acquisitions. Only when the lock is released, will the processors attempt to acquire the lock using the atomic instructions. The uncontended lock-acquisition latency is higher than the test-and-set lock implementation due to the extra load and branch instructions. However, the traffic during the time a lock is held is significantly reduced. Only a

Code 9.3 An implementation of test-and-test-and-set lock (TTSL).

```
1 lock: ld  R1, &lockvar // R1 = lockvar
2       bnz R1, lock     // jump to lock if R1 != 0
3       t&s R1, &lockvar // R1 = lockvar
4                        // if (R1 == 0) lockvar=1
5       bnz R1, lock     // jump to lock if R1 != 0
6       ret              // return to caller
7
8 unlock: st  &lockvar, #0 // lockvar = 0
9         ret              // return to caller
```

small fraction of lock acquisition attempts generate traffic.

Table 9.2 illustrates the performance of the TTSL implementation. Suppose that each processor needs to get the lock exactly once. Initially, no one holds the lock. Then, P1 executes a load and finds out that the lock is available. So it attempts a test-and-set and successfully obtains the lock. The test-and-set instruction does not incur a BusRdX bus transaction since the block is exclusively cached. Suppose that after P1 obtains the lock (while P1 is in the critical section), P2 and P3 attempt to acquire the lock. Compared to the test-and-set lock implementation, in TTSL, P2 executes a load, followed by a load by P3. Both find that the value in `lockvar` location is one, indicating the lock is currently held by another processor. So they keep reading from the cache block, waiting until the value changes. While they are waiting, no bus transaction is generated since they are spinning using a load instruction rather than a test-and-set instruction. One of such load is shown in line 5. In reality, it is likely that many loads are executed, rather than just one, so the advantage of TTSL over the test-and-set lock implementation increases. Note that an additional benefit of spinning using loads is that since bus bandwidth is conserved, the thread that is in the critical section does not experience the bandwidth contention that could slow it down.

Next, when P1 releases the lock, suppose that P2 is the first to suffer from a miss and discover that the lock has been freed; P2 then acquires the lock using the test-and-set instruction. This sequence produces two bus transactions: a bus read for loading the value, and a bus upgrade caused by the test-and-set instruction to write to the block. In contrast, in the test-and-set lock implementation, only one bus transaction is involved. Hence, the lock acquisition in TTSL incurs slightly more bandwidth usage than the test-and-set lock implementation, although overall, less bandwidth is used due to the use of loads for spinning.

Next, suppose that P3 spins on the lock variable (line 9 and 10). When the lock is released by P2 (line 11), P3 attempts to read, and successfully acquires the lock, and succeeds. And finally, P3 releases the lock.

Table 9.2: Illustration of TTSL implementation performance.

	Request	P1	P2	P3	Bus Request	Comments
0	Initially	–	–	–	–	lock is free
1	ld1	E	–	–	BusRd	P1 reads from &lockvar
2	t&s1	M	–	–	–	P1 obtains lock
3	ld2	S	S	–	BusRd	P2 reads from &lockvar
4	ld3	S	S	S	BusRd	P3 reads from &lockvar
5	ld2	S	S	S	–	P2 reads from &lockvar
6	unlock1	M	I	I	BusUpgr	P1 releases lock
7	ld2	S	S	I	BusRd	P2 reads from &lockvar
8	t&s2	I	M	I	BusUpgr	P2 obtains lock
9	ld3	I	S	S	BusRd	P3 reads from &lockvar
10	ld3	I	S	S	–	P3 reads from &lockvar
11	unlock2	I	M	I	BusUpgr	P2 releases lock
12	ld3	I	S	S	BusRd	P3 reads from &lockvar
13	t&s3	I	I	M	BusUpgr	P3 obtains lock
14	unlock3	I	I	M	–	P3 releases lock

9.1.5 Load Linked and Store Conditional Lock

Although TTSL is an improvement over the test-and-set lock implementation, it still has a major drawback in its implementation. As mentioned before, one way to implement an atomic instruction requires a separate bus line that is asserted when a processor is executing an atomic instruction. Such an implementation is not general enough as it only works for bus-based multiprocessors. In addition, it works well only if lock frequency is low. If programmers use fine-grain locks,

they use many lock variables simultaneously. There are many instances of lock acquisitions and releases, and many of them are unrelated as they are used, not for ensuring a critical section, but rather, for ensuring exclusive access to different parts of a data structure. If each of these lock acquisition asserts a single bus line, then unnecessary serialization occurs.

Another implementation, which reserves a cache block for the entire duration of the atomic instruction, is more general. It does not assume the existence of a special bus line so it can work with other interconnects as well. However, to prevent a cache block from being stolen by other processors, requests to the block must be deferred or negatively acknowledged. Such a mechanism can be quite costly to implement. Deferring requests requires an extra buffer to queue the requests, while negative acknowledgments waste bandwidth and incur delay as requests are retried in the future.

In order to avoid complexity associated with supporting an atomic instruction, an alternative is to provide an *illusion of atomicity* on a sequence of instructions, rather than true instruction atomicity. Note that the lock acquisition essentially consists of a load (`ld R1, &lockvar`), some instructions such as the conditional branch (`bnz R1, lock`), and a store (`st &lockvar, R1`). Atomicity implies that either none or all of the instructions appear to have executed. From the point of view of other processors, only the store instruction is visible, as it changes a value that may be seen by them. Other instructions (the load and the branch) have their effect on the registers of the local processor but their effect is not visible to other processors. Hence, from the point of view of other processors, it does not matter if the load and the branch have executed or not. From the point of view of the local processor, these instructions can be canceled easily by ignoring their register results, and re-executed later. Therefore, the instruction that is critical to the illusion of atomicity is the store instruction. If it is executed, its effect is visible to other processors. If it is not executed, its effect is not visible to other processors. Therefore, in order to give the illusion of atomicity to the sequence of instructions, we need to ensure that the store fails (or gets canceled) if between the load and the store, something has happened that potentially violates the illusion of atomicity. For example, if there is a context switch or interrupt that occurs between the load and the store, then the store must be canceled (servicing the interrupt may result in the value of the block to change without the knowledge of the processor, which breaks atomicity). In addition, with respect to other processors, if the cache block that is loaded has been invalidated before

the store executes, then the value in the cache block may have changed, resulting in a violation to atomicity. In this case, the store must be canceled. If, however, the block remains valid in the cache by the time the store executes, then the entire load-branch-store sequence can appear to have executed atomically.

The illusion of atomicity requires that the store instruction is executed conditionally upon detected events that may break the illusion of atomicity. Such a store is well known as a **store conditional** (SC). The load that requires a block address to be monitored from being *stolen* (i.e., invalidated) is known as a **load linked/locked** (LL). The LL/SC pair turns out to be a very powerful mechanism on which many different atomic operations can be built. The pair ensures the illusion of atomicity without requiring an exclusive ownership of a cache block.

An LL is a special load instruction that not only reads a block into a register, but also records the address of the block in a special processor register which we will refer to as a **linked register**. A SC is a special store instruction that succeeds only when the address involved matches the address stored in the linked register. To achieve the illusion of atomicity, the SC should fail when another processor has raced past it, successfully performed an SC, and stolen the cache block. To ensure the failure of SC, the linked register is cleared when an invalidation to the address stored in the linked register occurs. In addition, when a context switch occurs, the linked register is also cleared. When a SC fails (due to mismatch of the SC address and the address in the linked register), the store is canceled without going to the caches. Hence, to the memory system, it is as if the SC is never executed. All other instructions in the atomic sequence (LL included) can simply be repeated as if they have also failed.

Code 9.4 shows the code that implements a lock acquisition and release using an LL/SC pair, which is identical with Code 7.3 except that the load is replaced with the LL, and the store is replaced with an SC.

The LL instruction and the following branch instruction forms a tight loop that keeps reading (but not writing) the location where `lockvar` is until the value in the location turns 0. Thus it behaves similarly to a TTSL implementation which spins using loads. Therefore, while a lock is being held by a processor, the SC is not executed. This prevents repeated, useless, invalidations that correspond to failed lock acquisitions. Only when the lock is released, will the processors attempt to acquire the lock using the SC instruction. It is possible that multiple processors attempt to perform the SC simultaneously. On a bus-based multiprocessor, one of the SC will be granted a bus access and performs the store suc-

Code 9.4 An implementation of LL/SC lock. The code assumes that a failed SC returns a 0.

```
lock: LL   R1, &lockvar  // R1 = lockvar;
                         // LINKREG = &lockvar
      bnz  R1, lock      // jump to lock if R1 != 0
      add  R1, R1, #1    // R1 = 1
      SC   &lockvar, R1  // lockvar = R1;
      beqz R1, lock      // jump to lock if SC fails
      ret                // return to caller

unlock: st  &lockvar, #0 // MEM[&lockvar] = 0
        ret              // return to caller
```

cessfully. Other processors snoop the store on the bus and clear out their linked registers, causing their own SC to fail. In contrast to a test-and-set atomic instruction, when an SC fails, it does not generate a bus transaction (a test-and-set instruction always generates a BusRdX or BusUpgr).

> ■ *Did you know?*
>
> *LL/SC instruction pairs are supported in various instruction sets such as Alpha (`ldl_l/stl_c` and `ldq_l/stq_c`), PowerPC (`lwarx/stwcx`), MIPS (`LL/SC`), and ARM (`ldrex/strex`). The conditions that make SC fails can vary across implementations although sometimes the failure is not strictly necessary given the conditions. For example, in some implementations, SC fails when another LL is encountered, or even when ordinary loads or stores are encountered.*

Table 9.3 illustrates the performance of the LL/SC lock implementation. Suppose that each processor needs to get the lock exactly once. Initially, no one holds the lock. The sequence of bus transactions generated is identical to ones in the TTSL implementation, except that the LL replaces the regular load, and SC replaces the test-and-set instruction.

Therefore, performance-wise, the LL/SC lock implementation performs similarly to a TTSL implementation. A minor difference is when multiple processors simultaneously perform SCs. In this case, only one bus transaction occurs in LL/SC (due to the successful SC), whereas in the TTSL, there will be multiple bus transactions corresponding test-and-set instruction execution. However, this is a quite rare event since there are only a few instructions between a load and a store in an atomic sequence, so the probability of multiple processors executing

Table 9.3: Illustration of LL/SC lock implementation performance.

	Request	P1	P2	P3	Bus Request	Comments
0	Initially	–	–	–	–	lock is free
1	ll1	E	–	–	BusRd	P1 reads from &lockvar
2	sc1	M	–	–	–	P1 obtains lock
3	ll2	S	S	–	BusRd	P2 reads from &lockvar
4	ll3	S	S	S	BusRd	P3 reads from &lockvar
5	ll2	S	S	S	–	P2 reads from &lockvar
6	unlock1	M	I	I	BusUpgr	P1 releases lock
7	ll2	S	S	I	BusRd	P2 reads from &lockvar
8	sc2	I	M	I	BusUpgr	P2 obtains lock
9	ll3	I	S	S	BusRd	P3 reads from &lockvar
10	ll3	I	S	S	–	P3 reads from &lockvar
11	unlock2	I	M	I	BusUpgr	P2 releases lock
12	ll3	I	S	S	BusRd	P3 reads from &lockvar
13	sc3	I	I	M	BusUpgr	P3 obtains lock
14	unlock3	I	I	M	–	P3 releases lock

the same sequence at the same time is small.

However, the advantages of LL/SC over using atomic instructions are numerous. First, LL/SC implementation is relatively simpler (extra linked registers). Secondly, it can be used to implement many atomic instructions, such as test-and-set, compare-and-swap, etc. Hence, it can be thought as a lower level primitive than atomic instructions.

However, the LL/SC lock implementation still has a significant scalability problem. Each lock acquisition or lock release triggers invalidation of all sharers that spin on the lock variable because the value of the lock variable has changed. They in turn suffer from cache misses to reload the block containing the lock variable. Hence, if there are $\Theta(p)$ lock acquisitions and releases, and each acquisition or release triggers $\Theta(p)$ subsequent cache misses, then the total traffic of a LL/SC

lock implementation scales quadratically to the number of threads in the system, or $\Theta(p^2)$. This is in addition to the issue of lock fairness in which there is no guarantee that the thread that attempts to acquire the lock the earliest can actually acquire the lock earlier than others. The following two lock implementations will address each of these problems.

9.1.6 Ticket Lock

The **ticket lock** is a lock implementation that attempts to provide fairness in lock acquisition. The notion of fairness deals with whether the order in which threads first attempt to acquire a lock corresponds to the order in which threads acquire the lock successfully. This notion of fairness automatically guarantees that a thread is not starved out from failing to acquire a lock for a long period of time because other threads always beat to it.

To achieve such fairness, the ticket lock implementation uses a concept of a queue. Each thread that attempts to acquire a lock is given a ticket number in the queue, and the order in which lock acquisition is granted is based on the ticket numbers, with a thread holding the lowest ticket number is given the lock next.

To implement that, two variables are introduced. The first one is `now_serving` and the other is `next_ticket`. The role of `next_ticket` is to reflect the next available ticket number (or the order position) that a new thread should get. The role of `now_serving` is to reflect the position of the current holder of the lock. Necessarily, $next_ticket - now_serving - 1$ is the number of threads that are currently waiting for the lock, i.e. they have attempted to acquire the lock but have not obtained it. To obtain a lock, a thread reads the `next_ticket` into a private variable (say, `my_ticket`) and atomically increments it, so that future threads will not share the same ticket number. Then, it waits until `now_serving` is equal to the value of `my_ticket`. A thread that is currently holding the lock releases the lock by incrementing the `now_serving` variable so that the next thread in line will find that the new value of `now_serving` is equal to its `my_ticket`, and the thread can successfully acquire the lock. The complete code for lock acquisition and release are shown in Code 9.5.

Note that the code shows the implementation in a high-level language, assuming that there is only one lock (so a lock's name is not shown) and fetch the atomic primitive `fetch_and_inc()` is supported. As we have seen in the discussion of LL/SC lock implementation, such primitive can be implemented easily using a pair of LL and SC. The only thing that needs to be inserted between the

Code 9.5 Ticket lock implementation.

```
ticketLock_init(int *next_ticket, int *now_serving)
{
  *now_serving = *next_ticket = 0;
}

ticketLock_acquire(int *next_ticket, int *now_serving)
{
  my_ticket = fetch_and_inc(next_ticket);
  while (*now_serving != my_ticket) {};
}

ticketLock_release(int *now_serving)
{
  now_serving++;
}
```

LL and SC is an instruction that increments the value read from memory by LL. We also assume that `fetch_and_inc()` is implemented as a function that returns the old value of the argument prior to incrementing it. Finally, the values of `now_serving` and `next_ticket` are both initially initialized to 0 before use.

To illustrate how the ticket lock works, consider three processors contending for the lock and each of them will acquire and release the lock exactly once. Table 9.4 shows the values of the important variables at each step: `now_serving`, `next_ticket`, P1's `my_ticket`, P2's `my_ticket`, and P3's `my_ticket`. First, P1 tries to acquire lock, it executes `fetch_and_inc(next_ticket)` atomically, resulting in the value of `next_ticket` to increase to 1. It then compares `my_ticket` and `now_serving`, and since their values are equal, it successfully acquires the lock and enters the critical section. P2, contending for the lock, also attempts to acquire the lock by executing `fetch_and_inc (next_ticket)`, resulting in the `next_ticket` to increase to 2. It then compares `my_ticket`, which has a value of 1, and `now_serving`, which has a value of 0 (since the lock is still held by the thread having the ticket number 0). Since their values are not equal, P2 fails the lock acquisition, stays in the loop to keep on testing the values of `now_serving`. P3, also contending for the lock, attempts to acquire the lock by executing `fetch_and_inc(next_ticket)`, resulting in the `next_ticket` to increase to 3. It then compares `my_ticket`, which has a value of 2, and `now_serving`, which still has a value of 0 (since the lock is still

held by P1). Since their values are not equal, P3 fails the lock acquisition, and like P2, it stays in the loop to keep on testing their values. When P1 releases the lock, it executes the `now_serving++` statement, incrementing the `now_serving` to a value of 1. Now P2's `my_ticket` and `now_serving` match, P2 gets out of its loop and enters the critical section. Similarly, when P2 releases the lock, it executes the `now_serving++` statement, incrementing the `now_serving` to a value of 2. Now P3's `my_ticket` and `now_serving` match, P3 gets out of its loop and enters the critical section.

Table 9.4: Illustration of the ticket lock mechanism.

Steps	next_ticket	now_serving	**P1** my_ticket	**P2** my_ticket	**P3** my_ticket	**Comments**
Initially	0	0	0	0	0	all initialized to 0
P1: f&i(next_ticket)	1	0	0	0	0	P1 tries to acq lock
P2: f&i(next_ticket)	2	0	0	1	0	P2 tries to acq lock
P3: f&i(next_ticket)	3	0	0	1	2	P3 tries to acq lock
P1: now_serving++	3	1	0	1	2	P1 rels, P2 acqs lock
P2: now_serving++	3	2	0	1	2	P2 rels, P3 acqs lock
P3: now_serving++	3	3	0	1	2	P3 rels lock

The performance of ticket lock depends on how the `fetch_and_inc` atomic operation is implemented. If it is implemented with an atomic instruction, then its scalability is similar to that of the atomic instruction. If the `fetch_and_inc` is implemented using a LL/SC pair, then its scalability is similar to that of LL/SC lock implementation. In particular, the `fetch_and_inc` on `next_ticket` will cause a processor to invalidate all copies of the block containing `next_ticket`, and each processor subsequently suffers from a cache miss to reload the block. Similarly, since all processors are monitoring and spinning on the `now_serving`, when a processor releases the lock, all copies of the block containing `now_serving` are invalidated, and each processor subsequently suffers from a cache miss to reload the block. Since there are $\Theta(p)$ number of acquisition and releases, and each acquire and release causes $\Theta(p)$ invalidations and subsequent cache misses, the total traffic scales on the order of $\Theta(p^2)$.

The uncontended latency of a ticket lock implementation is slightly higher

than that of LL/SC because it has extra instructions that read and test the value of `now_serving`. However, the ticket lock provides fairness, which is not provided by the LL/SC lock implementation.

9.1.7 Array-Based Queuing Lock

At this point, readers may wonder if there is a lock implementation that has the fairness of the ticket lock and at the same time better scalability compared to all the previous lock implementations discussed so far. One answer to that lies in a lock implementation called the **array-based queuing lock** (ABQL). ABQL is an improvement over the ticket lock. It starts out from the observation that since threads already queue up waiting for the lock acquisition in the ticket lock, they can wait and spin on unique memory locations rather than on a single location represented by `now_serving`. The analogy for the new mechanism is for the current lock holder to pass the baton to the next thread in the queue, and for that thread to pass the baton to the next next thread, and so on. Each time the baton is passed, only the next thread needs to be notified.

To implement ABQL, we need to ensure that threads spin on unique memory locations. To achieve that, we can change the `now_serving` to an array, which we will rename to `can_serve` to reflect its role better. As with the ticket lock implementation, each thread that wants to acquire a lock obtains a ticket number, say x. It then waits in a loop until the xth element in the `can_serve` array has a value of 1, which will be set by the thread ahead of it in the queue. When a thread with a ticket number y releases a lock, it passes the baton to the next thread by setting the $(y+1)^{th}$ element in the `can_serve` array to 1. The complete code for lock acquisition and release are shown in Code 9.6.

Note that the code shows the implementation in a high-level language, assuming that there is only one lock and the atomic primitive `fetch_and_inc()` is supported. Initially, the values of `next_ticket` is initialized to 0. Elements of the array `can_serve` are initialized to 0 as well, except for the first element which is initialized to 1, in order to allow the first lock acquisition to be successful.

To illustrate how ABQL works, consider three processors contending for the lock and each of them will acquire and release the lock exactly once. Table 9.5 shows the values of the important variables at each step: `can_serve` (assuming it has four elements), `next_ticket`, P1's `my_ticket`, P2's `my_ticket`, and P3's `my_ticket`.

First, P1 tries to acquire lock, it executes `fetch_and_inc(next_ticket)`

Code 9.6 Array-Based Queuing Lock implementation.

```
ABQL_init(int *next_ticket, int *can_serve)
{
  *next_ticket = 0;
  for (i=1; i<MAXSIZE; i++)
    can_serve[i] = 0;
  can_serve[0] = 1;
}

ABQL_acquire(int *next_ticket, int *can_serve)
{
  *my_ticket = fetch_and_inc(next_ticket);
  while (can_serve[*my_ticket] != 1) {};
}

ABQL_release(int *can_serve)
{
  can_serve[*my_ticket + 1] = 1;
  can_serve[*my_ticket] = 0;    // prepare for next time
}
```

atomically, resulting in the value of `next_ticket` to increase to 1, and it obtains a value of 0 for its `my_ticket`. Then it checks whether `can_serve[my_ticket]` (`can_serve[0]`) is 1. Since the value is 1, it successfully acquires the lock and enters the critical section. P2, contending for the lock, also attempts to acquire the lock by executing `fetch_and_inc(next_ticket)`, resulting in the `next_ticket` to increase to 2, and it obtains the value of 1 for its `my_ticket`. It then checks whether `can_serve[1]==1`. But at this point `can_serve[1]` has a value of 0 since P1 has not released its lock. So P2 fails the lock acquisition and stays in the loop. P3, also contending for the lock, also attempts to acquire the lock by executing `fetch_and_inc(next_ticket)`, resulting in the `next_ticket` to increase to 3, and it obtains the value of 2 for its `my_ticket`. It then checks whether `can_serve[2]==1`. But at this point `can_serve[2]` has a value of 0 since P1 has not released the lock (P1 has not even acquired the lock). So P3 fails the lock acquisition, stays in the loop. When P1 releases the lock, it resets `can_serve[0] to 0` and sets `can_serve[1]` to 1, allowing P2 to get out of its loop and enter the critical section. When P2 releases the lock, it resets `can_serve[1] to 0` and sets `can_serve[2]` to 1, allowing P3 to get out of its loop and enter the critical section. Finally, P3 releases the lock by

setting `can_serve[3]` to 1, and resets `can_serve[2]` to 0.

Table 9.5: Illustration of ABQL mechanism.

Steps	next_ ticket	can_ serve	P1	P2	P3	Comments
			my_ticket			
Initially	0	[1, 0, 0, 0]	0	0	0	all initialized to 0
P1: f&i(next_ticket)	1	[1, 0, 0, 0]	0	0	0	P1 tries to acq lock
P2: f&i(next_ticket)	2	[1, 0, 0, 0]	0	1	0	P2 tries to acq lock
P3: f&i(next_ticket)	3	[1, 0, 0, 0]	0	1	2	P3 tries to acq lock
P1: can_serve[1]=1; can_serve[0]=0	3	[0, 1, 0, 0]	0	1	2	P1 rels, P2 acqs lock
P2: can_serve[2]=1; can_serve[1]=0	3	[0, 0, 1, 0]	0	1	2	P2 rels, P3 acqs lock
P3: can_serve[3]=1; can_serve[2]=0	3	[0, 0, 0, 1]	0	1	2	P3 rels lock

Consider the way a thread releases a lock. It writes to one element of the array `can_serve[my_ticket+1]`. Assume that the array is padded such that different array elements reside in different cache blocks. Since exactly one thread spins in a loop reading from the element, the element is only stored in one cache. The write only invalidates one cache block, and only one subsequent cache miss occurs. Hence, on each release, only $\Theta(1)$ traffic is generated. That means that since there are $\Theta(p)$ number of acquisitions and releases, the total lock release traffic scales on the order of $\Theta(p)$, a significant improvement over the ticket lock implementation. However, note that the `fetch_and_inc`'s scalability depends on its underlying implementation. If it is implemented with a LL/SC, its scalability may partially restrict the overall scalability of the ABQL implementation.

9.1.8 Qualitative Comparison of Lock Implementations

Table 9.6 compares the various lock implementations based on several criteria: uncontended latency, the amount of traffic following a single lock release, the amount of traffic waiting while the lock is held by a processor, storage overheads, and whether fairness guarantee is provided.

The uncontended latency is the lowest on simpler lock implementations such

Table 9.6: Comparison of various lock implementations.

Criteria	**test&set**	**TTSL**	**LL/SC**	**Ticket**	**ABQL**
Uncontended latency	Lowest	Lower	Lower	Higher	Higher
1 Release max traffic	$\Theta(p)$	$\Theta(p)$	$\Theta(p)$	$\Theta(1)$	
Wait traffic	High	–	–	–	–
Storage	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(p)$
Fairness guarantee?	No	No	No	Yes	Yes

as the test_and_set, TTSL, and LL/SC. Ticket lock and ABQL implementations execute more instructions so their uncontended lock acquisition latency is higher.

In terms of the traffic following a single lock release, assuming all other threads are waiting to acquire the lock next, test_and_set, TTSL, LL/SC, and ticket lock have the highest amount of maximum traffic. This is because all threads spin on the same variable so they may all cache the same block, and each lock release invalidates all other processors and forces them to suffer misses to reload the block. On the other hand, in ABQL, a single release only invalidates one other cache causing only one subsequent cache miss.

In terms of the traffic generated while a lock is held by a thread, test_and_set performs very poorly because all threads keep on attempting to acquire the lock using an atomic instruction and even a failed attempt still generates an invalidation of all sharers and subsequent acquisition attempts by the sharers. On the other hand, TTSL, LL/SC, ticket lock, and ABQL, spin wait using a load instruction. Hence, once a thread discovers that the lock is held, it does not attempt to execute an atomic instruction.

In terms of storage requirements, all of the lock implementations use one or two shared variables so the storage requirement is constant across number of processors. Only ABQL has a storage requirement that scales with the number of processors due to keeping the array `can_serve`.

In terms of guarantee of fairness, only the ticket lock and ABQL provide it, due to the use of a queue. In other implementations, it is possible (though in reality may not be likely) that a processor has a better chance at acquiring the lock more than others following a release. For example, that may occur when a thread that releases a lock (and invalidating other caches) quickly acquires the

lock to re-enter the critical section again. In the mean time, other processors have not even had a chance to re-attempt a lock acquisition because they are still reloading the invalidated block. In this case, it is possible that a thread keeps on acquiring and releasing a lock at the expense of other threads' ability to acquire the lock. Other causes may also be possible. For example, if the bus arbitration logic favors some requesters than others, some processors may be granted the bus faster than others. In a distributed shared memory system, one processor may reload an invalidated block faster than others, for example due to reloading the block from its local memory whereas others must reload it from a remote memory. Following a release, the processor that reloads the block from its local memory may race past others to attempt a lock acquisition.

Note, however, fairness guarantee creates a risk in performance. If a thread that is already in the queue waiting to get a lock is context switched out, then even when the lock becomes available, the thread will not attempt to acquire the lock. Other threads with larger ticket numbers cannot acquire the lock either because they must wait until the thread that is switched out has acquired and released the lock. Thus, the performance of all threads are degraded by the context switch of one of the threads. Therefore, care must be taken to ensure context switches do not occur when using the ticket lock or ABQL implementations.

From the point of view of software, it is not immediately clear which lock implementation is the best. For software showing a high degree of lock contention, ABQL offers the highest scalability. However, a high degree of lock contention is often a symptom of a deeper scalability problem, such as the use of lock granularity that is too coarse grain. In such a case, using ABQL improves scalability but may not make the program scalable. Better scalability can be obtained using a finer lock granularity. For example, in the linked list parallelization discussed in Chapter 5, using the fine grain lock approach, in which each node is augmented with its own lock, will likely make the contention level of any particular lock very low. In that case, ABQL is not only unnecessary, it is not even preferred due to a high uncontended latency and the fact that the storage overhead of one array for each node will be too high.

9.2 Barrier Implementations

Barriers are very simple and widely used synchronization primitives in parallel programs. We have discussed in Chapter 3 that in loop-level parallelization, barri-

ers are often used at the end of a parallel loop to ensure all threads have computed their parts of the computation before the computation moves on to the next step. In the OpenMP `parallel for` directive, a barrier is automatically inserted at the end of a parallel loop, and programmers must explicitly remove it if they believe that the lack of barrier does not impact the correctness of their computation.

In this section, we will look at how barriers can be implemented, and compare the characteristics of the implementations in various terms. Barriers can be implemented in software or directly in hardware. Software barrier implementation is flexible but is often inefficient, whereas hardware barrier implementation restricts the flexibility and portability of the software but is often a lot more efficient. The simplest software barrier implementation is an implementation called the **sense-reversal global barrier**. The key mechanism used is for threads to enter the barrier and spin on a location that will be set by the last thread that arrives at the barrier, releasing all threads that are waiting there. Obviously, spinning on a single location often involves a lot of invalidations when the location is written, restricting scalability. One of the more complex but more scalable barrier implementations is **combining tree barrier** in which the threads that participate in a barrier are organized like a tree, and a thread at a particular tree level spins on a common location only with its siblings. This restricts the number of invalidations as threads spin on different locations. Finally, we will look at a hardware barrier implementation support that is implemented in some very large machine with thousands of processors.

The criteria for evaluating barrier performance include:

1. **Latency**: the time spent from entering the barrier to exiting the barrier. The latency in the barrier should be as small as possible.

2. **Traffic**: the amount of bytes communicated between processors as a function of the number of processors.

In contrast to lock implementation, fairness and storage overheads are not important issues because barriers are a global construct involving many threads.

9.2.1 Sense-Reversal Centralized Barrier

A software barrier implementation only assumes that lock acquisition and release primitives to be provided by the system (through software, hardware, or a combination of hardware and software). For example, as long as one of test-and-set,

TTSL, LL/SC, ticket, or ABQL lock implementation is available, a barrier implementation can be constructed as well.

We note that from the point of view of programmers, a barrier must be simple, that is, a barrier should not require any arguments passed to it, either variable names or number of processors or threads. In OpenMP standard, for example, a barrier can be invoked simply as `#pragma omp barrier`. In the actual implementation, arguments may be used, as long as they are not exposed to the programmers.

The basic barrier implementation is shown in Code 9.7. The implementation makes use of several variables. `barCounter` keeps track of how many threads have so far arrived at the barrier. `barLock` is a lock variable to protect a modification to shared variables in a critical section. `canGo` is a flag variable that threads spin on to know whether they can go past the barrier or not yet. Hence, `canGo` is set by the last thread to release all threads from the barrier.

Code 9.7 Simple (but incorrect) barrier implementation.

```
// declaration of shared variables used in a barrier
// and their initial values
int numArrived = 0;
lock_type barLock = 0;
int canGo = 0;

// barrier implementation
void barrier () {
  lock(&barLock);
    if (numArrived == 0) { // first thread sets flag
      canGo = 0;
    }
    numArrived++;
    myCount = numArrived;
  unlock(&barLock);

  if (myCount < NUM_THREADS) {
    while (canGo == 0) {};    // wait for last thread
  }
  else {  // last thread to arrive
    numArrived = 0;  // reset for next barrier
    canGo = 1;         // release all threads
  }
}
```

For example, suppose that three threads P1, P2, and P3 arrive at the barrier in that order. P1 arrives at the barrier first and enters the critical section. Then it initializes the variable `canGo` to 0 so that itself and other threads will wait until the last thread arrives and sets it to 1. It then increments `numArrived` to 1, assigns its value to `myCount`, and exits the critical section. Next, it enters a loop to wait until canGo has a value of 1. The second thread P2 enters the barrier, increments `numArrived` to 2, discovers that it is not the last thread (`myCount` has a value of 2, smaller than `NUM_THREADS`) and also enters the loop to wait until canGo has a value of 1. Finally, the last thread P3 enters the barrier, and increments `numArrived` to 3. It discovers that `myCount` is equal to `NUM_THREADS` so it is the last thread to arrive at the barrier. It then resets the value of `numArrived` to 0 so that it can be used in the next barrier. Then, it releases all waiting threads by setting `canGo` to 1. This allows all waiting threads to get out of the loop and resume computation past the barrier.

Unfortunately, the code described above does not work correctly across more than one barrier. For example, when the last thread P3 sets the value of `canGo` to 1, it invalidates all cached copies of the block where `canGo` is located. Following the invalidations, the block resides in P3's cache, while P1 and P2 try to reload the block. Suppose that before that can happen, P3 enters the next barrier, now as the first thread that arrives at that second barrier. As the first thread, it resets the variable `canGo` to 0. It can do so quickly since it has the block in its cache in a modified state. When P1 and P2 reload the block, they discover that the value of `canGo` is 0 as affected by the second barrier rather than 1 as released from the first barrier. Therefore, P1 and P2 stay in the loop of the first barrier and never get released, while P3 waits in the loop of the second barrier, never to be released either.

One possible solution to the "deadlock" situation discussed earlier is to have a two-step release, in which before the first thread that enters a barrier initializes the `canGo` variable, it waits until all threads have been released from the previous barrier. Implementing such a solution requires another flag, another counter, and extra code, which can be quite costly. Fortunately, there is a simpler solution based on an observation that the error arises when the first thread entering the next barrier resets the value of `canGo`. If we avoid this reset, then the thread that enters the next barrier will not prevent other threads from getting out of the first barrier. To avoid resetting the value of `canGo`, in the second barrier, threads can instead wait until the value of `canGo` changes back to 0, which is accomplished by the last

thread that enter the second barrier. With this approach, the value of canGo that releases threads alternates from 1 in the first barrier, 0 in the second barrier, 1 in the third barrier, 0 in the fourth barrier, and so on. Because the value is toggling between barriers, this solution is referred to as the **sense-reversing centralized barrier**. The code for the barrier is shown in the following (Code 9.8).

Code 9.8 Sense-reversing barrier implementation.

```
// declaration of shared variables used in a barrier
int numArrived = 0;
lock_type barLock = 0;
int canGo = 0;

// thread-private variables
int valueToWait = 0;

// barrier implementation
void barrier () {
  valueToWait = 1 - valueToWait; // toggle it
  lock(&barLock);
    numArrived++;
    myCount = numArrived;
  unlock(&barLock);

  if (myCount < NUM_THREADS) {  // wait for last thread
    while (canGo != valueToWait) {};
  }
  else {  // last thread to arrive
    numArrived = 0;  // reset for next barrier
    canGo = valueToWait;   // release all threads
  }
}
```

The code shows that each thread, when entering a barrier, first toggles the value that it will wait from 0 to 1 or from 1 to 0. Then, it increments the counter and waits until the value of canGo is changed by the last thread. The last thread is the one that toggles the value of canGo.

The centralized (or global) barrier implementation uses a critical section inside the barrier routine so as the number of threads grow, the time in the barrier increases linearly. Actually, it may increase super-linearly depending on the underlying lock implementation. In addition, the traffic is high since each thread increments the variable numArrived, invalidating all sharers of the block, which

then have to reload the block through cache misses. Since there are $\Theta(p)$ such increments, and each increment can cause $\Theta(p)$ cache misses, the total traffic for the barrier implementation scales quadratically with the number of processors, or $\Theta(p^2)$. Hence, unfortunately, the centralized barrier is not very scalable.

9.2.2 Combining Tree Barrier

In a combining tree barrier, the situation in which all threads sharing and spinning on a single location (e.g. `barCount` or `canGo`) is avoided. Threads are logically organized as leaf nodes of a tree, with the root and intermediate nodes as locations on which the threads will spin on. The barrier starts with leaf nodes that are siblings (i.e., having a common parent node) spinning on a variable represented by the parent node. Other threads having a different parent spin on their own variables. We can think of a parent node as a sub- or mini-barrier since only a few threads that are its descendants participate. An invalidation of a location represented by the parent node only affects threads that are spinning on that location. Once threads of a common parent node have all arrived at the barrier, one thread moves on to represent the parent node in the barrier at the next (higher) level. This process is repeated until the root node is reached, which indicates that all threads have reached the barrier. Finally, the root releases all threads by setting a flag that all threads monitor, or by propagating the release flag down the tree all the way to the leaf nodes.

A combining tree barrier needs the storage space that is equivalent to the size of the tree, that is, $\Theta(p)$. The amount of traffic scales with the number of nodes, i.e. $\Theta(p)$, which is favorable compared to a centralized barrier's $\Theta(p^2)$. However, the latency is now higher, since to know that all threads have reached the barrier, we need to traverse up the tree, participating in $\Theta(log\ p)$ barriers at various levels of the tree, versus the centralized barrier's $\Theta(1)$.

9.2.3 Hardware Barrier Implementation

The simplest hardware barrier implementation in a bus-based multiprocessor is a special bus line that implements an OR logic. Each processor that arrives at the barrier asserts the barrier line. However, since the barrier implements an OR logic, the line is only high when all processors have reached the barrier and asserted the line.

On a non bus-based multiprocessor system, a special-purpose network for propagating barrier signals can be used. One example of this approach can be

found in the IBM BlueGene supercomputer. The IBM BlueGene system dominates the world's most powerful supercomputers according to www.top500.com (as of 2007). It was designed to scale up to thousands of processors, with one system having as many as 65 thousand processors. Obviously, with such a huge system, none of the software barrier implementations we have discussed can scale well. First, they rely on executing many instructions, including atomic instructions of which their scalability is not very high. They also rely on cache coherence protocols to propagate values of synchronization variables as they are changed by a thread. The cache coherence protocol itself cannot scale up to tens of thousands of processors. Therefore, the BlueGene/L system provides a specialized hardware support for fast barrier.

The BlueGene/L system [Adiga, et al., 2002] has several types of interconnection network among processors. One network, a three-dimensional torus, is used for regular data communication between processors. There is also a fast Ethernet network for accessing a file system and for network management. What is interesting is that there are two additional, special-purpose, networks. The first one is a **collective network** which provides for low-latency global communications, including global arithmetic operations such as global summation. The second specialized network is a global **barrier network** for implementing barriers and for broadcasting notifications to all processors. Rather than spinning on a variable, relying on a cache coherence protocol to distribute the block where the variable resides, and using many software instructions, a hardware barrier network works literally by sending signals along the wires in the network, and a hardware controller in each node that passes the signal along to other nodes. Hence, the speed is very high, and in most cases the scalability is very good.

The collective network was designed to support global operations such as reduction, such as summing up elements of an array or matrix, taking their maximum, minimum, bitwise logical OR/AND/XOR, etc. The network is organized as a tree, in which each tree node is a node of processors. To sum up elements of an array, the summation starts at the leaves of the tree. A node collects the sum from its children nodes, accumulates the value to its own sum, and propagates it to its parent. Since children nodes can perform the summation in parallel, the total latency for such an operation scales as $\Theta(log\ p)$. The collective network also supports broadcast of data starting from the root to all nodes going down the tree. This can be done in parallel with regular data communication across the torus. Each link in the collective tree network has a target bandwidth of 2.8 Gb/s so it

takes only about 16 cycles to transmit a 8-byte value (excluding the latency at the switch).

The barrier network is a separate network that supports global barrier. The barrier network is also organized as a tree. To perform a global barrier, a node collects signals from its children nodes, and propagate the signal up to its parent node, and this is repeated until the root node gets the signal. When the root node gets a signal, it knows that all nodes have arrived at the barrier. Next, it needs to release the nodes from the barrier. To achieve that, it sends a release signal down the tree to all nodes. The signals that are sent around are processed by special hardware logic at the barrier network controller of each node. Hence, they can propagate very fast. The round-trip latency of a barrier over 64K nodes is reported to take only 1.5 μs, corresponding to only a few hundred 700-MHz processor cycles, which is very fast compared to any software implementation (which easily takes that much time or more for just a few nodes).

9.3 Exercise Problems

1. *(20 Points)* **Lock Performance.**

Consider a four-processor bus-based multiprocessor using the Illinois MESI protocol. Each processor executes a `test-and-test&set` or a `LL/SC` lock to gain access to a null critical section. The initial condition is such that processor 1 has the lock and processors 2, 3, and 4 are spinning on their caches waiting for the lock to be released. Every processor gets the lock once and then exits the program.

The codes for `test-and-test&set` lock and unlock implementation:

```
lock: ld  R, L          // R = &L
      bnz R, lock      // if (R != 0) jump to lock
      t&s R, L         // R = &L; if (R == 0) L=1
      bnz R, lock      // if (R != 0) jump to lock
      ret

unlock: st L, #0       // write ``0'' to &L
        ret
```

Thus, the lock primitive has only two memory transactions: BusRd generated by the `ld` instruction, and a BusRdX generated by the `t&s` instruction.

The codes for `LL/SC` lock and unlock implementation:

```
lock: ll   R, L          // R = &L
      bnz  R, lock       // if (R != 0) jump to lock
      sc   L, #1         // L=1 conditionally
      beqz lock          // if SC fails, jump to lock
      ret

unlock: st L, #0       // write ``0'' to &L
        ret
```

Thus, the lock primitive has only two memory transactions: BusRd generated by the `ld` instruction, and a BusRdX generated by a successful `sc` instruction (and no bus transaction if `sc` fails).

Considering only the bus transactions related to lock-unlock operations:

(a) What is the least number of transactions executed to get from the initial to the final state for test-and-test&set and LL/SC?

(b) What is the worst-case number of transactions for test-and-test&set and LL/SC?

For each case, use a table with the following format. For example, the best case happens when there is no lock contention (no simultaneous lock acquire). The first few lines of the answer are shown:

Bus	Action	P1	P2	P3	P4	Comment
	Initially	S	S	S	S	Initially, P1 holds the lock
1	st1	M	I	I	I	P1 releases lock
2	ld2	S	S	I	I	P2 read misses after invalidation
3	t&s2	I	M	I	I	P2 executes t&s and issues BusRdX
	st2	I	M	I	I	P2 releases lock
5	and so on ...					

2. (*20 Points*) **Lock Performance**.

The code that implements test-and-test&set lock and unlock operations is as follows:

```
lock: ld  R, L          // R = &L
      bnz R, lock       // if (R != 0) jump to lock
      t&s R, L          // R = &L; if (R == 0) L=1
      bnz R, lock       // if (R != 0) jump to lock
      ret

unlock: st L, #0        // write ``0'' to &L
        ret
```

Thus, the lock primitive has only two memory transactions: BusRd generated by the ld instruction, and a BusRdX generated by the t&s instruction.

The code that implements LL/SC lock and unlock operations is as follows:

```
lock: ll   R, L         // R = &L
      bnz  R, lock      // if (R != 0) jump to lock
      sc   L, #1        // L=1 conditionally
      beqz lock         // if SC fails, jump to lock
      ret
```

```
unlock: st L, #0      // write ``0'' to &L
        ret
```

Thus, the lock primitive has only two memory transactions: BusRd generated by the `ld` instruction, and a BusRdX generated by a successful `sc` instruction (and no bus transaction if `sc` fails).

Consider a three-processor bus-based multiprocessor using the Illinois MESI protocol. Each processor executes a `test-and-test&set` or a `LL/SC` lock to gain access to a null critical section. Consider the following lock-/unlock sequence:

- Initially: P1 holds the lock
- P2 and P3 read the lock
- P1 releases the lock
- P2 and P3 read the lock
- P2 acquires the lock successfully
- P3 attempts to acquire the lock but is unsuccessful
- P2 releases the lock
- P3 reads the lock
- P3 acquires the lock successfully
- P1 and P2 read the lock
- P3 releases the lock

Considering only the bus transactions related to lock-unlock operations:

(a) Show the states of each cache for the sequence assuming the test-and-test&set lock implementation. Use the following template. Bus transactions corresponding to the first three steps are shown.

Bus Trans.	Action	P1	P2	P3	Comment
-	Initially	M	I	I	Initially, P1 holds the lock
1 (BusRd)	`ld2`	S	S	I	P2 read misses on the lock
2 (BusRd)	`ld3`	S	S	S	P3 read misses on the lock
3 (BusUpgr)	`st1`	M	I	I	P1 releases the lock
4	and so on ...				

(b) Show the states of each cache for the sequence assuming the LL/SC lock implementation. Use the following template. Bus transactions corresponding to the first three steps are shown.

Bus Trans.	Action	P1	P2	P3	Comment
-	Initially	M	I	I	Initially, P1 holds the lock
1 (BusRd)	`ld2`	S	S	I	P2 read misses on the lock
2 (BusRd)	`ld3`	S	S	S	P3 read misses on the lock
3 (BusUpgr)	`st1`	M	I	I	P1 releases the lock
4	and so on ...				

3. (*15 Points*) **Using LL/SC**

Use LL/SC instructions to construct other atomic operations listed, and show the resulting assembly code segments.

(a) `fetch&no-op L` performs an atomic sequence of reading the value in the location `L` and storing it back into the same location `L`.

(b) `fetch&inc L` performs an atomic sequence of reading the value in the location `L`, increment the value by one, and write the new value to `L`.

(c) `atomic_exch R, L` performs an atomic sequence of swapping or exchanging the value held in register `R` and location `L`.

4. (*10 Points*) **Implementing Locks**

Implement `lock()` and `unlock()` directly using the atomic exchange instruction. The instruction `atomic_exch R, L` performs an atomic sequence of swapping/exchanging the value held in register `R` and location `L`. Use the following convention: a value of "1" indicates that a lock is currently held by a process, and a value of "0" indicates that a lock is free. Your implementation should not repeatedly generate bus traffic when a lock is continuously held by another process.

5. (*10 Points*) **Barrier Implementation.**

A proposed solution for implementing the barrier is the following:

```
BARRIER (Var B: BarVariable, N: integer)
{
  if (fetch&add(B,1) == N-1)
    B = 0;
```

```
    else
      while (B != 0) {};    // spin until B is zero
  }
```

Describe a correctness problem with the barrier implementation. Then, rewrite the code for `BARRIER()` in a way that avoids the correctness problem.

6. **Read and Write Lock**. Show the machine code for implementing a read and write lock using the LL/SC primitives.

7. **Combining Tree Barrier**. Write the pseudo code for a combining tree implementation, assuming that the tree is binary (each node receives barrier completion signals from two "children" nodes, except the leaf nodes).

Chapter 10

Memory Consistency Models

Contents

10.1 Programmers' Intuition **284**

10.2 Architecture Mechanisms for Ensuring Sequential Consistency **290**

10.2.1 Basic SC Implementation on a Bus-Based Multiprocessor 290

10.2.2 Techniques to Improve SC Performance 293

10.3 Relaxed Consistency Models **296**

10.3.1 Safety Net 297

10.3.2 Processor Consistency 297

10.3.3 Weak Ordering 299

10.3.4 Release Consistency 302

10.3.5 Lazy Release Consistency 307

10.4 Synchronization in Different Memory Consistency Models . **309**

10.5 Exercise Problems **314**

As mentioned in Chapter 7, one important problem that multiprocessor systems must address is the *memory consistency* problem, which deals with how accesses (loads and stores) to any memory address are ordered with respect to one another as seen by all processors. Note that this is not a problem that is covered by the cache coherence protocol, since a cache coherence protocol only deals with how accesses to a single memory block address are ordered. Accesses to different addresses are not the concern of cache coherence protocols. In fact, unlike the cache coherence problem that only occurs for systems with caches, the memory

consistency problem exists on systems with and without caches, although caches may exacerbate it.

The objective of this chapter is to discuss the memory consistency problem in more detail, what solutions are offered in current systems, and how they relate to performance. We will begin the discussion with programmers' intuition regarding the ordering of memory accesses. We will show that they in general conform to *full ordering* of memory accesses, and the model that corresponds to such ordering requirements is referred to as the **sequential consistency** (SC) model [1]. We will then discuss how SC can be provided in a multiprocessor system: which parts of the processor or coherence protocol mechanisms must be changed. We will discuss that such changes are too restrictive and likely produce a high performance overhead. As a result, many processors do not implement SC. Rather, they implement more relaxed consistency models, such as **processor consistency**, **weak ordering**, and **release consistency**. We will discuss each of them and show how performance is impacted by different consistency models.

10.1 Programmers' Intuition

Our discussion of memory consistency models start out with inferring what intuition programmers *implicitly* assume with regards to the ordering of memory operations. Consider a pair of post-wait synchronization. Readers may recall that this type of synchronization is needed for DOACROSS and DOPIPE parallelism, among others. Suppose that thread P0 produces a datum to be consumed by thread P1. Thread P0 executes the post in order to let thread P1 know that the datum is ready. Thread P1 executes a wait which makes it block until the corresponding post is executed. To implement the post-wait synchronization, a shared variable is initialized to "0". For the *post* operation, the variable value is set to "1", while for the *wait* operation, a loop waits until the value of the variable is set to "1". This is illustrated in Code 10.1. In the illustration, the shared variable that implements the post-wait pair is `datumIsReady`.

Code 10.1 Example of the use of post-wait synchronization.

P0:	**P1:**
`1 S1: datum = 5;`	`1 S3: while (!datumIsReady) {};`
`2 S2: datumIsReady = 1;`	`2 S4: ... = datum;`

[1] Note that this is not related to the store conditional instruction discussed in Chapter 9, for which we also abbreviate as SC.

In the code above, it is clear that programmers' intention is for the synchronization operation to function properly, that is, for thread P1 to read a value of 5 from `datum`. Programmers do not expect to get a different value from `datum`. However, to get a value of 5 on the read from `datum`, the necessary requirement is that at P0, S1 must be performed before S2. If S2 is performed before S1, then `datumIsReady` is set to 1 before `datum` is set to 5, possibly leading to S4 reading a value of `datum` that is different than 5. Similarly, to get a value of 5 on the read from `datum`, it is also a necessary requirement that S3 is performed before S4, since otherwise, S4 may read a stale value of `datum`. Therefore, *implicitly, programmers expect the order in which memory accesses are executed in a thread to follow the order in which they occur in the source code.* In other words, programmers expect the order of memory accesses expressed in the source code to be obeyed as the source code is compiled into binary code, and as the binary code is executed by the machine. We will refer to such an expectation as the **program order** expectation.

However, the program order expectation is not the only implicit expectation made by programmers. Consider the following code (Code 10.2).

Code 10.2 The illustration of the programmers' atomicity expectation. Initially, the values of x, y, z, xReady, and xyReady are zero.

P0:

```
S1: x = 5;
S2: xReady = 1;
```

P1:

```
S3: while (!xReady);
S4: y = x + 4;
S5: xyReady = 1;
```

P2:

```
S6: while (!xyReady);
S7: z = x * y;
```

The code makes use of two post-wait pairs of synchronization, one pair between P0 and P1, and another between P1 and P2. The code shows that P0 sets `x` to a value of 5, then sets `xReady` to 1 to tell P1 that `x` now has a new value. P1 waits for the value, uses it to sets a new value to `y` (which should be $5 + 4 = 9$). P1 then sets `xyReady` to 1 to indicate that both `x` and `y` have new values. P2 waits until `xyReady` has a value of 1, and then proceeds to read the value of `x` and `y`.

In the code above, it is clear that the programmers' expectation is for the statement S7 to yield a multiplication result of 45 ($= 5 \times 9$) as a value assigned to `z`. They expect that first, the value of `x` is produced by P0, then the value of `y` is produced by P1, and finally both values of `x` and `y` are consumed by P2. However, such an expectation actually **implicitly** assumes the **atomicity** of memory accesses, i.e. an expectation that each memory access occurs in an instant, with-

out being overlapped with other memory accesses. In the example, the expectation assumes that when P0 writes to x, the write is instantly propagated to both P1 and P2. If the write to x is not propagated to P2 instantly, then P2 may read an old value of x. Consider, for example, somehow the write propagation from P0 to P2 is not instantaneous, as illustrated in Figure 10.1. Suppose that all processors execute all memory accesses in program order. P0 executes S1, which writes a value of 5 to x. That write is propagated to P1 and P2, however it reaches P1 faster (Circle 1) than P2 (Circle 3). Then, P0 performs a write to xReady, which is also propagated to P1. Suppose that P1 now sees the new values of xReady and x (in an invalidation-based protocol, P1 has received invalidations and has reloaded blocks containing xReady and x through cache misses). So P1 executes its next statement, assigning a value of 9 to y, and then sets xyReady to 1. Suppose that those writes are propagated to P2 (through invalidations and subsequent cache misses in an invalidation-based protocol). P2 then gets out of its loop and reads the value of x and y. Note that here, P2 reads a fresh value of y since the write to y by P1 has been fully propagated. However, it reads a stale value of x since the write to x by P0 has not been propagated to it yet (i.e., in an invalidation protocol, the invalidation has not reached P2's cache yet). Therefore, P2 reads the old value of x, which is 0, and assigns a wrong value of 0 to z.

Figure 10.1: Illustration of the violation of store atomicity.

How can an invalidation sent out by P0, corresponding to the write of x, reach P1 but not immediately P2? The answer is that this can happen in some types of interconnection networks. On a bus-based multiprocessor, the write will reach the bus, and all snoopers (P1's and P2's) will see the write and invalidate their corresponding blocks in the caches. So, such a situation would not occur. However, if the multiprocessors are interconnected using a point-to-point interconnection,

separate invalidation messages are sent from P0 to P1 and P2. It is entirely possible for the invalidation message sent to P1 to arrive earlier than the invalidation message sent to P2, due to the messages going to different routes, or due to the traffic that delays the delivery of the invalidation going to P2.

Let us get back to the discussion on programmers' expectation. In the code shown above, it is clearly the case that programmers expect the code to function properly, which means that write propagation must happen instantly, or in technical terms, **atomically**. The write by P0 must happen in such a way that it is propagated at once, in a non-divisible step, to both P1 and P2. Thus, by the time P1 gets out of the loop, the write to `x` would have fully propagated to all processors.

Overall, we can express programmers' implicit expectation of memory access ordering as: *memory accesses coming out of a processor should be performed in program order, and each of them should be performed atomically.* Such an expectation was formally defined as the **Sequential Consistency** (SC) model. The following is Lamport [Lamport, 1979]'s definition of sequential consistency:

> A multiprocessor is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program.

Let us look at a few more examples to see what results are possible in SC for various cases. Consider the following code (Code 10.3). The code for P0 consists of two writes to `a` and `b`. The code for P1 consists of two reads from `b` and from `a`. The code contains no synchronization between the two threads, which means that the outcome of the program is **non-deterministic**. Suppose that the system on which the threads run ensure SC. One possible outcome is when S1 and S2 occur before S3 and S4, in which case S3 and S4 will read a value of 1 for both `a` and `b`. Another possible outcome is when S3 and S4 both occur before S1 and S2, in which case S3 and S4 will read old values of `a` and `b`, which is zero. Another possible outcome arises when S1 is performed first, followed by S3 and S4, then S2. In such a case, S3 will print the old value of `b`, which is zero, while S4 prints the new value of `a`, which is 1. However, under SC, it is impossible for S3 to read out a new value of `b`, which is "1", and for S4 to read out the old value of `a`, which is "0". Overall, the final values of `(a, b)` that are possible under SC are (0, 0), (1, 0), and (1, 1), while the final value of (1, 0) is not possible under SC.

Code 10.3 An example illustrating SC and non-SC outcomes. Initially, the values of a and b are both zero.

P0:	**P1:**
`S1: a = 1;`	`S3: ... = b;`
`S2: b = 1;`	`S4: ... = a;`

To reason about why the final outcome is impossible in SC, note that SC defines program order and atomicity of memory accesses. Let us denote the ordering of memory accesses to be represented by the "→" symbol. In SC, we know that the ordering S1 → S2 and S3 → S4 will be enforced. For statement S3 to read out a new value of b, S3 must have been performed after S2, i.e., S2 → S3. But since S1 → S2 and S3 → S4, it can be concluded that S1 → S2 → S3 → S4. On the other hand, for statement S4 to read out the old value of a, S4 must have been performed before S1, i.e. S4 → S1. But since S1 → S2 and S3 → S4, it can be concluded that S3 → S4 → S1 → S2. However, both conclusions cannot be true simultaneously as they lead to a contradiction. Thus, the result is impossible in a system that guarantees SC.

However, on a system that does *not* enforce SC, this final outcome becomes possible. For example, suppose that the program order between S3 and S4 is no longer ensured. Then it is possible for S4 to be performed earlier than S3. In such a case, we may have S4 performed first (resulting in reading a value of "0" for a), followed by S1 and S2, and followed lastly by S3 (resulting in reading a value of "1" for b).

Now let us consider another example in the following code (Code 10.4). The code for P0 consists of a write to a followed by a read from b. The code for P1 consists of a write to b followed by a read from a. The code contains no synchronization between the two threads, which means that the outcome of the program is **non-deterministic**. Suppose that the system on which the threads run ensure SC. One possible outcome is when S1 and S2 occur before S3 and S4, in which case S2 will read out 0, while S4 will read out 1 (the new value produced by S1). Another possible outcome is when S3 and S4 both occur before S1 and S2, in which case S2 will read out 1 (the new value produced by S3), while S4 will read out 0. Another possible outcome arises when S1 is performed first, followed by S3, then by S2, and finally by S4. In such a case, S2 will read out 1 (the new value of b produced by S3), and S4 will read out 1 (the new value of a produced by S1). However, the final case in which both S2 and S4 read out a "0" is impossible under SC. Thus, the final results of (a, b) may be (1, 0), (0, 1), or (1, 1) under SC,

but it cannot be (0, 0).

Code 10.4 A code example illustrating SC and non-SC outcomes. Initially, the values of a and b are both zero.

P0:	**P1:**
`1S1: a = 1;`	`1S3: b = 1;`
`2S2: ... = b;`	`2S4: ... = a;`

To reason about why this final outcome is impossible in SC, recall that SC defines program order and atomicity of memory accesses. Hence, we know that the ordering S1 → S2 and S3 → S4 will be enforced. For statement S2 to read out a value of "0", S2 must have been performed before S3, i.e. S2 → S3. But since S1 → S2 and S3 → S4, it can be concluded that S1 → S2 → S3 → S4. For statement S4 to read out a value of "0", S4 must have been performed before S1, i.e. S4 → S1. But since S1 → S2 and S3 → S4, it can be concluded that S3 → S4 → S1 → S2. Both conclusions cannot be true simultaneously since one contradicts another. Therefore, the result is impossible in a system that guarantees SC.

However, on a system that does *not* enforce SC, this final outcome is possible. For example, suppose that the program order between S3 and S4 is no longer ensured. Then it is possible for S4 to be performed earlier than S3. In such a case, we may have S4 performed first (resulting in reading a zero for a), followed by S1 and S2 which reads out zero for b, and followed lastly by S3.

Both Code 10.3 and 10.4 are unlikely to be encountered in a majority of programs because of the non-deterministic outcome of the programs. Non-deterministic programs are notoriously difficult to reason for correctness and to debug. However, sometimes they are a legitimate class of programs, such as when the computation can tolerate non-determinism and imprecision (Code 3.14 and Code 3.16 in Section 3.3 are some examples of codes that can tolerate non-deterministic results). Non-deterministic programs can also arise when programmers mistakenly omit synchronization operations, in which case the non-determinism occurs by accident. One relevant and important question is whether programs that have non-deterministic outcomes can tolerate the additional non-determinism that arises from machines that do not ensure SC. The likely answer would be yes for programs that have intentional non-determinism, but it is less clear for programs in which the non-determinism occurs due to accidental omission of synchronization (debugging such programs becomes harder in non-SC machines).

Still, for a large majority of parallel programs, deterministic program output is the case. Therefore, we need to keep in mind when discussing memory consistency models that SC results must be ensured for deterministic programs that have proper synchronization, while also keeping in mind that non-SC results may be tolerable for programmers in non-deterministic programs.

10.2 Architecture Mechanisms for Ensuring Sequential Consistency

From the previous discussion, we have established that in order to ensure SC, memory accesses from a processor must follow program order and each of them must be atomic. In this section, we will see how that can be achieved in a real implementation. First, we will discuss a basic implementation that works. However, while it works, it suffers significantly in terms of performance due to the lack of overlapping of memory accesses. Then, we will discuss techniques that can improve the performance of SC by allowing memory operations to be overlapped with respect to one another. Note that overlapping memory accesses can potentially violate their atomicity. Therefore, a safety mechanism is needed to detect atomicity violation and recover from it. We will discuss such a safety mechanism and how it is implemented in real processors.

10.2.1 Basic SC Implementation on a Bus-Based Multiprocessor

In the basic implementation of SC, memory accesses must be performed atomically with respect to one another. The simplest and correct way to achieve atomicity is to perform one memory access at a time without overlapping them. To perform memory accesses one at a time, we need to know when a memory access starts and ends. Consider a load instruction. A load instruction is logically executed in four steps. First, its effective address is computed in a functional unit. Then, a cache access to the effective address is issued to the memory hierarchy. The memory hierarchy finds the data value associated with that address (which may be in the cache, in memory, or in a remote cache). Finally, it returns the value to the destination register of the load instruction. Note that the first step (computation of effective address) is not affected by other memory accesses and will work the same way regardless of program order, hence this step can be overlapped with any steps of other memory accesses. The final step (returning the value that

has been read from the cache to the destination register of the load) is also unaffected by other memory accesses since the value has already been obtained from the cache, and hence it can also be overlapped with any steps from other memory accesses. Therefore, from the perspective of memory access ordering, a load starts when its access is issued to the cache and completes when the value is obtained from the cache. These two steps are the ones that must be performed atomically.

Similarly, a store instruction is executed in several steps. First, its effective address is computed in a functional unit. Then, if it is no longer *speculative* (e.g., it is not an instruction in the wrong path of a branch and is free of exception), the store can be committed or retired (readers are referred to textbooks that describe mechanisms for out-of-order execution for more details). It is retired by copying its destination address and value to be written to a structure called the **write buffer**, which is a first-in-first-out (FIFO) queue containing stores that have been retired from the processor pipeline. Later, the store value in the write buffer is released to the cache. The store directly modifies the cached copy if it is the sole copy in the system, or it sends invalidations to other caches if there are other copies. A store is considered complete when it has fully propagated to other processors. In an update protocol, that corresponds to updating all other cached copies. In an invalidation protocol, that corresponds to having invalidated all other copies. Note that the first step (computation of effective address) is not affected by other memory accesses and will work the same way regardless of program order, hence this step can be overlapped with any steps of other memory accesses. Also, note an important distinction between a store and a load is that the cache access occurs very late in a store instruction (after the store is retired) whereas in a load instruction the cache access occurs early (before the load can be retired). Another important distinction is that whereas a load instruction only involves one processor, a store instruction involves invalidating multiple caches. Hence, from the perspective of memory access ordering, a store starts when its effective address is issued, and is complete by the time it has been performed globally (i.e., it has finished propagating its value to all processors).

How can the completion of a store be detected? In a basic implementation, one way to detect the completion of the store is to require all sharers to acknowledge the invalidation, and for the store originator to collect all invalidation acknowledgments. In a bus-based multiprocessor, however, simply posting the read exclusive request on the bus ensures that all snoopers will see the store. If we assume that once a snooper sees the request, it *atomically* or *instantaneously* invalidates

the cached copy at the node in which the snooper is located, then a store can be considered complete when the read exclusive request reaches the bus. However, in a system that does not rely on broadcast and snooping, it is necessary for the requestor to obtain all invalidation acknowledgment messages from all sharers before it can conclude that the store is complete. Note that in the case that there are no other shared copies (e.g., the block is already exclusively cached when the store is issued to the cache), the store is considered complete right after it accesses the cache to update the value in the cached copy. No write propagation is necessary since there are no sharers.

Now that we have established that a load is complete when it has obtained its value from the cache, and that a store is complete when its read exclusive request has reached the bus, we will illustrate how SC can be enforced in a bus-based multiprocessor. Suppose that we have the following code (Code 10.5), showing five memory accesses coming out of a processor. In the basic implementation, each memory access must complete before the next one (in program order) can start. Hence, before the first load is complete, the second load cannot be issued to the cache. Consider the case in which the first load suffers a cache miss while the second load will have a cache hit if it is allowed to go to the cache. Due to the strict ordering, the second load cannot be allowed to issue to the cache even though the block containing B is in the cache. The second load must wait until the first load finishes its cache miss and obtains the value from the cache.

Code 10.5 Code example illustrating basic SC mechanisms.

```
S1: ld R1, A
S2: ld R2, B
S3: st C, R3
S4: st D, R4
S5: ld R5, D
```

In the same way, the store in S3 cannot be allowed to access the cache or obtain bus access until the load in S2 has completed (obtained the value from the cache). The store in S4 cannot be allowed to access the cache until the store in S3 has generated a request on the bus. Finally, the load in S5 cannot access the cache until the store in S4 has generated a request on the bus, even when the store and the load are to the same address. Bypassing a store value to a younger and dependent load is not allowed in SC as the atomicity of the store is violated.

In the basic implementation, SC imposes a huge performance restriction as it

does not allow memory accesses to be overlapped or reordered. Many techniques in out-of-order execution of instructions which have been shown to be beneficial for exploiting *instruction level parallelism* (ILP), must be disabled. For example, a write buffer allows a store to be retired, and the following load to complete even before the just-retired store reaches the cache. This must be disabled because the load cannot be issued to the cache until the store has obtained a bus access. Second, two loads cannot be issued to the cache at the same time. A modern cache allows **non-blocking** accesses, in that it can serve a cache access while cache misses from older accesses have not completed. Enforcing SC means that a non-blocking cache loses its effectiveness. Finally, compiler optimizations that reorder memory accesses in the compiled code are not allowable either. Hence, the performance of an SC machine is significantly restricted.

10.2.2 Techniques to Improve SC Performance

We have discussed how a basic SC implementation can be very restrictive in terms of performance. In this section, we will describe what optimizations are possible in order to improve performance while still preserving SC semantics. The key to improving the performance of an SC implementation is to make the execution of memory accesses faster and to allow memory accesses to be overlapped with respect to one another. However, some of these overlaps may violate the atomicity of the accesses. Therefore, a safety mechanism is needed to detect atomicity violation and recover from it.

The first performance enhancement technique that we will discuss avoids overlapping of the execution of loads and stores, but overlaps data fetches that are generated by these loads and stores. The idea is to make the time to complete a load or store as short as possible. For example, a load that hits in the cache will complete sooner than a load that misses in the cache because it obtains the data from the cache faster than from the main memory. Similarly, a store that finds the block in the cache in an exclusive/modified state does not need to go to the bus since we are sure that there are no other sharers in the system, thus there is no need for an invalidation. In such a case, the store only needs to write to the cached copy and it completes right away.

How can we maximize the probability that a load hits in the cache and a store finds the block to be written in the cache in an exclusive or modified state? By issuing prefetch requests as soon as the effective addresses of loads and stores are available or can be predicted. For example, when a load's effective address is

generated, we can issue a prefetch to the cache even when there are older loads or stores that have not completed. In fact, with a prefetching engine, if the addresses involved in the loads can be predicted, they can be prefetched even before the load instructions are fetched or executed. When the older loads or stores are completed, the load which has already issued a prefetch can access the cache and completes sooner due to the cache hit. Similarly, when a store's effective address is generated, we can issue a prefetch exclusive (essentially an upgrade or a read exclusive request on the bus), even when there are older stores or loads that have not completed. When older loads or stores have completed, and if the prefetch has completed too, the store can access the cache and completes right away without going to the bus.

Note that the prefetches do not obviate the need for loads and stores to access the cache. The loads or stores still must access the cache in a non-overlapped way to guarantee atomicity, however the probability that they find the blocks they need in the cache in the proper state improves with prefetching. Note, however, that prefetching may not always improve performance. One reason is that it is possible for a block that has been prefetched to be stolen before the load or store that generates the prefetching accesses the cache. For example, after a block is prefetched into the cache, it may be invalidated by another processor. Or after a block is prefetched in an exclusive state into the cache, another processor may read from the block, necessitating it to respond with the data and to downgrade its state to shared. In such cases, the prefetches have been useless as the corresponding store still needs to upgrade the block to an exclusive state. In fact, they have not only been useless, but have also been harmful as they incur unnecessary traffic and delay at other caches. Hence, while prefetching helps improve the performance of an SC implementation, it is not a perfect solution.

The second technique for improving the performance of an SC implementation is to rely on speculative accesses. We can overlap the execution of a load with the execution of an older load by speculatively assuming that the older load executes atomically. If we find that the older load has not really executed atomically, then we have wrongly allowed the younger load to proceed without waiting for the older load to complete. In this case, we have to cancel the younger load and re-execute it. Note that the ability to cancel an instruction and re-execute it is already provided by most out-of-order processors to provide a *precise interrupt mechanism*. Hence, all we need to add is the detection on when the atomicity of a load is broken.

To achieve that, we can recall that there is a similar technique for providing an illusion of atomicity between two memory accesses: the load linked (LL) and the store conditional (SC) instructions, used for implementing a lock synchronization. In the LL/SC lock implementation (Section 9.1.5), the load linked instruction appears to have executed atomically if by the time the store conditional instruction is executed, the block that was read by the load linked instruction has not been stolen from the cache. If the block is invalidated before an SC is attempted, then the SC fails and the LL is re-executed. Or if the program is context switched before the SC is attempted, the SC also fails and the LL is re-executed.

We can overlap the execution of two loads using a similar mechanism. Suppose we have two loads to different addresses coming from a processor. The first load suffers a cache miss. Under the basic SC implementation (even with the use of prefetching) the second load cannot access the cache until the first load has obtained its data from the cache. With speculation, we can indeed allow the second (the younger) load to access the cache. However, we should mark the second load as speculative. If the block read by the second load has not been stolen (invalidated or naturally evicted from the cache) by the time the first load completes, then the value obtained by the second load would have been the same if it had waited to issue to the cache after the first load completed. Therefore, the second load has executed as if the first load executed atomically. In this case, the speculation is successful and we have overlapped the execution of two loads. The second load can now be marked non-speculative from the point of view of memory access ordering (it may still be speculative in some other sense such as with regard to an incorrect branch path). However, if the block read by the second load has been invalidated (or evicted from the cache) by the time the first load completes, then the illusion of atomicity has been broken. The second load, if it is issued to the cache now, may obtain a different value compared to when it speculatively accessed the cache. Hence, the speculation has failed and the second load must be re-executed. The miss-speculation incurs a performance penalty for having to flush the pipeline and re-execute a load and all instructions that are younger than the miss-speculated load.

Similarly, a younger load can be speculative with respect to an older store that is not yet completed. When the older store is completed, the younger load must be re-executed if its block has been stolen or replaced from the cache. This allows the use of a write buffer in a system that enforces SC.

The discussion on speculative execution primarily deals with speculative loads.

Applying speculation to a store instruction is more difficult. The reason is that while a load can be canceled and re-executed easily, a store cannot be canceled easily. A store, after accessing the cache, has already deposited a new value in the cache and potentially has propagated the value to the main memory and other processors. Hence, canceling a store is a much harder problem. Consequently, speculative execution is typically only implemented for loads. Stores are issued to the cache one at a time.

Overall, we have discussed two very powerful mechanisms for improving the performance of an SC implementation. The first technique (prefetching) does not allow overlapping of memory accesses, but the atomic execution of each memory access is likely to be shorter than without prefetching. The second technique (speculative load accesses) genuinely allows overlapping of memory accesses (loads). Both techniques are implemented in the MIPS R10000 processor, which supports SC, and even in the Intel Pentium architecture, which supports a memory consistency model that is slightly weaker than SC. Both techniques are highly effective in making SC performing better. However, there is still a fundamental performance problem that cannot be dealt with by prefetching or speculative execution: the compiler still cannot reorder memory accesses when compiling the program. Only relaxed memory consistency models allow compilers to reorder memory accesses.

10.3 Relaxed Consistency Models

We have discussed that programmers' intuition largely follows SC, what requirements of execution are needed to guarantee SC, and implementations of SC on real systems. We have also discussed that SC is restrictive in terms of what compiler optimizations are allowed. In this section, we will discuss various memory consistency models that relax the memory access ordering restriction of SC. In general, they perform better than SC, but impose an extra burden on the programmer to ensure that their programs conform to the consistency model that the hardware provides.

Relaxing memory access ordering may allow execution that deviates from programmers' intuition. Therefore, typically a safety net is provided in order for programmers to specify strict ordering between a pair of memory accesses. We will first discuss the safety net before discussing relaxed consistency models. Then, we will discuss several relaxed consistency models, focusing on three variants called

the **processor consistency** (PC) model, **weak ordering** (WO) model, and **release consistency** (RC) model. There are many more variants implemented in a real system, but PC, WO, and RC provide good representations for them.

10.3.1 Safety Net

The safety net for enforcing strict ordering between two memory accesses typically comes in the form of **fence** instructions (also referred to as **memory barrier** instructions). The semantics of a fence instruction is that *the fence instruction prohibits the execution of the memory accesses following the fence until all memory accesses preceding the fence have completed.* Sometimes the fence only applies to stores, in which case it imposes ordering only between stores that precede it and stores that follow it, and it is referred to as **store fence/barrier**. Sometimes the fence only applies to loads, in which case it imposes ordering only between loads that precede it and loads that follow it, and it is referred to as **load fence/barrier**.

In a real implementation, the fence requires the following mechanism. When a fence is encountered, the memory accesses following the fence are canceled by flushing them from the pipeline (or avoiding fetching them if they are not yet fetched), and all preceding memory accesses are completed. That is, loads must obtain data from the cache, and stores must access the cache and generate bus requests. Once all memory accesses older than the fence complete, the processor state is restored to the state prior to the fence instruction, and execution is resumed from that point.

It is the responsibility of programmers to insert fence instructions in their code for correct execution. Not inserting fence instructions when they are needed may result in incorrect execution and non-deterministic outcome. Inserting fence instructions when they are not needed results in an unnecessary performance degradation.

10.3.2 Processor Consistency

Recall that in SC, each load/store is ordered with regard to the following load/store. In other words, the ordering of the following pairs is enforced: load → load, load → store, store → load, and store → store. In the Processor Consistency (PC) model, the ordering between an older store and a younger load (store → load) is relaxed. When a store has not completed, a younger load is allowed to issue to the cache and even complete. The significance of this is that store instructions can be queued in the write buffer, and be completed at a later time without the use of

load speculation. In the mean time, loads do not need to wait for the older stores to complete and can access the cache, thereby reducing the load latency.

Using write buffers is no longer a problem in a system that supports PC. Note that the ordering between stores is still enforced, which means that the order in which stores access the cache must follow program order. Hence, the optimization technique used in the SC system that relies on exclusive prefetching is also applicable in PC, although its importance is somewhat diminished because younger loads can issue early despite the completion time of older stores. In addition, the ordering between loads is also enforced. Therefore, prefetching blocks early is also still applicable in PC as it is in SC. One distinction between PC and SC is that when a load is issued to the cache and bypasses an older store, it is not treated as speculative because the reordering is allowed by the PC model, whereas in SC the load is speculative and must be rolled back if the illusion of atomicity is broken. Therefore, recovery techniques for loads that issue to the cache early are no longer needed, resulting in fewer pipeline flush and recovery, and better performance.

How closely does PC correspond with natural programmers' intuition? The answer is quite well. Note that PC only relaxes one ordering constraint out of four, hence it only affects the correctness of code that has a store followed by a load. The behavior of the regular post-wait synchronization pair shown in Code 10.1 in which the producer sets `datum` and `datumIsReady` is not changed because the ordering between two stores is still enforced. In addition, the behavior of the consumer which reads `datumIsReady` and `datum` is also unchanged since the ordering between two loads is still enforced.

There are some cases in which PC yields different results compared to the SC model. For example, consider Code 10.4, in which each processor executes a code that consists of a store followed by a load. Since PC does not guarantee ordering between a store and a younger load, non-SC outcome can result in this case, yielding an outcome in which both `a` and `b` have a final value of 0. Note that the example shows code that is not synchronized, and can produce non-deterministic outcome even under SC.

In general, for properly synchronized programs, PC yields the same outcome as SC. Consider a post-wait synchronization in which a processor produces a data value and sets a flag to signal the availability of the new value. This involves at least two store instructions, which are ordered with respect to each other in PC. At the consumer, at least two loads are involved, and their ordering is also guaranteed

under PC. Thus, a post-wait synchronization produces the same outcome in PC as in SC. In a lock synchronization, the lock release operation involves setting a value to a lock variable using a store instruction. Loads that follow the lock-release store are not in the critical section, so their ordering with respect to the lock-release store is not crucial to correctness. Letting younger loads complete before the lock-release store is executed achieves a similar effect as when the lock is released late, which is a correct behavior. A lock acquire operation involves both loads and a store that changes the value held in the lock variable. The lock-acquire store may be followed by loads that are in the critical section, so their ordering is crucial to correctness. However, a lock-acquire store is often implemented either using an atomic instruction or a store conditional instruction that ensures the atomicity of the store. When such an atomic store fails, the lock acquisition is re-attempted, and the execution of any younger loads will be repeated later. When the atomic store succeeds, the lock acquisition is successful, and in this case, the loads that precede the atomic store must have read a value that indicates the lock was free. And this is only possible when no processors were in the critical section. Since all loads are ordered with respect to each other, the loads younger than the atomic store would have obtained the most recent values released by other processors, or when the lock acquire fails, they will be repeated.

10.3.3 Weak Ordering

Another attempt to relax the memory consistency model comes from an observation that a majority of programs are properly synchronized. When programmers want a certain memory access in one thread to occur after another memory access in another thread, programmers will rely on synchronization to achieve it. The synchronization can be in the form of barriers, point-to-point synchronization, etc. For example, a post-wait pair may be used in order to make sure a (consumer) load occurs after a (producer) store has completed. Programs that are not properly synchronized have non-deterministic behavior. For example, without the post-wait synchronization, a consumer load may read a newly produced value or may read an old value, depending on the relative timing of the execution of threads. Since it is rare that programmers tolerate non-deterministic results, we can reasonably assume that most (bug-free) programs are properly synchronized.

If programs are properly synchronized, there is no **data race** that can occur. *Data race* is defined as simultaneous accesses by multiple threads to a single location in which at least one of the access is a write. Data race is a necessary condition

for non-deterministic results to occur. Simultaneous loads do not change the outcome of the loads hence they cannot produce data race. Simultaneous stores may overwrite each other depending on the timing, hence they can induce data race. A simultaneous load and store also may cause the load to return different values depending on the timing of the load and store, causing non-deterministic results. Therefore, in a properly synchronized program, programmers' expectation is that no data race can occur.

The implication of the lack of data race in a properly synchronized program is that ordering of memory accesses can be relaxed except at synchronization points. Since data race cannot occur, *it is safe to reorder memory accesses except at synchronization points*. To illustrate this point, assume that synchronization works as it should, and consider the following example. Suppose that different threads enter a critical section protected by a lock, and exit the critical section to do more computation. Note that if the synchronization works correctly, then only one thread will be in the critical section at a time. Therefore, the ordering of memory accesses inside the critical section does not matter and does not need to follow program order, since we know that only one thread executes in the critical section. Furthermore, the ordering of memory accesses outside of the critical section also does not matter and does not need to follow program order, because if programmers had cared about the relative ordering of memory accesses, they would have inserted synchronization to ensure that. Consequently, outside of synchronization points, program order does not need to be maintained. However, ordering of synchronization accesses is still required to ensure the proper behavior of synchronization.

Such an observation is the basis for a relaxed memory consistency model referred to as the **weak ordering** (WO) model. The WO model uses two assumptions: (1) programs are properly synchronized, and (2) programmers correctly express to the hardware which loads and stores act as synchronization accesses. Based on the assumptions, we can define the correct behavior of a synchronization access: (1) before a synchronization access can issue, all prior loads, stores, and synchronization accesses must have completed; and (2) all loads, stores, and synchronization accesses following it must not have issued. In other words, any pair of accesses in which one occurs prior to the synchronization access and another occurs after the synchronization access, are strictly ordered.

To see why this works, consider the post-wait synchronization pair used for the producer-consumer communication in Code 10.1. If the store to and load from `datumIsReady` are properly identified as synchronization accesses, the code

executes correctly. At the producer, the synchronization access cannot issue until all prior memory accesses are complete, including the store to `datum`. Hence, the store to `datum` has fully propagated before the store to `datumIsReady` is allowed to access the cache. At the consumer, all memory accesses following a synchronization access (the load from `datumIsReady`) cannot issue until the synchronization access is complete. That is, before the load from `data` can issue to the cache, the load from `datumIsReady` must have completed first. Hence, WO has guaranteed that the load from `data` gets a newly produced value.

What mechanism is used in the processor to implement WO? When a synchronization access is encountered at the processor pipeline, first, all accesses that follow it are flushed from the pipeline or not fetched into the pipeline, in effect canceling their execution. Then, the synchronization access itself is stalled until all prior memory accesses have completed. That is, all prior loads must have obtained their values, and all prior stores have emptied from the write buffer and have fully propagated their values (through invalidation).

WO is much more relaxed than SC. Compilers can freely reorder loads and stores as long as they do not cross synchronization boundaries. During execution, the execution of loads and stores can be reordered and overlapped, and there is no need to execute them atomically, as long as ordering is enforced around synchronization accesses. Consequently, WO provides better performance than SC. However, it comes at the expense of requiring synchronization accesses to be both properly identified and expressed to the hardware.

Identifying synchronization accesses is trivial if they use special types of instructions used for synchronization, such as atomic fetch-and-op, test-and-set, load linked, store conditional, etc. The architecture can ensure WO semantics are obeyed at these boundaries. If, however, regular loads and stores are used for synchronization, then programmers must label them, or insert fences to enforce ordering. Both alternatives are shown in Code 10.1. With labeling, the synchronization store (for the *post* operation) is identified, and the synchronization load (for the *wait* operation) is also identified. In some implementations, labeling requires changing the synchronization loads and stores to atomic instructions or instructions that the hardware knows to be clearly used for synchronization.

If fences are used, they can be inserted to wrap the synchronization accesses to ensure their ordering with respect to older and younger memory accesses. For example, the store to and load from `datumIsReady` are synchronization accesses, hence they are wrapped by fences. We will discuss this issue in more details in

Section 10.4.

Code 10.6 Post-wait synchronization with fences for the weak ordering model.

Supporting post-wait synchronization using labeling:

P0:

```
st &datum, #5 // datum=5;
(sync)st &datumIsReady,#1
```

P1:

```
L: (sync) ld R1, &datumIsReady
   sub R1, R1, #1 // R1=R1-1
   bnz R1, L
   ld R2, &datum // R2 = datum;
```

Supporting post-wait synchronization using fences:

P0:

```
S1: datum = 5;
    fence
S2: datumIsReady = 1;
    fence
```

P1:

```
    fence
S3: while (!datumIsReady) {};
    fence
S4: ... = datum;
```

Compared to PC, WO tends to provide better performance due to more relaxed ordering between synchronization points. However, if the size of a critical section is small, or if critical sections are frequently entered and exited, there may be only a few memory accesses between synchronization points. In such a case, only a few memory accesses can be overlapped and there is little opportunity to improve performance from overlapping their execution. When critical sections are small, PC may outperform WO. In PC, a lock release operation or the *post* part of the post-wait synchronization consist of a single store instruction. The store instruction may be followed immediately by loads. With WO, the synchronization store and younger loads will be ordered, whereas in PC they are not ordered. Hence, in this case, PC may achieve a higher degree of overlapping.

10.3.4 Release Consistency

A more aggressive attempt to relax the memory consistency model comes from an observation that there are two types of synchronization accesses: a type that *releases* values for other threads to read, and another type that *acquires* values that are released by another thread. For example, in a post-wait synchronization, the producer thread sets a flag to signal that data is ready. Setting the flag uses a synchronization store that needs to propagate the new flag value and all data values prior to the synchronization store. Hence, the synchronization store acts as a release. The consumer thread reads the flag before it accesses the newly produced data, and hence the flag read acts as an acquire.

Similarly, a lock release operation can be considered as a type of release synchronization, and the lock acquire operation as a type of acquire synchronization. To see why this makes sense, consider the sequence of execution shown as follows (Figure 10.2). The figure shows two threads that want to enter a critical section, read the most recent value of `sum` and accumulate an array element to it, and exit the critical section. The code shows that P0 arrives at the critical section before P1. In order for the code at P1 to execute correctly, it must be the case that when P1 reads from `sum`, it sees the latest value that was produced by P0 during the time P1 executed in its critical section. With the critical section protecting the access to `sum`, only one thread can be in the critical section at a time. Hence, the synchronization itself has ensured that the successful lock acquisition at P1 occurs after the lock release by P0, i.e. S3 → S4. In order for the code to perform correctly, P0 must have fully propagated the new value of `sum` that it wrote in statement S2, by the time it releases the lock, i.e., we must maintain the S2 → S3 ordering. If not, it is possible that when the lock is released and subsequently acquired by another thread, the new value of `sum` has not propagated completely. In addition, P1 must not have read from `sum` before the lock is fully acquired, i.e. we must maintain the S4 → S5 ordering. If not, P1 may read a stale value when it executes the statement S5 in its critical section.

Figure 10.2: Illustration of acquire-release synchronization.

Hence, it is imperative that all memory accesses prior to the lock release have been completed *before* the lock release. In addition, when P1 enters the critical section, it must not have performed any memory accesses in its critical section. Thus, it is imperative that all memory accesses following the lock acquisition to be issued after the lock acquisition is complete.

Note that correctness is not affected if memory accesses following the `unlock`

issue before the `unlock` completes because its ordering effect is similar to delaying the lock release – lengthening the critical section. Hence, while memory accesses prior to the `unlock` must be completed prior to the `unlock`, memory accesses following the `unlock` can issue early without waiting for the `unlock` to complete. If there are data dependences between the memory accesses in the critical section and ones after the critical section, they will be handled correctly by uniprocessor dependence detection and enforcement mechanisms. Hence, to ensure correctness, we can conclude that an `unlock` should prevent *downward migration* of memory accesses but *not upward migration*.

Similarly, correctness is also unaffected if memory accesses prior to a `lock` issue after the lock acquisition is performed. The effect of allowing that is similar to the case in which the lock is acquired early, before it is really needed in the critical section. However, the code inside the critical section cannot be issued before the lock acquisition is complete because mutual exclusion may not be guaranteed. Hence, to ensure correctness, we can conclude that a lock acquisition should prevent *upward migration* of memory accesses but *not downward migration*.

Now, let us consider the post-wait synchronization pair used for the producer-consumer communication in Code 10.1. The requirement for correctness is that the *post* operation should not issue until all prior memory accesses are complete, especially the store to `datum`. Otherwise, thread P1 may find that the *post* has been completed only to read the stale value of `datum`. In addition, the read from `datum` should not be executed until after the *wait* operation is complete, i.e. a value of "1" in the `datumIsReady` is observed. Therefore, S2 acts as a release synchronization since it signals the readiness of values produced up to this point, while S3 acts as an acquire synchronization since it reads values released by another thread. Just like in the lock acquire and release example, here S2 (a release synchronization) must also prevent downward migration, while S3 (an acquire synchronization) must also prevent upward migration. S2 does not need to prevent upward migration, since the effect of executing younger loads/stores early is as if S2 is executed late. Likewise, S3 does not need to prevent downward migration, since the effect of executing older loads/stores late is as if S3 is executed early. Finally, note the programming complexity over WO is that while in WO we only need to distinguish the store to and load from `datumIsReady` as synchronization accesses, in RC, we need to specify that S2 is a release synchronization while S3 is an acquire synchronization.

From the intuition of the correct behavior of `lock` and `unlock`, and *post* and

Figure 10.3: Illustration of an acquire and release in post-wait synchronization.

wait, we can now discuss the formal correctness requirements of an acquire and a release synchronization.

> An acquire synchronization must ensure that no younger load/store executes before the acquire is complete. A release synchronization must ensure that all older loads/stores complete before the release is issued. Acquires and releases must execute atomically with respect to one another.

The final requirement states that acquires and releases must be atomic with respect to each other. This implies that they must appear not to have overlapped their execution with respect of one another. This requirement makes sense. For example, a lock acquisition cannot be overlapped with a lock release to fully guarantee mutual exclusion.

Note that the requirements for acquire and release synchronizations imply that the execution of two critical sections on a thread can be overlapped! For example, suppose that a thread acquires and releases lock A and then acquires and releases lock B. The requirement for the acquisition is to prevent upward migration, and hence all younger loads, stores, and even lock acquisitions can be executed as soon as the current lock acquisition is completed. Figure 10.4 illustrates this case, using code that shows two critical sections and memory accesses inside the first critical section (block 1), between the first and second critical section (block 2), and inside the last critical section (block 3). The lock release of A prevents downward migration, but not upward migration. Hence, the execution of loads/stores from block 2 can be moved up as far as right after the completion of the acquisition of lock A. So can the execution of the acquisition of lock B. As a result,

the execution of loads/stores from block 1, 2, and 3, can actually be overlapped. However, lock acquisitions have to be ordered because they both prevent upward migration, while lock releases also have to be ordered because they both prevent downward migration. The ability to overlap the execution of memory accesses in block 1, 2, and 3 both dynamically (in hardware) and statically (through memory access reordering by the compiler) stands in contrast to WO. In WO, lock acquisitions and lock releases act to order all loads/stores before and after them. Thus, the execution of block 1, 2, and 3 with respect to each other is ordered in WO, and overlap of memory accesses only occurs within each block.

Figure 10.4: Illustration of allowed execution order under RC.

Similar to WO, RC requires all synchronization accesses in a program to be identified correctly and completely, so that the hardware can ensure correct execution of properly synchronized programs. No guarantee is given for programs that are not properly synchronized (e.g. missing synchronization when it is needed), or for programs that are properly synchronized but the synchronization accesses are not properly expressed to the hardware. Furthermore, compared to WO, RC requires more from the programmers. They must label synchronization accesses as acquires or releases, not just as synchronization accesses. Hence, programming complexity is higher in RC than in WO.

What mechanism is used in the processor to implement RC? Recall that a release synchronization must prevent downward migration. When a release synchronization access is encountered in the processor pipeline, the release access is stalled (not committed) until all prior memory accesses have completed. That is, all prior loads must have obtained their values, and all prior stores have emptied the write buffer and have fully propagated their values (through invalidation).

An acquire synchronization must prevent upward migration. When an acquire

synchronization access is encountered in the processor pipeline, some of the accesses that are younger than the acquire may have executed in an out-of-order processor pipeline. Such a situation contradicts with the need to prevent them from issuing to the cache and the rest of the memory hierarchy. An out-of-order processor already has a mechanism to cancel the effect of speculative instructions in order to arrive at an instruction-specific state. An acquire synchronization access can rely on such a mechanism. When an acquire synchronization access is encountered in the processor pipeline, all instructions younger than it (including all loads and stores) are canceled, and re-executed after the acquire synchronization completes. Alternatively, when an acquire synchronization access is decoded in the processor pipeline, all younger instructions are discarded from the fetch and decoded stages and the further fetching is stalled until the acquire synchronization instruction has retired.

Similar to WO, RC allows the compiler to freely reorder loads and stores except that they cannot migrate upward past an acquire synchronization and cannot migrate downward past a release synchronization. However, the flexibility and performance advantage of RC comes at the expense of requiring synchronization accesses to be properly identified and identified as acquires or releases. Unlike in WO, synchronization accesses cannot be easily identified by instruction opcodes alone. For example, the appearance of special instructions such as atomic fetch-and-op, test-and-set, load linked, store conditional, etc. does not give an indication of whether they are acquire or release synchronization. While a release always involves a store, an acquire may involve both loads and stores. Hence, the burden is on programmers' shoulders to properly identify acquire and release synchronization accesses.

10.3.5 Lazy Release Consistency

A further optimization of the Release Consistency (RC) can be achieved when we consider that the thread that executes an acquire synchronization does not need values written by another thread until the acquire synchronization is completed. For example, as can be observed in Figure 10.2, in a critical section in thread P1, the value of `sum` set by thread P0 in its previous critical section is not needed by P1 until after P1 obtains the lock or until after P0 releases its lock. Also, as can be observed in Figure 10.3, thread P1 does not need the most recent value of `datum` until it has finished its *wait* operation or until P0 has finished its *post* operation. Interestingly, this presents a performance optimization policy through modifications

to the behavior of cache coherence. In essence, the timing for performing write propagation can be tweaked when acquire and release synchronization accesses are properly identified.

For example, consider the top portion of Figure 10.5, which shows when propagation is performed on a cache coherent system that relies on the RC model. `datum` is propagated completely before the propagation of `datumIsReady`. However, the value of `datum` is not really needed until after thread P1 completes its acquire synchronization. In a consistency model referred to as **Lazy Release Consistency** (LRC), all values written prior to the release synchronization are propagated together with the propagation of the release synchronization itself. For example, in the bottom part of Figure 10.5, the new values of `datum` and `datumIsReady` are propagated together at the release point.

Figure 10.5: Illustration of Lazy Release Consistency.

At this point, readers may wonder why delaying write propagation can be beneficial. Performing write propagation in bulk will slow down the release and subsequent acquire synchronization, and hence on a hardware cache coherence system there may not be any gain. However, in a system that has little bandwidth available between processors, or in a system that suffers from higher overheads due to frequent propagation of small amount of data versus infrequent propagation

of a large amount of data, LRC can really help performance. An example in which both of these situations exist is a multiprocessor that does not provide a shared memory memory abstraction in hardware, but instead employs a software layer that provides a shared memory abstraction. In such a "**software shared memory**" system, write propagation is performed at the page granularity, hence it is extremely expensive to propagate a page each time one block in the page is modified. Hence, write propagation is delayed until the release synchronization point is reached, at which time entire pages that have been modified up to this point are propagated.

LRC is not employed in hardware-supported cache coherent systems because write propagation can be efficiently performed even at a small granularity of a cache block.

10.4 Synchronization in Different Memory Consistency Models

Some programs or thread libraries have been developed for one platform that has a specific memory consistency model. They may not have been programmed for portability to systems with a different memory consistency model. This section discusses how they can be ported from one model to another to achieve equivalent execution. Note that memory consistency models must be considered mostly by programmers dealing with synchronization primitives, such as in thread libraries or in the OS.

The first step programmers must take is to analyze what ordering is guaranteed by the system, and what ordering is required by their software to perform correctly. Then, they need to impose additional ordering requirements for example by inserting fences to their code, or remove unnecessary ordering requirements such as by removing fences. Superfluous fences only reduce performance, but missing fences may produce incorrect execution. Hence, programmers naturally tend to be more conservative than aggressive in imposing ordering. Interestingly, anecdotal evidence points to an excessive insertion of fence instructions in complex code such as the Operating System (OS). The reason for this is that some OSes needs to run on multiple machines with various memory consistency models. To ease porting, programmers sometimes intentionally sacrifice performance by inserting excessive fence instructions assuming the most relaxed memory consistency model. Thus, the code requires very little modification when ported to

machines with stricter memory consistency models.

For now, however, let us assume that we start with a program written by programmers with sequential consistency in mind, and that program order is maintained during compilation. We want to implement the program on several different consistency models, and when fences are used, we want to use as few fences and as weak fences as possible in order to still guarantee correct execution. We will use the *post-wait* synchronization example to illustrate the differences in these models. For the machines, we will highlight the differences between a system that provides acquire vs. release types of fences, a system that provides load vs. store vs. full types of fences, and a system that provides labeling of stores and loads as release and acquire synchronization.

Our first example is the IBM PowerPC-style architecture. The architecture style does not support labeling of regular loads/stores as acquire or release synchronization accesses. However, it provides two types of fence instructions: an all-out fence (preventing upward and downward migration), and an acquire fence (preventing upward migration) only. The regular fence is provided through the instruction `lsynch` while the acquire fence is provided through the instruction `isynch`. Suppose that we want to implement the post-wait synchronization from Code 10.1. The producer code needs to use the regular fence between the write to `datum` and the write to `datumIsReady`, as shown in Code 10.7.

Code 10.7 Post synchronization with IBM PowerPC style.

```
1 datum = 5;
2 lsynch
3 datumIsReady = 1;
```

The reason for using a regular fence here is to strictly order the two writes. Avoiding only upward migration above the fence does not prevent the write to `datum` from moving below the fence, while avoiding downward migration does not prevent the write to `datumIsReady` from moving up past the fence. Both cases may cause either the write to `datum` to be propagated late, or `datumIsReady` to be propagated too early, which are both incorrect. Therefore, in the absence of labeling acquire or release synchronization, a regular fence instruction is needed to provide strict ordering at the release side.

At the consumer (acquirer) side, we need to use `isynch` which prevents upward migration only, as shown in Code 10.8

The `isynch`'s position prevents the load from `datum` from issuing before

Code 10.8 Wait synchronization with IBM PowerPC style.

```
while (!datumIsReady) {};
isynch
... = datum;
```

the `isynch` is complete. The `isynch` itself is not executed until the program goes out of the loop after the load to `datumIsReady` reads the new value of `datumIsReady`. Notice that the control flow dependence between the loop and the `isynch` (and hence also the load from `datum`), ensures that the load from `datum` will return the new value of `datum`.

Compared to the support of annotating acquire and release synchronization accesses, the support in the form of fences is more restrictive at the release side because a full fence is used (preventing upward and downward migration) compared to an acquire synchronization (which prevents only upward migration).

Now let us consider another example in which the fences are of load or store types. For example, the Sun SPARC architecture uses Total Store Ordering (TSO) or Partial Store Ordering (PSO). In PSO, the execution ordering between two stores is not guaranteed unless a store fence is inserted between them. The store fence instruction (`stbar`) must be explicitly inserted by programmers. For the post-wait synchronization pair, the producer executes two stores, so clearly a store fence is sufficient, as illustrated in Code 10.9.

Code 10.9 Post synchronization with Total Store Ordering style.

```
datum = 5;
storefence
datumIsReady = 1;
```

The consumer executes a series of loads within the `while` loop, and a load that reads from `datum`. The loads within the while loop all read the same address so they do not need fences among themselves. Between the loads in the loop and the load that reads from `datum`, however, a load fence is needed (if the system has not already guaranteed ordering between loads). This is shown in Code 10.10.

The final example is for the IA-64-style instruction set architecture that supports several instructions: `mf` which is a full memory fence, `st.rel` which is a release store synchronization instruction, and `ld.acq` which is an acquire load synchronization instruction. To implement the post-part of the synchronization,

Code 10.10 Wait synchronization with Total Store Ordering style.

```
while (!datumIsReady) {};
loadfence
... = datum;
```

we can label the store to `datumIsReady` as a release synchronization access, as shown in Code 10.11.

Code 10.11 Post synchronization with IA-64 style.

```
// suppose R1 = 5, R2 = 1
st &datum, R1 // datum = R1
st.rel &datumIsReady, R2 // datumIsReady = R2
```

which ensures that the store to `datum` is completed prior to the execution of the store to `datumIsReady`.

The consumer performs a series of loads within the `while` loop that are involved in the *wait* synchronization, and a load that reads from `datum`. Since the loads are involved in an acquire-type of synchronization, we can simply label the load instruction by replacing it with the `ld.acq` instruction, as shown in Code 10.12.

Code 10.12 Wait synchronization with IA-64 style.

```
wait: ld.acq R1, &datumIsReady  // R1 = datumIsReady
sub R2, R1, #1 // R2 = R1 - 1
bnz R2, wait   // jump to wait if R2 is not 0
ld R3, &datum  // R3 = datum
```

This guarantees that the load from `datum` does not execute until the load from `datumIsReady` is completed.

Overall, different architectures support different memory ordering guarantees and safety nets. This prevents low-level code, such as the OS or thread libraries, from being portable. The benefits of RC in allowing reordering of memory accesses compared to other stronger memory consistency models are apparent and appealing. However, the complexity of programming them, especially considering the different ways of expressing synchronizations to different architectures, is potentially daunting. To avoid such complexities in dealing with various architectures, more recently, the memory consistency model is exposed in the program-

ming language, such as in the **Java Memory Model**. With a memory model at the programming language level, it is the responsibility of a Java compiler to translate code to different architectures in a manner that conforms to those architectures.

10.5 Exercise Problems

1. (*15 Points*) Suppose that we want to run a piece of code on a machine that does not impose any ordering of memory operations (it can reorder loads/stores that are not data-dependent). However, the machine provides a fence instruction that, if inserted in a code, prevents issuing memory operations that come after the fence before the memory operations that are before the fence are globally performed. Assume that all lock primitives have been annotated as/with acquire synchronization while all unlock primitives have been annotated as/with release synchronization. The code to be executed is:

```
lock1
load A
store B
unlock1
load C
store C
store D
lock2
load E
store F
unlock2
```

   (a) Insert as few fence instructions as possible to ensure a sequential consistent execution of the code

   (b) Repeat it for processor consistency

   (c) Repeat it for weak ordering

2. (*30 Points*) **Performance of memory consistency models**. Given the following code fragment, we want to compute its execution time under various memory consistency models. Assume a processor architecture with an arbitrarily deep write buffer. All instructions take 1 cycle, ignoring memory system effects. Both read and write instructions that miss in the cache take 100 cycles to complete (i.e., to perform globally). Assume the cache lines are not shared (thus, a write hit does not generate invalidations). Locks are

cachable and loads are non-blocking. Assume all the variables and locks are initially uncached and all locks are unlocked. Further assume that once a line is brought into the cache it does not get invalidated for the duration of the code's execution. All memory locations referenced here are distinct and mapped to different cache lines. The processor can only issue one instruction at any given cycle, and it gives preference to instructions that are earlier in program order. Assume that all lock primitives have been annotated as/with acquire synchronization while all unlock primitives have been annotated as/with release synchronization.

(a) If the sufficient conditions for sequential consistency (SC) are maintained, how many cycles will it take to execute the code?

(b) Repeat part (a) for weak ordering

(c) Repeat part (a) for release consistency

Use the following notation in each of the table's cell: (starting time, ending time).

Instruction	Hit/Miss	SC	WO	RC
Load A				
Store B				
Lock (L1)				
Store C				
Load D				
Unlock (L1)				
Load E				
Store F				

3. (*40 Points*) **Results under different memory consistency models**. Given the following code segments, specify what read results are possible (or not possible) under sequential consistency (SC), Processor Consistency (PC), Weak Ordering (WO), and Release Consistency (RC). Assume that all variables are initialized to 0 before this code is reached. All the instructions are annotated as regular loads (LD), stores (ST), synchronization operations (SYN), acquire (ACQ), and release (REL).

In addition, if you find results that are impossible under SC, **give the ordering** of operations that produce those results.

(a)

P1	P2
(ST) A=2	(LD) read A
(ST) B=3	(LD) read B

For your answer, you can use the following template for your answer, but add explanation of the ordering that result in impossible results for SC.

Possible Result	SC	PC	WO	RC
read A returns 0, read B returns 0				
read A returns 2, read B returns 0				
read A returns 0, read B returns 3				
read A returns 2, read B returns 3				

(b)

P1	P2
(ST) A=2	(LD) read B
(ST) B=3	(LD) read A

For your answer, you can use the following template for your answer, but add explanation of the ordering that result in impossible results for SC.

Possible Result	SC	PC	WO	RC
read A returns 0, read B returns 0				
read A returns 2, read B returns 0				
read A returns 0, read B returns 3				
read A returns 2, read B returns 3				

(c)

P1	P2
(ST) A=2	(ACQ) while (flag==0) ;
(REL) flag=1	(LD) print A

For your answer, you can use the following template for your answer, but add explanation of the ordering that result in impossible results for SC.

Possible Result	SC	PC	WO	RC
read A returns 0, read flag returns 0				
read A returns 2, read flag returns 0				
read A returns 0, read flag returns 1				
read A returns 2, read flag returns 1				

(d)

P1	P2	P3
(ST) A = 1	(LD1) read A	(LD2) read B
	(ST) B = 1	(LD3) read A

For your answer, you can use the following template for your answer, but add explanation of the ordering that result in impossible results for SC.

Possible Result	SC	PC	WO	RC
LD1 LD2 LD3 = 0 0 0				
LD1 LD2 LD3 = 0 0 1				
LD1 LD2 LD3 = 0 1 0				
LD1 LD2 LD3 = 0 1 1				
LD1 LD2 LD3 = 1 0 0				
LD1 LD2 LD3 = 1 0 1				
LD1 LD2 LD3 = 1 1 0				
LD1 LD2 LD3 = 1 1 1				

Chapter 11

Distributed Shared Memory Multiprocessors

Contents

11.1 Approaches to Large Scale Multiprocessors 320

11.2 Building a Directory-Based Coherence Protocol 322

11.2.1 Directory Format 327

11.3 Basic DSM Cache Coherence Protocol 332

11.4 Implementation Correctness and Performance 338

11.4.1 Handling Races due to Out-of-Sync Directory State . 338

11.4.2 Handling Races due to Non-Instantaneous Processing of a Request 342

11.4.3 Write Propagation and Serialization 352

11.4.4 Synchronization Support 354

11.4.5 Memory Consistency Models 356

11.5 Exercise Problems 357

In earlier chapters, we have discussed how the shared memory abstraction can be supported on a bus-based multiprocessor through various cache coherence protocols in Chapter 8. While the hardware support for a bus-based multiprocessor is relatively simple because it extends mechanisms that already exist on a single processor system (cache states, bus transactions, etc.), bus-based multiprocessors suffer from the limitation of not being scalable.

The objective of this chapter is to discuss the design of scalable shared memory multiprocessor systems. A bus-based multiprocessor can hardly scale beyond

8-16 processors due to the limitation of a bus. In order to provide a shared memory abstraction for larger systems consisting of tens to hundreds of processors, a new organization is needed. In this organization, each node consists of a processor (or a group of processors), a cache hierarchy, and a local memory. The hardware takes the memories of different nodes to form a single memory addressable by all processors. Such a system is also known as a **Distributed Shared Memory** (DSM) system, referring to the nature of distributed memories that have an abstraction of a shared memory, or **Non-Uniform Memory Access** (NUMA) system, referring to the fact that accessing different parts of the memory takes a different amount of time to access. Compared to a bus-based multiprocessor system, a DSM system can scale to a larger number (hundreds) of processors. However, the hardware support in a large-scale DSM is costly, and having a very large DSM may not be the most cost-effective solution. Nevertheless, in this chapter, we will assume that the main goal is to achieve a scalable shared memory multiprocessor, and discuss what hardware support is required.

11.1 Approaches to Large Scale Multiprocessors

There are two aspects that restrict the scalability of bus-based multiprocessors. The first aspect is the *physical* scalability limitation of the interconnection (i.e., the bus), while the second aspect is the scalability limitation of the *protocol* (i.e. broadcast and snooping). The physical scalability limitation of the bus is due to physical constraints. While a bus offers the advantage of being extensible (i.e. adding more processors does not require a new bus), connecting more processors to the bus reduces the speed of the bus. A bus needs an arbiter to avoid conflicting use by multiple processors. To get an access to the bus, the processor sends a request to the arbiter, and the arbiter grants the bus access to the processor. As a bus becomes longer, the delay to send a request to the arbiter and for the arbiter to send a bus grant signal increases. Clock skew also becomes more pronounced on a longer bus. As a result, this reduces the bus clock frequency, which in turn reduces the available bandwidth.

The second limitation of a bus is due to the coherence protocol used in a bus-based multiprocessor that relies on broadcasting and snooping. As the number of processors that connect to the bus increases, there are more bus transactions placed on the bus. They may overwhelm the available bandwidth of the bus, which is already decreased due to the increased wire length when more processors are

connected to it.

What are the possible ways to make a shared memory multiprocessor more scalable? Figure 11.1 shows several alternatives. The first alternative is to keep a snoopy protocol but replace the bus with a set of smaller buses, or replace the bus with **point-to-point** interconnection. In the former approach, processors are divided into groups, and processors in a group are interconnected with a leaf-level bus, and leaf-level buses are interconnected with a root-level bus. In this organization, all the buses use short wires and can be clocked at a high frequency. This organization is used in the KSR-1 multiprocessor [Windheiser et al., 1993]. However, the root-level bus needs to have more bandwidth than leaf-level buses in order to prevent it from becoming the performance bottleneck in the system.

In the latter approach, i.e. using point-to-point interconnection, a link only connects two nodes, and is not shared by multiple nodes. Therefore, each link in a point-to-point interconnection does not require arbitration and uses relatively short wires. It can be clocked at a high frequency and provides a higher bandwidth compared to a bus. Such advantages of point-to-point interconnection over a bus make it easier to scale the multiprocessor to a larger number of nodes. Some medium-scale multiprocessors such as that based on the AMD Opteron use point-to-point interconnection in conjunction with a snoopy cache coherence protocol [AMD, 2005]. However, at some point the protocol constraint puts a lid on scalability because the total traffic in each link grows linearly with the number of processors due to the reliance on broadcast and snooping. Therefore, snoopy protocols are not used for large (tens to hundreds of nodes) shared memory multiprocessors.

		Interconnection	
		Bus	Point–to–point
Protocol	Snoopy	Least scalable	More scalable
	Directory		Most scalable

Figure 11.1: Ways to scale a bus-based multiprocessors.

The most scalable organization uses point-to-point interconnection and, rather than relying on broadcasting and snooping, relies on a **directory** approach. In a directory approach, the information about which caches have a copy of a block is maintained in a structure called the directory. With such an approach, locating cached copies of a block does not rely on broadcasting a request to all processors, but rather by inquiring the directory about which caches may have copies of the block, and sending the request only to those caches. If a block is cached by all processors, then the request must be sent to all processors, which yields comparable traffic to a broadcast/snoopy approach. However, if a block is cached by only a few processors, the traffic savings of the directory approach over a broadcast/snoopy approach is substantial. Which case is more likely in reality? Well-tuned applications exhibit little data sharing for blocks that are frequently read and written because if there is a lot of such data, the applications will suffer from excessive serialization due to the critical sections needed to protect accesses to the data, as well as from coherence misses due to invalidations to the data. Thus, in well-tuned applications, most data sharing occurs only for data that is read only, and there is little sharing for data that is frequently read and written. For such data, a directory approach can yield a substantial traffic savings compared to the broadcast/snoopy approach because most writes will cause only a few caches to be invalidated. One case in which there may be little saving in traffic is when there is highly-contended synchronization in which a variable is frequently read and written by all processors. But again, this case implies a high number of invalidations which restrict the performance and scalability in other ways, which cannot be solved by using a bus.

11.2 Building a Directory-Based Coherence Protocol

In order to build a DSM system, there are many questions that we need to consider: which memory a block must be placed in, where the directory of a particular block can be found, how the directory information is organized, how to handle races in the protocol, how to ensure write propagation and write serialization in a DSM, and what performance characteristics and optimizations can be applied.

The first question deals with where (i.e., which memory) to place a particular block. One necessary criteria for the placement is that the memory in which a block is placed must be directly locatable without any reliance on searching. Thus, we can only use a simple physical address-to-memory mapping function. Performance-wise, the ideal case is to place the block in the memory of the pro-

cessor that will access it the most, since accessing the local memory can be a lot faster than accessing a remote memory. The question deals with two issues. The first question deals with how physical address is mapped to memory, while the second deals with how a virtual page is mapped to a physical page frame. One way to map physical address to memories is to *interleave* them at the block level. For example, physical block 1 is mapped to memory of node 1, physical block 2 is mapped to memory node 2, and so on (Figure 11.2). The advantage of this interleaving is to distribute blocks of a page evenly on many memories, allowing better balancing of accesses to different memory. Unfortunately, the approach works against spatial locality of accesses. A thread tends to access multiple blocks of a page consecutively. With block interleaving, most of the accesses will be remote memory accesses which incur high latencies. Hence, in most DSM implementations, block interleaving is not used. Instead, memories are treated as linear and no interleaving is employed (Figure 11.3).

Figure 11.2: Mapping physical address to memories using block interleaving.

Without interleaving, consecutive physical page frames ("pfr" in the figure) are mapped to a memory, and the next consecutive page frames are mapped to the next memory, and so on. Without interleaving, the challenge of allocating blocks as close possible to the processor that uses them the most becomes a *page allocation* problem. The Operating System (OS) must be designed to map a page to the physical page frame that resides in the node in which the processor that

Figure 11.3: Mapping physical address to memories without interleaving. pfr = physical page frame.

uses it the most resides. When a parallel program begins its execution, it starts with only a few allocated pages. As it executes, it suffers from page faults from accessing pages that have not been allocated. At this time, the OS can choose which page frame to allocate the page. While the goal of page allocation should be to allocate a page on the memory that is the closest to the processor that accesses it the most, the OS lacks good information about the processor access patterns at the time it needs to allocate a page. Hence, it is typical that the OS relies on simple heuristics. In a single processor system, the heuristic it uses is to place a new page on the page frame that was least recently accessed, because the old page in the frame is likely to not be reused in the future. However, in a DSM system, the OS must be aware of how far that page frame is from the processor that likely needs it. One popular heuristic is to allocate the least-recently used frame on the node that incurs the page fault, i.e. on the same node where the processor that first accesses it resides. Such a policy is referred to as the **first touch** policy. The rationale behind this heuristic is that the processor that first accesses the data in a page is likely to be the same processor that accesses data in the page the most.

As discussed in Section 4.3.6, a pitfall of the first touch policy occurs when the main data structure (e.g., the matrices) are initialized in a different traversal order than that in the main computation. Allocating pages according to the initialization traversal order which is inconsistent with the traversal order in the main

computation can lead to many remote memory accesses during the main computation. Worse, if the initialization is performed sequentially by the main thread, all pages of the data structures will be allocated on a single node. During main computation, all but one node will access a remote memory. Not only does this incur a high average memory access latency, but this also causes contention on that node, which further increases the average memory access latency. Such a scenario can actually be quite common when programmers use an automatic parallelizing compiler. The compiler typically employs (reasonable) heuristics that skip parallelization of loops or loop nests that have a very small body. However, since an initialization loop or loop nest often has a small body, it is often not parallelized. A common solution for this problem is to manually force the parallelization of the initialization loop, and ensure that the traversal order used in the initialization loop is the same as that used by the loops in the main computation.

An alternative page allocation policy is *round robin*, in which pages are allocated in different memory nodes in a round robin manner. Round robin page allocation sounds like a bad idea since it virtually guarantees that there will be many remote memory accesses. However, it spreads pages across nodes in a balanced manner so contention on any particular node rarely occurs with this policy.

Regardless of the page allocation policy used, a given physical block address maps to a single memory using a mapping function that is fixed at the system boot time. The memory where a block maps to is referred to as the **home node** of the block.

■ *Did you know?*

There has been a research proposal to treat the memory as a cache, that is, the notion of a home node for a block address is eliminated. Like in caches, a block can be cached in the local memory and remote memories. Such an architecture is referred to as the **Cache-Only Memory Architecture** *(COMA). COMA allows blocks to be replicated in different memories, allowing local memory access by processors in the corresponding nodes. However, keeping cache-like memories comes with a significant cost and complexity. A special mechanism is needed to locate the memory where a block may reside, if the block is not found in the local memory. In addition, care must be taken to avoid replacing the only copy of a block from the memory.*

So far, we have discussed how physical address maps onto memory. Given an access by a processor, the system has the ability to locate the home node of

the block address accessed by the processor. However, the valid value of the block may not reside in the home node of the block. It may be cached dirty by a processor. A read must obtain the latest value of the block to ensure write propagation. Similarly, a write must find out which caches store a copy of a block, and must invalidate those copies to ensure write propagation. While a read or write request can be broadcasted to find all caches that have a copy of the block, such an approach is not scalable. Therefore, a **directory** that keeps track of which caches have a copy of a block, and in what state they cache it, must be kept. A logical question is where should the directory information of a block be stored, and what information format is needed in the directory?

The directory information of a particular block must be found quickly and easily in order for a read or write to proceed quickly. One alternative is to keep the directory in the main memory. Such an approach is referred to as **memory-based directory**. Another alternative is to keep the directory in the caches where copies of a block reside. Such an approach is referred to as **cache-based directory**. In a typical cache-based directory protocol, the information of a block that is stored in and shared by several caches is kept as a doubly linked list. Each cache keeps pointers to the next and previous caches in the sharing list. In theory, keeping the list of sharers in a distributed fashion increases the scalability of the protocol. Unfortunately, maintaining the list in a way that all the processors in the system have a coherent view of the current list of sharers is complicated. Thus, despite one of such protocols called **Scalable Coherent Interface** (SCI) ratified as an IEEE standard in 1992, it has not been implemented commercially as widely as memory-based directory protocols.

In this chapter, our discussion will center around memory-based directory protocols. In the memory-based directory approach, we can designate a single node to keep the directory information of all blocks in the system. This approach is referred to as the **centralized directory**. Unfortunately, a centralized directory is not scalable as it receives requests from all processors, and its central location is far away from most processors. The alternative to the centralized directory approach is the **distributed directory**, in which the directory information is scattered at different nodes. A logical way for scattering the directory information is to keep the directory information of a block in the home node of that block. Since given a block address, finding the home node is straightforward, finding the directory information for the block is also straightforward.

The basic protocol for serving a load miss is illustrated in Figure 11.4. The

figure shows how the read that corresponds to the load miss is handled, when a block resides in a cache in a modified state. Since the cached copy is the only valid copy in the system, the processor which performs the read must be able to obtain the block. To do that, the *requestor* (P1) first inquires the home node (P2), which also keeps the directory information of the block (Circle 1). The directory must include information that indicates the fact that the latest value of the block is not available in the home node, and which cache has the latest value of the block, called the *owner* node (P3). Then, the directory can forward the request to the owner (Circle 2), which responds by supplying the block to the requester (Circle 3), either directly or through the home node.

Figure 11.4: Basic protocol operation in a DSM system for a read request.

Figure 11.5 shows how a write is handled when a block is shared among several sharers. First, the requestor (P1) sends the write request to the home node (P2). The requestor may or may not already have the block in its cache (the figure shows the case in which the requestor has the block in its cache). The home node, which keeps the directory information for the block, looks up the directory to discover which caches have a copy of the block. Then, it sends invalidations to all other sharers (Circle 2), and the sharers (P2 and P3) respond by notifying either the requestor or the directory of the receipt of the invalidations (Circle 3).

11.2.1 Directory Format

In the previous section, we have discussed the basic protocol for serving a read or write request. Essentially, when the read/write request reaches the directory, the directory must know: (1) where the latest copy of the block resides, whether in the home node memory, or in a processor's cache, and (2) which cache(s) has a

Figure 11.5: Basic protocol operation in a DSM system for a write request.

copy of the block. We will discuss the two requirements separately.

In order to satisfy the first requirement, the directory must keep the state of the block in the cache, that is, it must know whether the block is dirty or clean. This can be achieved by keeping the current cache state of the block, for example, whether the block is not cached, cached in the shared state, or cached in a modified state. For example, if the caches use MESI states for each block, then ideally the directory at home should also keep a MESI state that perfectly reflects the state of the block in the cache(s). However, the directory state cannot always fully reflect the cache state of a block. For example, in MESI, if a block is cached in an Exclusive state in a node, when the processor in the node wants to write to the block, it can do so silently since it knows that no other cached copies exist. Hence, the block state is changed from Exclusive to Modified silently, and the block value is also changed silently. In this case, the directory never sees such a transition from Exclusive to Modified. With MESI, the directory cannot distinguish between a block that is cached in Exclusive state or Modified state. Thus, it can only keep three states: Exclusive/Modified, Shared, or Invalid.

To satisfy the latter requirement of keeping the information of which caches have a copy of a block, there are several possible formats the directory can keep. The important factors in choosing which format to use for a particular case are the *amount of storage overheads* and whether the information can overflow the allocated storage format or not.

One format is to keep a bit that corresponds to each cache that indicates whether the block is cached there or not. This format is referred to as the **full bit vector** directory format. If there are p processors, each block in memory would

then require p bits of information. Note that since the cache block size does not change with the number of processors, the ratio of directory storage overhead to the storage for a data block grows linearly with the number of processors. The overhead can be quite high for a larger machine. For example, assuming a cache block size of 64 bytes, when there are 64 processors, the storage overhead ratio is $\frac{64}{64\times 8} = 12.5\%$, which is substantial. It becomes worse quickly with a larger number of processors. For 1024 processors, the storage overhead ratio becomes $\frac{1024}{64\times 8} = 200\%$! If the directory information is kept in the main memory, that means that two thirds of the main memory capacity is used up for storing directory information, and only one third is available for storing data. Clearly, the full bit vector is too costly to use in large DSM systems.

A simple way to reduce the storage overhead of the directory is to use one bit to represent a group of caches, rather than just one cache. If any cache in a group has a copy of a block, the bit value corresponding to the group is set to "1". If the block is not cached anywhere in the group, the bit value is set to "0". This format is referred to as the **coarse bit vector** directory format. Since the information is kept per group, when there is an invalidation, the invalidation must be sent to all caches in the group, even when only one cache in the group has the block. Thus, traffic increases compared to the full bit vector approach. However, the storage requirement for the directory decreases as well. If the number of caches per group is g, the storage overhead in the sharing bit vector per group is $\frac{p}{g}$. In the example above, for a 1024-node system with 64-byte block size, and a group size of 4, the storage overhead ratio is lowered to $\frac{\frac{1024}{4}}{64\times 8} = 50\%$.

A different approach to reduce the storage overhead is to not keep full directory information of all blocks in the main memory, but instead keep partial directory information, or keep full directory for only a limited number of blocks. One format that uses this approach is the **limited pointer** format, which keeps a limited number of pointers to sharers in the system. For example, for a given block in memory, suppose we keep a pointer to only n caches. Each pointer requires $log_2(p)$ number of bits to represent the ID of the node. Hence, the storage requirement per block is $n \times log_2(p)$. For a 1024-node system with 64-block size, and 8 pointers per block, the storage overhead ratio is $\frac{8\times log_2 1024}{64\times 8} = 15.6\%$, which is significantly lower than a full bit vector format. Of course, the amount of storage overhead reduction depends on how many pointers are kept. However, a particular block is rarely cached in many locations, and hence most of the time four to eight pointers would be sufficient.

One question is in the rare case in which the number of sharers is larger than the number of pointers, how can this situation be handled by the directory? In this case, the directory runs out of IDs that it can keep. There are several strategies to handle this case. One strategy is to not allow the case to occur. That is, when there is a cache which wants to keep a copy of a block, the directory sends an invalidation to a cache that currently keeps a copy of a block to make a room to store the ID of the new cache. This strategy can backfire when the reason that there are many sharers of a block is because the block contains data used for global synchronization, such as a centralized barrier. In this case, naturally many sharers are present, and when the number of sharers is kept artificially low, the barrier can become significantly slower. Other strategies, such as reverting to broadcast when the number of sharers exceeds the number of pointers, suffers from a similar problem. Another possible strategy is to keep a pool of free pointers which can be allocated when such a heavy amount of sharing occurs. Since not many blocks are heavily shared, the number of such free pointers does not need to be large.

The limited pointer format keeps limited directory information (when overflowed) for all blocks in memory. Another way to rely on partial information is to keep full directory information only for a subset of blocks in memory. Note that in a typical configuration, the number of blocks in the main memory greatly exceeds the number of blocks that can be cached, because the main memory size greatly exceeds the cache size. Thus, for most blocks, the directory state will indicate that they are not cached at all. One way to compress the directory storage is to remove the directory information for such blocks, and only keep the directory information for blocks that are cached. This leads to a format referred to as the **sparse directory** format. In the sparse directory format, when there is a request for a block, the request is checked against the entries in a **directory cache**. If the entry for the block is found in the cache, the directory information is retrieved. If it is not found in the cache, the block must not have been cached before. Hence, a new entry is created in the directory cache for that block. With a sufficient directory cache size, the directory information of all cached blocks should fit in it. There is, however, a risk for overflow. The reason for this is that a clean block may be replaced silently from the cache, without notifying the directory cache. Thus, the directory cache may still keep the directory entry for the block, preventing the entry from being used for a block that may actually be cached. When the directory cache runs out of space, there is a possibility of evicting an entry that corresponds to a block that is still cached, while keeping entries corresponding to blocks that are no longer

cached. Therefore, an overflow is a possibility, and must be handled with similar strategies used for handling the overflow of the limited pointer scheme.

Note that the directory formats are not exclusive of one another in their use. We can simultaneously employ several formats. For example, the limited pointer may be combined with the coarse format in that a pointer refers to a group of nodes rather than to a single node. Or, a coarse bit vector format or limited pointer format may be used in conjunction with a sparse directory in order to further reduce their sizes. Finally, we can switch from one format to another as situations change. For example, the SGI Origin 2000 system uses a limited pointer scheme when a block is only cached in one processor, but uses a full bit vector (with a bit corresponding to a node of two processors) when a block is cached by more than one processor, and uses a coarse bit vector for very large machine configurations.

Another issue is related to where the directory information is physically located in a node. There are several choices. One choice is to allocate a part of the *main memory* for storing directory entries. At boot time, the hardware sets aside a portion of memory so that it is managed by the coherence controller hardware and is invisible to the OS. The advantage of this scheme is that it is simple and can be supported without modifying the main memory. The disadvantage, however, is that accesses to the directory compete for main memory bandwidth with accesses to data. It is possible that this contention introduces a significant delay to data access and to directory access. So instead, some implementations allocate a separate memory to store directory information that is accessible using separate wires from the coherence controller. While this solves the contention issue, it is a more costly approach. Another approach is to keep the directory information in the same chip as the processor. The advantage is that accesses from the local processor can quickly determine whether the block can be fetched from local memory or from a remote cache, in contrast to spending a lot of time accessing an off-chip directory. The disadvantage is of course there is less die area available on chip for other uses.

With regard to the type of storage used for keeping the directory information, both DRAM and SRAM can be used. The tradeoffs are straightforward: DRAM is denser but is slower to access, while SRAM is not as dense and faster to access. Large DSM often rely on DRAM directory as the storage required is large, but an SRAM directory cache offers an appealing way to reduce the average directory access time.

■ *Did you know?*

In the IBM Power4 architecture [IBM, 2002], the directory information is kept on the processor chip. To reduce the size of the directory, the information kept in the directory only has one bit that indicates whether the block is only cached by the local processors (there are two processors in one chip), or possibly cached by other, remote, processors. If the block is only locally cached, a read or write request is processed locally. If, however, the block may be remotely cached, the directory information does not have sufficient information to tell which other processors may have a copy of the block. Thus, the request is broadcasted to all other processors. This increases traffic significantly when many blocks are shared by several processor chips. However, this is made rare by the virtue of (1) the multiprocessor system is targeted only for medium scale multiprocessor, and (2) most well-tuned programs have little data sharing between processors.

11.3 Basic DSM Cache Coherence Protocol

So far, we have discussed how a basic protocol works and what information needs to be kept in the directory. Now, we will look at implementation details of a DSM cache coherence protocol. The discussion will proceed in two steps. First, we will describe a working protocol, while leaving out discussion on some design choices that can be made and what implications the choices cause. After that, we will discuss the correctness and performance issues in more details.

To start off with a base coherence protocol, let us assume that the caches of all nodes use MESI states. Therefore, in the directory, we can keep the following states for each memory block:

1. **Exclusive or Modified** (EM): the block is either cached in Exclusive or Modified state in only one cache.

2. **Shared** (S): the block is cached cleanly, possibly by multiple caches.

3. **Uncached** (U): the block is uncached or cached in invalid state.

Recall the directory's inability to distinguish between a block that is exclusively cached cleanly (in the Exclusive state) or dirty (in the Modified state). Hence, the directory lumps them together in the EM state.

The following request types are supported:

1. **Read**: read request from a processor.
2. **ReadX**: read exclusive (write) request from a processor that does not already have the block.
3. **Upgr**: request to upgrade the state of a block from Shared to Modified, made by a processor that already has the block in its cache.
4. **ReplyD**: reply from the home to the requestor containing the data value of a memory block.
5. **Reply**: reply from the home to the requestor *not* containing the data value of a memory block.
6. **Inv**: an invalidation request sent by the home node to caches.
7. **Int**: an intervention (downgrade to state Shared) request sent by the home node to caches.
8. **Flush**: the owner of a cache block sends out its cached data block.
9. **InvAck**: acknowledgment of the receipt of an invalidation request.
10. **Ack**: acknowledgment of the receipt of non-invalidation messages.

The first three types of messages (Read, ReadX, and Upgr) are made by the requestor and sent to the home node of the data block, where its directory information resides. Read is a request to read from a block that is not found in the cache (a read miss), while ReadX and Upgr are requests to write to a block that is not found or found in the cache, respectively. The next two types of messages (ReplyD and Reply) are the response of the directory to requests made by the requestor node. The response may contain data (ReplyD) or only contain non-data information (Reply). The next two types of messages (Inv and Int) are messages sent by the home node to sharers (to invalidate them), or to an owner of a block (to downgrade its state to Shared). The last three types of messages (Flush, InvAck, and Ack) are the response to Inv and Int messages to write-back/flush a dirty cache block, an acknowledgment of the receipt of an Inv message, or other types of messages. The Ack message here is broadly defined and can include various information that is needed.

Note that some of the messages can be combined into a single message, such as when the current owner of a block flushes the block as well as acknowledges

an invalidation request. In this case, we will denote such a combined message as "Flush+InvAck".

The finite state machine corresponding to the coherence protocol employed at the directory is shown in Figure 11.6. The left hand side of the slash symbol ("/") is the request received from a processor by the directory, and the *sender* of the message (the requesting node) is shown in the parenthesis. The right hand side of the slash corresponds to the messages sent by the directory as a response to the request received from the processor. The *destination* of the message is shown in the parenthesis, where "A" stands for the requesting node, "O" stands for the owner node (node that caches the block in state Exclusive or Modified), and "Shr" stands for sharing nodes excluding the requesting node (nodes that cache the block in state Shared). Note that the coherence action in the directory that manipulates the sharing bit vector is not shown in the figure.

Figure 11.6: Simplified directory state finite state coherence protocol diagram.

Let us first consider when the directory state is Uncached (U). If there is a read exclusive request, the directory fetches the block from the local memory and sends a data reply message to the requestor, and transitions to the EM state (it also sets the bit in the sharing bit vector that corresponds to the requesting node – not shown in the diagram). If there is a read request, the directory fetches the block from local memory and sends a data reply to the requestor. If the sharing vector indicates that there are no sharers (all bits have a value of zero), then the requestor

will be the first sharer in the system, and the directory transitions to state EM. Otherwise, the new directory state is Shared.

When the directory state is Shared, when there is a Read request, the directory knows it has a valid block in the local memory. Thus, it replies with data to the requestor and updates the sharing vector accordingly. The directory state remains Shared. If the request is read exclusive, the directory responds by sending the data value of the block to the requestor, and also invalidation messages to all current sharers. The data reply message sent to the requestor also includes the number of invalidation acknowledgments the requestor can expect to receive. The directory state transitions to EM. If it is an upgrade request, however, the directory sends invalidations to sharers, and a reply without data to the requestor.

Finally, when the directory state is EM, if a read request is received, then the new state will be Shared, and an intervention request is sent to the owner of the block. The intervention request also contains the ID of the requestor so that the owner knows where to send the block. If a read exclusive request is received, then a new processor wants to write to the block, so the directory needs to send an invalidation to the current owner. The invalidation request also contains the ID of the requestor so that the owner knows where to send the block. Note that the block may be in either Exclusive or Modified state at the owner; the directory does not know which is the case. In either case, upon receiving intervention or invalidation, the owner must respond appropriately.

To illustrate the operation of the protocol, consider an example of a stream of memory accesses to a single memory block shown in Table 11.1, assuming a three-node multiprocessor in which each node has a processor, a cache, and a memory with a directory of all local memory blocks. We assume the directory format to consist of the directory state and a full-bit sharing vector. The table shows requests made by various processors in the following format: Rx or Wx, where R stands for a read request, W stands for a write request, and x stands for the ID of the processor making the request. Messages generated as a result of the processor request are also shown, in the following format: Msgtype($Src \rightarrow Dest$).

Note that in this illustration, we assume that all messages that correspond to each request occur atomically, i.e. messages from different requests are not overlapped. Overlapping of messages from different requests can introduce potential correctness problems and must be handled correctly. They will be discussed in later sections. For now, we assume that requests are processed one at a time.

The example does not make any assumption on where the home of the block

Table 11.1: Illustration of directory-based coherence protocol operations.

Request	P1	P2	P3	Directory	Messages	#Hops
Initially	–	–	–	U, 000	–	
R1	E	–	–	EM, 100	Read(P1→H) ReplyD(H→P1)	2
W1	M	–	–	EM, 100	–	0
R3	S	–	S	S, 101	Read(P3→H) Int(H→P1) Flush(P1→H // P1→P3)	3

W3	I	–	M	EM, 001	Upgr(P3→H)
					Reply(H→P3) // Inv(H→P1)
					InvAck(P1→P3)
						3

R1	S	–	S	S, 101	Read(P1→H) Int(H→P3) Flush(P3→P1 // P3→H)	3
R3	S	–	S	S, 101	–	0
R2	S	S	S	S, 111	Read(P2→H) ReplyD(H→P2)	2

resides. Each time a message is sent from a processor to another processor or the home node, it is counted as one hop, and is shown in one line. A message that cannot be sent before a prior message is received indicates message dependence and they must be sent in two different hops. Messages that do not have dependence can be sent in parallel, denoted with the symbol "//" in the table.

Initially, the block is uncached, and hence the directory content indicates the Uncached (U) state, and the full bit sharing vector has 0 in all bits. When processor P1 wants to read the block, it sends a Read request to the home node (first hop). The home node checks the directory and finds that the block is uncached, hence it has the latest value of the block. It then sends a ReplyD message containing the data value of the block to P1. It also changes the state of the block to EM, with the sharing vector bit for P1 set to "1" (the sharing full-bit vector becomes "100").

Next, P1 wants to write to the block that it has just brought into its cache. It checks the state of the block, and since the block is in the Exclusive state, it changes the state to Modified and writes to the block silently, i.e. without sending any messages. The directory is unaware of this action, but it does not need to,

because the directory state of the block is already EM.

When P3 wants to read from the block, it suffers a cache miss, and this causes a Read message to be sent to the home node. The home node checks its directory and finds that the state is EM and the block is exclusively cached by P1. Since it does not know whether the block is clean or dirty in P1, it assumes that the block is dirty and sends a intervention message to it. In the message, it includes the ID of P3 so that later P1 can send the data directly to P3. Home then includes P3 as a sharer by updating its sharing vector to "101". When P1 receives the intervention request, it sends its data block to P3 using the Flush message. It downgrades its block state to Shared. In MESI, the Shared state implies a clean block, thus it also needs to send a separate Flush message to the home node containing the data value of the block. The home node uses the flushed block to update the block in the local memory. This transaction proceeds in three hops: from P3 to Home, Home to P1, and P1 to Home and P3.

After reading the block, P3 wants to write to it. It finds the block in the Shared state in its cache, so it does not have permission to write to the block. Hence, it sends an upgrade request using the Upgr message to the home node. The home node finds that currently P1 also caches the block and must be invalidated. Thus, it sends an Inv message to P1. In parallel, it sends a reply to P3 to notify it to expect an invalidation acknowledgment from P1. Home also changes the directory state to EM, and removes P1 from its sharing vector, so the new sharing bit vector value is "001". When P1 receives the invalidation message from Home, it invalidates its block and sends an invalidation acknowledgment message (InvAck) to P3. When P3 receives the InvAck from P1, it knows that it can now change its block state to Modified and write to the block.

Later, P1 wants to read from the block that was just invalidated in the previous event. P1 suffers a cache miss as it finds the block in state Invalid in its cache, so it sends out a Read message to Home. Home finds the block is in state EM and the sharing vector indicates that P3 has the block. So it sends an intervention message to P3, and P3 responds by downgrading its cache state to Shared and flushes the block to both P1 and Home. Home changes the directory state to Shared, updates the sharing vector to "101", and uses the flushed block to update the block value in the local memory.

When P3 wants to read from the block, it finds it in its cache in a Shared state. This results in a cache hit, and no messages need to be sent.

Finally, P2 wants to read the block. It sends a Read request to Home. The

directory state at home is Shared, indicating that the block is clean. Hence, the home sends a ReplyD message containing the block that it fetches from its local memory.

Let us now compare the basic directory coherence protocol scheme for DSM versus a coherence protocol for a bus-based multiprocessor. First, it is clearly more complicated, with more message types than in a bus-based multiprocessor, the maintenance of a directory at home nodes, and possible protocol races when multiple messages from different requests are overlapped. In addition, in a bus-based multiprocessor, all transactions only incur at most two hops (the request and subsequent reply on the bus), while directory coherence sometimes requires three-hop transactions. In addition, cache-to-cache transfer that can be employed in a bus-based multiprocessor cannot be applied with directory-based coherence. Since all read or write misses need to go to Home, the fastest way to supply data is when the Home has them in the local memory. In this case, no cache to cache transfer can be exploited.

11.4 Implementation Correctness and Performance

For our simple protocol we discussed in the previous section, we have assumed that (1) the directory state and its sharing vector reflects the most *up to date* state in which the block is cached, and (2) messages due to a request are processed atomically (without being overlapped with one another). In real systems, both assumptions do not necessarily apply. This results in various protocol races that need to be handled properly. Some of them can be handled relatively simply by the directory alone. But in other cases, the directory alone cannot handle them correctly. The behavior of the cache coherence controllers at each node must also be modified to aid the directory. We will now discuss how various races can be handled. Note that it is not the purpose of this chapter to exhaustively discuss all possible races. We will mostly focus on major races to give readers a flavor of the complexity in handling them.

11.4.1 Handling Races due to Out-of-Sync Directory State

The reason why the directory state and cache states can get out of sync is because they are not updated in lock step, and some events in the caches are *never seen* by the directory state. For example, a block may be replaced from a cache, and the corresponding node is no longer a sharer. However, the directory does not see this

event and still thinks that the node is a sharer of the block. Several *odd* situations can occur due to the directory having an inconsistent view of the cache states.

Consider when the directory thinks that a node is a sharer of a block, caching the block in a clean Shared state. Let us call the node A. However, the block is replaced by the node A from its cache silently. When the directory receives a read exclusive request to the block from another node, it sends invalidation messages to all nodes that the directory thinks as sharers of the block, including node A. This results in a situation in which a node receives an invalidation message to a block that it no longer has in its cache. In this case, the correct response is to acknowledge the receipt of the invalidation by sending an invalidation acknowledgment message to the requester. No harm is done here since the intended final state for the block is Invalid, and the block is already uncached (which is similar to being invalidated).

Another situation is when the directory thinks a node is already a sharer of a block, but the directory receives a read request from the node. This situation occurs because the node that kept the block in its cache has silently evicted the block, while the directory still thinks the node is a sharer. When the node wants to read from the block, it suffers a read miss, and as a result, a read request is sent to the directory. Handling this case is also simple. The directory can reply with data to the requestor and keep the sharing bit vector unchanged, since the bit vector already reflects the node as a sharer.

When the directory indicates that a block is exclusively cached (the directory state is EM), similar odd situations can occur. For example, a Read or ReadX request may arrive at the directory from a node that the directory thinks as caching the block exclusively (in Exclusive or Modified state). In this case, apparently the node that owns the block has evicted the block. If the block was clean, no write back (or flush) had occurred, whereas if the block was dirty, the flush message has yet to reach the directory. The directory cannot just reply with data since it may not have the latest data. In contrast to the case in which the directory state is Shared, with an EM state, the directory cannot reply with data just yet. However, it cannot just wait for the flushed block to arrive either because it may never come (the block might be clean and was evicted silently by the requestor while it was in the Exclusive state). Therefore, this is not a problem that can be solved just by the protocol at the directory alone. The coherence controller at each processor node must also participate. In particular, when the processor evicts a dirty block and flushes the block to the main memory at the home node, it must keep track

of whether the write back has been completed or not, in a structure called the *outstanding transaction buffer*. The home node must send an acknowledgment upon the receipt of a flush message. In addition, the coherence controller at the processor side must also delay Read or ReadX requests to a block that is being flushed, until it gets an acknowledgment from the directory that the flushed block has been received. This way, the directory will never see a Read/ReadX request to a block from a node that is still flushing it. Thus, when it receives a Read/ReadX request, the directory knows that it must be the case that the block was clean and evicted silently. Therefore, it can safely reply to the requestor with the data value of the block.

The finite state machine corresponding to the coherence protocol employed at the directory is shown in Figure 11.7. The left hand side of the slash symbol ("/") is the request received from the processor by the directory, and the *sender* of the message (the requesting node) is shown in the parenthesis. The right hand side of the slash is the messages sent by the directory as a response to the request received from the processor. The *destination* of the message is shown in the parenthesis, where "O" stands for the owner node (node that caches the block in state Exclusive or Modified), and "Shr" stands for sharing nodes (nodes that cache the block in state Shared), excluding the requesting node. The coherence action in the directory that manipulates the sharing bit vector is not shown in the figure.

Note that because the directory may be out-of-sync with the cache states, there may be requests arriving to the directory from the owner (i.e. the node that the directory thinks as caching a block in Exclusive or Modified state). In order to distinguish requests from the current owner versus from other processors, the EM state is split into two: EM_A which indicates that the directory thinks that the current owner is node A, and EM_B which indicates that the directory thinks that the current owner is node B. When the current state is EM_A, how the protocol reacts depends on whether the request comes from the current owner (A) or from other nodes (B). Likewise, When the current state is EM_B, how the protocol reacts depends on whether the request comes from the current owner (B) or from other nodes (A). Since the EM state is now split into two, there are now two additional edges that represent transitions between the two EM states.

Since the processing of requests from a current sharer or from non-sharer nodes is the same, the Shared state is not split into two. Node "X" inside parentheses indicates any node.

Let us first consider when the directory state is Uncached (U). If there is a read

Figure 11.7: Simplified directory state finite state coherence protocol diagram, with modifications to handle out-of-sync view of cache states at the directory.

exclusive request, the directory fetches the block from the local memory, sends a data reply message to the requestor, and transitions to one of the EM states (which one does not matter). If there is a read request, the directory fetches the block from local memory and sends a data reply to the requestor. If the sharing vector indicates that there are no sharers (all bits have a value of zero), then the requestor will be the first sharer in the system, and the directory transitions to one of the EM states. Otherwise, the new directory state is Shared.

When the directory state is Shared, when there is a Read request, the directory knows it has a valid block in the local memory. Thus, it replies with data to the requestor and updates the sharing vector accordingly. The directory state remains Shared. This is regardless of whether the requestor is already a current sharer or not a sharer. If the request is read exclusive, the directory responds by sending the data value of the block to the requestor, and also invalidation messages to all current sharers (except the requestor). Its state transitions to one of the EM states (which one does not matter). If it is an upgrade request, however, it only sends

invalidations (without replying with data).

Suppose the directory state is EM_A (identical responses are produced when the state is EM_B). If a read request coming from a different node (B) is received, then the new state will be Shared, and an intervention request is sent to the owner of the block (A). The intervention request also contains the ID of the requestor so that the owner knows where to send the block. If a read exclusive request from another node (B) is received, then the directory sends an invalidation message to the current owner (A), sends a reply message to B telling it to expect to receive a reply from A, and ownership transfer is indicated by transitioning from state EM_A to EM_B. The invalidation request also contains the ID of the requestor so that the owner knows where to send the block to. Note that the block may be in state either Exclusive or Modified at the owner; the directory does not know. In either case, upon receiving intervention or invalidation, the owner must respond appropriately.

Let us now discuss the new case that occurs due to the directory having an out-of-sync view of the cache states. When the state is EM_A, with an out-of-sync view of cache states, the directory may receive a read or read exclusive request from node A. This is because node A suffers a read or write miss after the clean block was silently evicted from its cache (if it had been dirty, the request would have been stalled at the requestor and instead the directory would receive a flush message from node A prior to receiving the read or read exclusive request). Since the directory already notes A as the owner of the block, it can simply react by replying with the requested data block.

Overall, handling the out-of-sync view of the cache states at the directory can be accomplished by relatively simple modifications to the coherence protocol at the directory and and to each node's cache coherence controller.

11.4.2 Handling Races due to Non-Instantaneous Processing of a Request

When messages that correspond to a request do not happen instantaneously, then it is possible for messages from two different requests to be overlapped. For example, a processor may try to read from a block and another processor may try to write to the block. Both the read and read exclusive requests to the same block may occur simultaneously in the system. If not handled correctly, overlapped processing of requests may produce incorrect results.

To illustrate the danger of not handling overlapped processing of requests cor-

rectly, let us consider one example, which is illustrated in Figure 11.8. Suppose that we have two nodes making requests to a single block: node A and B. Home node H keeps the directory information for the block. Let us assume that initially the block has a directory state Shared. First, node A makes a read request (Circle 1), and since the block is clean, the directory replies with the data (Circle 2). Unfortunately, the ReplyD message is delayed in the network, e.g. due to traffic, link failure, etc. In the mean time, the directory sets the bit corresponding to A in its sharing bit vector to indicate that now A is a sharer of the block. Then, node B makes a read exclusive request to the directory (Circle 3). Because the directory thinks that the block is cached by A, it sends an invalidation message to A (Circle 4). The invalidation message arrives at A sooner than the data reply message that was sent earlier. This is a possibility in a network that allows messages between a pair of nodes to travel using different paths. Hence, the order in which messages are received may be different than the order in which they are sent.

Figure 11.8: An example of a protocol race that arises from overlapped processing of requests. This race is called the *early invalidation race*.

Let us explore the situation in more detail. The problem that arises as a result of the race is with regard how A should respond to the invalidation message that it receives. From the point of view of the directory, it saw the read request before it saw the read exclusive request. However, because the processing of the prior read request is not complete (A has not received its data reply yet) by the time the directory starts serving the next request, then the processing of both requests is overlapped. The desired outcome is for node A to see the same ordering of the requests as seen by the directory. However, instead, A receives the invalidation message before its data reply, the opposite order that the directory saw. Since A does not know the ordering that the directory saw, it does not know how to respond to the invalidation message.

Without overlapped request processing, when node A receives an invalidation

message to a block that it cannot find in its cache, it knows that the invalidation is to a block that it already evicted (as discussed in Section 11.4.1). The correct response in this case is to acknowledge the receipt of the invalidation message because it no longer has the block, as if it has just invalidated the block. However, with overlapped request processing, receiving an invalidation message to an uncached block may be due to the early invalidation race shown in Figure 11.8. If it applies the same response (replying with an invalidation acknowledgment), the outcome is incorrect. When A acknowledges the invalidation message, it acts as if the write by B has fully propagated. But later, when A receives the data reply message (Circle 2), it places the block in its cache, and allows the processor to read from the block that is supposed to have been invalidated! The final outcome is that to the directory, the read request by A occurs before the write request by another node, while to node A, the write request by another node occurs before its read request.

How can we remedy this situation? The brute force approach will be just to avoid overlapped processing of multiple requests to a block altogether. This can be achieved by serializing the processing of requests at the home node. For example, the directory can require node A to acknowledge the receipt of the data reply it sent out. Before the directory receives this acknowledgment from A, it refuses to service other requests, for example by sending a negative acknowledgment (NACK) to B. Unfortunately, this serialization may significantly delay the processing of the read exclusive request from B and significantly degrade performance. Hence, we need to a better solution that minimizes the serialization of request processing at the directory.

Approaches to Overlapping Request Processing

In order to figure out how much processing of requests can be overlapped at the directory, first we need to be able to distinguish when the processing of two requests is overlapped and when it is not. In other words, we need to know the start and end time of request processing. If the end time of the current request processing is greater than than the start time of the next request, then their processing is overlapped; otherwise it is not. What is the start time and end time of request processing? It depends on the type of the request, and how the request is handled. Before going into a more specific answer, let us first consider the need to know which is the earlier request and which is the later request out of two requests to a single memory block.

In a bus-based multiprocessor system, the bus provides a medium that determines the order of two requests. Since all read misses and writes to a shared block must go to the bus, one request will be granted the bus sooner than the other, and therefore the bus access order provides a consistent view of the order of two requests as seen by all processors. In a DSM multiprocessor, no medium is shared and visible by all processors. A request first needs to be sent from the requestor to the home node where the directory is kept. Since requests can be generated from different nodes, it is practically impossible to prevent different requests from being simultaneously issued by different nodes. Therefore, the time at which the requests are issued cannot be used for ordering the request. However, since all of the requests will eventually reach a single node (the home node), the home node can take the role of "deciding" the order of requests to a single memory block.

Relating back to determining when processing of a request starts and when it ends, we can now conclude that the processing of a request starts when the directory receives the request, and not when the request is sent by the requestor. Consider Figure 11.9, showing the processing of a read request. The top part of the figure shows a read request to a block that is clean, i.e. the home has the most up-to-date value of the block, hence it can reply to the request with the data block. The bottom part of the figure shows a read request to a block that is exclusively cached by another node. In this case, the home responds by sending an intervention message to the current owner, which in turn responds by sending its block to the home (to update the copy at home) and to the requestor (so the requestor can obtain the latest copy as well).

The important question is from the start of the processing of the request, when can the processing be considered complete? An intuitive answer to this is when all nodes involved in the processing have received the message they are supposed to receive, and they have notified home of the receipt of such messages. We refer to this as a *home-centric* scheme, since the home in the end decides when the processing of request is completed. The completion of request processing for the home-centric scheme is shown in the dashed line in Figure 11.9. The drawback of the home-centric scheme is that the processing period is long. The home must wait until the requestor has sent an acknowledgment ("Done") of the relevant message before considering the processing is complete. In the mean time, it cannot serve a new request. Such a lengthy processing period and serialization can seriously degrade performance. Hence, a scheme that minimizes the processing period is needed.

Figure 11.9: The processing period of a Read request.

An alternative scheme is to use the requestor's help to ensure the ordering of request processing. Note that without modifications at the requestor's cache coherence controller, the protocol race illustrated in Figure 11.8 will occur instead with this scheme. At the requestor, we must now require the coherence controller to maintain an **outstanding transaction buffer** which keeps track of requests that the node has sent but has not received a response yet. For example, if a node has sent a Read request to home, it will keep track of the request in the buffer. If it is the first request that the directory decides to process, the node can expect to receive a data reply from home. Otherwise, if the directory is already processing another request, it defers processing this request. The requestor will get a NACK message instead. Therefore, the requestor must wait until the reply (either data or NACK) arrives. If, instead of such reply, it gets another type of message, then it can conclude that there has been a protocol race. It should not handle such a message until either the data reply or NACK is received. Hence, it must keep a buffer that can hold messages that it cannot serve right away. While this *requestor-assisted* scheme introduces extra hardware cost at the cache coherence controller of each

node, it enables the home to complete request processing much sooner than in the home-centric scheme. For example, for the read to a clean block request, as soon as the directory sets the sharing bit for the requestor, changes the state if necessary, and sends a data reply message to the requestor, the request processing can be considered complete (Figure 11.9).

For a read request to a block in state EM, the processing period is different because the final state at the directory is Shared, but to be in the Shared state, the directory must have a clean copy. Therefore, even for the requestor-assisted scheme, the request processing is considered complete only when the home node receives the flush message from the current owner. At that time, the home node can update its copy, and with a clean copy, it can start processing new requests. Note that since the processing in this case is not instantaneous, the directory must keep track of the entire processing period so that it can either buffer new requests and serve them later, or reply with NACKs to new requests so that the requestor can try them at a later time. In order to keep track of the ongoing processing period, the directory state can transition to a special state referred to as the **transient** or **busy** state. A transient state is needed to serve a read request to a block in state EM, but not needed for a read request to a block in the Uncached or Shared state since the processing ends instantaneously (Figure 11.9).

Figure 11.10 shows the processing period of a read exclusive request for both the home-centric and requestor-assisted scheme. The top part of the figure shows a read exclusive request to a block that is currently uncached. The home responds to this request by sending a data reply message. The second part of the figure shows a read exclusive request to a block that is currently in state Shared in the directory. The home responds to this request by sending invalidation messages to all sharers, which in turn send invalidation acknowledgment messages to the requestor. The third part of the figure (without WB race) shows a read exclusive request to a block that is exclusively cached by another node (the directory state is EM). The home responds to this request by sending an invalidation message to the owner, which in turn responds by supplying data to the requestor. The bottom part of the figure (with WB race) will be elaborated on later.

For the home-centric scheme, the processing of a request is considered complete when the requestor has received all messages that it waits for, and after the home receives an acknowledgment ("Done") from the requestor of the receipt of such messages. Note that the home-centric scheme delays the processing of a new request significantly, since the processing involves several steps and potentially

Figure 11.10: The processing period of a ReadX request.

a large number of nodes. Thus, a requestor-assisted scheme is again a more attractive solution. In the requestor-assisted scheme, the read exclusive request is complete immediately after the directory has modified its state and sharing vector

to reflect the requestor as the new owner of the block, and also after the directory has sent the data reply or invalidation messages to current sharers or to the current owner. As before, to support the requestor-assisted scheme, the cache coherence controller at each node must have an ability to defer the processing of an incoming message until it receives the response message that corresponds to its outstanding request.

For the first case (ReadX to Uncached block), it must defer the processing of a message other than the data reply or NACK from the home node. For the second case (ReadX to Shared block), it must wait until it gets a NACK from the home node, or an invalidation acknowledgment (InvAck) from all current sharers. How can the requestor know how many invalidation acknowledgment messages to expect? While not shown in the figure, this requires the home node to send a reply message to the requestor containing the number of sharers it finds in the sharing bit vector. For the third case (ReadX to a block in EM state), the requestor should wait until it either gets a NACK or data flush from the current owner of the block. Note that overall, the requestor does not know in advance which case it will encounter at the directory. Hence, after making a read exclusive request, it can wait until it gets a NACK, a data reply, or a reply from the home that indicates it should wait for an InvAck or Flush from another node. All other message types must be deferred.

Note, however, for the last case, there is an additional complexity. We have treated the protocol race due to early eviction write backs (Section 11.4.1) and due to overlapped processing separately, where in reality both can occur simultaneously. While there are several such possible cases, we will discuss one of them (related to the case shown at the bottom of Figure 11.10). The race occurs when the home sends an invalidation message to the current owner based on its knowledge about the node being the owner of the block, while the owner has actually evicted the block. In that case, when the owner receives an invalidation message from home, it no longer has the block as it has written the block back (or evicted it silently if it was clean). Hence, it cannot supply the block to the requestor. In such a case, the home must supply the block to the requestor when it receives the flushed block. Therefore, it cannot consider the processing complete until after it receives either the acknowledgment from the owner or the written back block from the owner, because it may still need to send the block to the requestor [1].

[1] In the Origin 2000 coherence protocol, rather than waiting for the owner's reply, the home node speculatively sends the block to the requestor as soon as it receives the ReadX request. This way, if a clean block was evicted early, the owner can notify the requestor that it no longer has the block. At

Finally, to handle the long processing periods in several cases, the directory state needs to be augmented with transient states. One state is needed for handling the read to a block in EM state. Since the final directory state is Shared, we will refer to this transient state as "EM2S" state. Another state is needed for handling a read exclusive request to a block in EM state. Since the initial state is EM and the final state is also EM (but with a different owner), we refer to this state as "EMA2EMB" or "EMB2EMA" state. Adding these transient states to the state diagram of the directory coherence protocol shown in Figure 11.7 results in a more complete protocol shown as follows in Figure 11.11.

Figure 11.11: A more complete directory state finite state coherence protocol diagram, showing the handling of several protocol races.

In the figure, transient states are shown in rectangles. New transitions into and out of transient states are shown in dashed lines, and the events that trigger such transitions are shown as labels of the dashed lines. To reduce clutter, labels for other state transitions are not shown (they have been shown in Figure 11.7).

that time, the requestor already has the valid copy of the block that the home sent earlier. It does not need to suffer from an additional latency for waiting the home node to send the block.

Note that in the EM_A state, if there is a read request by a different node (denoted by B), the directory state transitions to a transient state EM2S. In this state, the processing of the request is considered not complete, so any new requests to this memory block are responded to with a NACK message. When the flush message by node A is received by the home, the directory considers the request complete and hence the state transitions out of the transient state to the Shared state. A similar response occurs when the starting state is EM_B instead.

If while in the EM_A state, a read-exclusive request by a different node arrives at the directory, the directory state transitions to a transient state EMA2EMB. In this state, the processing of the request is considered not complete, so any new requests to this memory block are responded to with a NACK message. When the flush message by node B is received by the home (indicating that B wrote back the block early before it received the invalidation message), the directory considers the request as complete and hence the state transitions out of the transient state to the EM_B state. Alternatively, when the acknowledgment message by node B is received by the home (indicating the receipt of the invalidation message, including when B had silently evicted the clean block before it received the invalidation message), the directory considers the request as complete and hence the state transitions out of the transient state to the EM_B state. A similar response occurs when the starting state is EM_B instead.

Overall, we have shown one example of how a coherence protocol for a DSM system is implemented, including how an important subset of protocol races are handled. Some protocol races are omitted from the discussion, such as when a write back occurs when the directory is in a transient state. The approach discussed, which uses a combination of a limited use of requestor-assisted approach, and the reliance on using negative acknowledgment, is similar to the coherence protocol that is implemented in the SGI Origin 2000 [Laudon and Lenoski, 1997]. We note that this is not the only possible approach. A different protocol may be constructed when we rely more on the home-centric approach, with the obvious implication of the longer time to process a request. Alternatively, a different protocol may also be constructed when we rely even more on requestor-assisted approach. This may eliminate the transient states, and the negative acknowledgments that correspond to these states. The right choice depends on many parameters, such as the throughput of the node coherence controller, and the availability of bandwidth interconnecting the processors.

■ *Did you know?*

One design example that took a more extreme requestor-assisted scheme is the AlphaServer GS320 system [Gharachorloo et al., 2000]. It removes the transient states and negative acknowledgments associated with them by relying on the requestor to correctly order the processing of its request with other requests. A good discussion on the benefits and drawbacks of these approach can be found in [Chaudhuri and Heinrich, 2004].

The types of protocol races that can occur also depend on the property of the interconnection network. If the network guarantees that two messages sent by a node to the same destination node arrive in the same order as they were sent, there are fewer cases of protocol races. For example, the race illustrated in Figure 11.8 cannot occur because the invalidation message will not arrive at node A before the data reply message arrives at node A. But races due to an out-of-sync view of cache states at the directory remain, and need to be handled correctly.

11.4.3 Write Propagation and Serialization

In this section, we will discuss the reasoning behind the correctness of the invalidation-based directory coherence that was discussed in the previous section. First, recall that there are two requirements for supporting a correct coherence protocol: *write propagation*, which allows a write to propagate the new value written, and *write serialization*, which ensures that two writes are seen in the same order by all processors.

As with a bus-based multiprocessor, ensuring write propagation is simply achieved using the invalidation protocol. When a processor wants to write a block, it sends a ReadX or Upgr request to Home, which in response sends invalidations to all other sharers. This forces them to suffer a cache miss when they try to access the block in the future. In addition, on a cache miss to a recently-invalidated block, the directory finds the most recent copy of the block by by looking up its state and the sharing vector, and if the block is dirty, Home requests the processor that has the block to supply the block to the requesting processor.

Write serialization is a subset of the issue of avoiding overlapped request processing, which in this case, involves the processing of two write requests. With the home-centric approach, the overlapping is avoided completely at the home node. With the requestor-assisted approach, the overlapping is avoided by the cooperation of the home and the requestor. Figure 11.12 illustrates the case in which three

nodes (A, B, and C) want to write to a single block. Each of them issues a ReadX request to home. Recall that the processing period for such a request at home is from the time home receives the request until home obtains an acknowledgment message from the current owner (the bottom part of Figure 11.10). Thus, in this case, while the directory is still processing the ReadX request from A, and has not received the acknowledgment message from node O (current owner), the directory state enters a transient state and either sends NACKs to incoming requests (from B and C) or buffer them for later processing.

Figure 11.12: Write serialization with a requestor-assisted scheme.

Thus, ReadX from B cannot be processed until the home gets an acknowledgment message from O, at which time the ReadX from A is considered complete. Then, while the ReadX request from B is served, the directory again reenters a transient state and defers the processing of the ReadX request from C, until an acknowledgment is received from A. At that time, the processing of the ReadX request from B is considered complete. Only then, the ReadX request from C can be processed.

Note, however, that the requestor must still assist home in avoiding ReadX request processing overlap. For example, the Flush message sent by node O to node A may arrive later than the invalidation message sent by home to A when home is processing the ReadX from B. Consequently, in a requestor-assisted approach, node A defers the processing the invalidation message until after the flush message from O is received. Similarly, when the ReadX of B is being processed, B defers processing the invalidation message from home until after it receives the flush message from A; when the ReadX of C is being processed, C defers processing the invalidation message from home until after it receives the flush message from B; and so on.

■ *Did you know?*

An interesting alternative for handling coherence protocol races and enforcing write serialization is through the use of token coherence *[Martin et al., 2003]. In a token coherence protocol scheme, each block is associated with a limited number of tokens equaling the number of nodes in the system. Each operation is allowed to proceed if it has obtained a sufficient number of tokens. A read operation requires only one token, but a write operation requires all tokens before it can proceed. With token coherence, protocol races manifest in some operations not obtaining all the tokens they need. For example, if there are two write operations that are overlapped, both writes may not get a sufficient number of tokens to proceed, hence they fail and retry. In essence, handling races becomes simpler but there is a possibility of starvation in which nodes keep failing their operations and keep retrying. Hence, there needs to be a mechanism to detect a repeated failure and arbitrate competing operations to guarantee forward progress.*

11.4.4 Synchronization Support

Synchronization support in a DSM system is similar to that in a bus-based multiprocessor, which at least includes some kind of atomic memory operations, which are implemented on top of the coherence protocol.

Atomic operations are easy to support with the ability of the directory to avoid overlapped request processing utilizing transient states. For example, consider a system that does not provide an LL/SC instruction pair. To implement an atomic read-write-modify sequence, the node can send a special request to the directory. Upon the receipt of this request, the directory can invalidate all sharers, and enter a transient state. During this time, all new requests to the block, including ones due to atomic operations, are negative acknowledged. When the requesting processor has received all invalidation acknowledgments, it performs the atomic operation. After the atomic operation completed, it can send another message to home signaling that its atomic operation is completed. Having received this message, the directory knows that the atomic operation has completed, hence the directory state can transition out of the transient state to an EM state. It can then serve new requests. Using such a mechanism, we can ensure the processing of atomic operations are not overlapped with one another.

If an LL/SC instruction pair is supported, then no special requests or a transient

state are needed at the directory. All the directory needs to do is just to perform its normal cache coherence protocol processing.

Interestingly, however, a DSM system can provide an opportunity to minimize the number of invalidations by performing an atomic operation locally (*in-memory*). To achieve this, first the synchronization variables must be placed in an uncacheable page, so that they are never cached by the nodes. Then, an atomic operation executed by a node incurs a special message to the directory that includes the type of operation to be performed and the address for which it is to be performed. The directory can fetch the block, perform the operation, and place the block back in memory. If there are multiple such requests, the directory performs them one at a time sequentially. Such in-memory synchronization greatly limits the amount of traffic generated on a highly-contested synchronization, since each atomic operation no longer generates invalidation messages. The drawback of in-memory synchronization is that in the event of low contention, a cacheable synchronization variable may perform better. Moreover, when the synchronization variable has strong temporal locality, such as when a thread repeatedly acquires and releases a lock without interference from other threads, a cacheable synchronization variable will perform better.

■ *Did you know?*

An in-memory synchronization mechanism was supported in the SGI Origin2000 distributed shared memory multiprocessor [Laudon and Lenoski, 1997], which at that time (in 1998) was the largest shared memory multiprocessor system ever made, with 128 processors. A highly-contended cacheable synchronization can generate, in the worst case, a very high amount of coherence traffic in such a large system. Hence, an in-memory synchronization mechanism is provided.

Without an in-memory atomic operation, however, the array-based queuing lock (ABQL) will generate less traffic than other schemes for highly contended locks, at the expense of higher uncontended synchronization latency.

Finally, centralized barrier implementations can become expensive quite quickly as the size of DSM increases, both due to the increasing number of nodes that must be invalidated, and due to the increasing distance between nodes where invalidation messages must travel. Therefore, a special barrier network becomes more appealing (Section 9.2.3).

11.4.5 Memory Consistency Models

To support sequential consistency (SC), memory accesses coming out of a node must be issued and completed in strict program order, i.e., the start of a read/write (to any address) must be after the completion of the previous read/write (to any address). Detecting the completion of a memory access is the primary challenge in enforcing SC. The completion of a read is simply when the data that needs to be read is returned to the processor from the cache. The completion of a write can be detected when all sharers have sent acknowledgment of the receipt of invalidation messages. Since the requestor eventually has to be notified of the completion of the write (so that it can write to the block and start processing younger requests), it makes sense to collect the invalidation acknowledgments at the requestor directly, rather than at the home which still has to send a message to the requestor after they are fully collected. As described in Chapter 10, the performance of SC can be improved by using load prefetching (so that a read can likely complete faster by finding the data in the cache), and by using read-exclusive prefetching (so that a write is likely to have the block already cached in an exclusive state and can complete faster). In addition, load speculation can be employed. In a DSM system, those optimizations apply as well.

Relaxed consistency models improve the performance of SC by allowing memory accesses to be overlapped and occur out of program order. The larger the DSM system, the larger the advantage of relaxed consistency models over SC, due to higher latencies to process requests. With higher latencies due to the distance between nodes, it is more important to hide them by overlapping their execution.

11.5 Exercise Problems

1. (*20 Points*) **Storage Overheads**.

   Suppose we have a cache-coherent shared memory machine and we want to implement various directory-based protocol. Assume a 64-byte cache line, 1 GB main memory. Each node consists of only one processor. State the storage overhead in the directory of a 16, 64, 256, and 1024 processor machine for the following directory-based protocols:

   (a) Full-bit vector

   (b) Coarse vector with 4-processors per bit

   (c) Limited pointer with 4 pointers per memory block with broadcast (also called Dir_4B, where B means broadcast on pointer overflow)

   Note that the storage overhead should be calculated as the ratio of the number of bytes required for the directory over the number of bytes of the main memory. Use the following template:

Scheme	16-proc	64-proc	256-proc	1024-proc
Full bit vector				
Coarse vector 4 procs/bit				
Dir_4B				

2. (*45 Points*) **Coherence Protocol Operations**.

   Following the example in the class, assume a 3-processor multiprocessor system with directory-based coherence protocol. Assume that the cost of a network transaction is solely determined by the number of sequential hops involved in the transaction. Each hop takes 50 cycles to complete, while a cache hit costs 1 cycle. Furthermore, ignore NACK traffic and speculative replies.

   Display the state transition of all the 3 caches, the directory content and its state, and the network messages generated for the reference stream shown in the tables.

   (a) Assume a full-bit vector directory scheme. Each cache has MESI states, and the directory has 3 states: U for unowned, EM for exclusive or modified, and S for shared.

MemRef	P1	P2	P3	Directory	Messages	Cost
r1						
w1						
r2						
w2						
r3						
r1						
w1						
r1						
w2						
					TOTAL	

(b) Assume a full-bit vector directory scheme. Each cache has MESI states, and the directory has four states: E for exclusive, M for modified, S for shared, U for unowned. You can assume a new message type "UpgrAck" that is used by the home to reply to an upgrade request from the owner of a cache line.

MemRef	P1	P2	P3	Directory	Messages	Cost
r1						
w1						
r2						
w2						
r3						
r1						
w1						
r1						
w2						
					TOTAL	

3. (*15 Points*) **TLB Coherence**.

Researchers have noticed that the a similar coherence problem exists for TLB entries in multiprocessor systems. If a page address translation changes while the translation is cached in the TLBs of some processors, the change must be propagated to the affected TLBs. One way to do it is to flush the TLB content (either the entire TLB or only the affected translations).

For a CC-NUMA multiprocessor with software-reloaded TLBs, suppose a

page needs to be migrated. Explain the detailed trade-offs of the following schemes: (i) only TLBs that currently have an entry for a page, (ii) only TLBs that have loaded an entry for a page since the last flush, or (iii) all TLBs in the system.

Hint: consider at least the following two criteria: the cost of doing the actual TLB flush and the difficulty of tracking necessary information to implement the scheme.

4. **Overlapped processing.**

Suppose a 4-processor multiprocessor system uses a directory-based coherence protocol with full bit vector. The directory keeps U, EM, and S states, while the cache maintains MESI states. Assume that cost of a network transaction is solely determined by the number of protocol hops involved in the transaction, and each hop has a latency of 50 cycles.

Suppose that a parallel program incurs the following accesses to a single block address: r1, r2, w3, and r4, where r indicates a read request, w indicates a write request, and the number indicates the processor issuing the request. Suppose that the requests are issued almost simultaneously to the directory, but the directory receives them in the following order: r1, r2, w3, r4. Assume that the occupancy of the directory (i.e., the length of time the directory looks up and updates the directory state) is 10 cycles.

(a) What is the latency to complete the processing of all the requests using a home-centric approach?

(b) What is the latency to complete the processing of all the requests using a requestor-assisted approach, which tries to overlap the request processing as much as possible? Assume that the occupancy of the directory (i.e., the length of time the directory is unable to serve a request) is 10 cycles.

Chapter 12

Interconnection Network Architecture

Contents

12.1 Link and Channel 363
12.2 Network Topology 365
12.3 Routing Policies and Algorithms 371
12.4 Router Architecture 378
12.5 Exercise Problems 381

So far our discussion has been centered on how to build a correct and efficient shared memory multiprocessor, by providing cache coherence, memory consistency, and synchronization primitives. We have assumed that messages are reliably sent from one processor to another with a low latency. However, we have not discussed how messages can be sent reliably and fast from one node to another.

The objective of this chapter is to discuss the fabric that interconnects multiple processors. Two of the most important performance metrics for an interconnection network are **latency** and **bandwidth**. A shared memory multiprocessor has unique requirements for interconnection fabric performance, compared to other systems, such as Local Area Networks (LANs) or the internet, due to the several unique characteristics of communication in a shared memory multiprocessor. One unique characteristic is that messages are very short. Many messages are coherence protocol requests and responses not containing data, while some messages contain a small amount (cache-block size) of data, which is 64 or 128 bytes in current systems. Secondly, messages are generated frequently, since each read

or write miss potentially generates coherence messages involving several nodes. Finally, since a message is generated due to a processor read or write event, the ability of the processor to hide the message communication delay is relatively low. Therefore, shared memory multiprocessor interconnection fabric must very low latency and high bandwidth communication between nodes, optimized for handling small and almost uniformly-sized packets.

Such considerations carry several implications. First, while general networks such as TCP/IP rely on many layers of communication protocols to provide a flexible functionality, shared memory multiprocessor interconnection network should be designed with as few layers of communication protocols as possible in order to minimize the communication latency. For example, a shared memory multiprocessor interconnection network may use only two protocol layers: *link-level* protocol to ensure reliable delivery of a single packet over a single link, and *node-level* protocol to ensure reliable delivery of a packet from one node to another node. Secondly, communication protocol should be stripped down to essential features. For example, complex policies, such as end-to-end flow control, and flow control involving packet dropping, can be harmful to performance. Instead, flow control is only performed at the link level, to avoid overflowing the buffers at the receiver. In addition, packet dropping should typically be avoided.

Another unique aspect that needs to be considered in designing the interconnection network for a shared memory multiprocessor is how it interacts with other components of the multiprocessors such as coherence protocol and consistency model.

Overall, choosing the right interconnection fabric design for a given shared memory multiprocessor is a challenging task. It is often the case the optimal interconnection network for a particular shared memory multiprocessor system is not optimal for other shared memory multiprocessor systems, due to the difference in performance, scalability, and cost requirements. To help understanding such trade-offs, we will discuss the components of interconnection fabric, and various policies that are applicable for each component, and their implications on performance of the system.

Interconnection network glues multiple processors together to form a shared memory multiprocessor system, as shown in Figure 12.1. A processor, or a group of processors, are encapsulated into a **node**. A node consists of the processors, their caches ($), memories (M), communication assist (CA), which is the logic that interfaces with the network fabric, and a **router**, which connects to other

routers. In a small network, sometimes one router is sufficient to connect several processors together. In larger networks, multiple routers are necessary and they must connect to the communication assist of individual nodes as well as to other routers. In the example in the figure, each router connects to four other routers. The physical wires between two routers are referred to as a **link**. A link can be **unidirectional** in which data can only be sent in one direction, or **bidirectional** in which data can be sent in both directions. One important characteristic of an interconnection network is the shape of the network, referred to as the network **topology**. For example, the network topology shown in the figure is a two-dimensional mesh. In the following sections, we will discuss each of these components in more details.

Figure 12.1: Components of an interconnection network.

12.1 Link and Channel

A link is a set of wires that connect two nodes. The minimum amount of data that can be transmitted in one cycle is called a **phit**. A phit is typically determined by to the width of the link. For example, a 10-bit link may have a phit of 1 byte (8 bits) with the remaining 2 bits used for control signals or parity information. However, data is transferred at the granularity of link-level flow control unit called a **flit**. A **flit** worth of data can be accepted or rejected at the receiver, depending on the flow control protocol and amount of buffering available at the receiver. The flow control mechanism at a link/channel typically relies on the receiving buffer to send a continuous stream of signals to the sender that tells it to send more flits

(if there is much room left in the buffer), or stop sending new flits (if there is little room left in the buffer).

A link, the receiver and sender, make up a **channel**. When a large message is transmitted on a channel, it is fragmented and encapsulated into **packets**, which is a unit of transmission in the network. Figure 12.2 illustrates how a long message is broken into packets. Each part of the message becomes a packet **payload**, which is encapsulated by a **header** and **trailer**. The fragmentation allows a message to be transmitted in parts independently of one another, and the header and trailer ensure reliable delivery of the message and contain sufficient information to reassemble the packets into the original message at the destination. When a packet is transmitted over a link, it is transmitted over the link in flit granularity, e.g. multiple flits may be required to transmit a packet.

Figure 12.2: Message and packet.

The header of a packet typically contains routing and control information that enables the router to determine what to do with the packet. By examining the packet, a router knows which link the packet should be sent to next, without having to examine the rest of the packet (payload and trailer). The trailer of a packet follows the packet payload and contains information such as a **checksum**, which is a string of redundant bits that allow the detection or correction of the packet if some bits have changed value due to transmission error. Note that in shared memory multiprocessors, most messages correspond to coherence protocol requests. Hence, they transmit at most one cache block worth of data. Often, one packet is sufficient to send a single message. Only when a large number of bytes are transmitted, multiple packets are required.

The maximum rate at which a packet can be streamed through a link is re-

ferred to as the link **bandwidth**. The link bandwidth is determined by the clock frequency of the link multiplied by the number of bits of data that can be streamed in one cycle.

12.2 Network Topology

Network topology is the shape of an interconnection network, i.e. how nodes are interconnected together. For a small multiprocessor, sometimes a single router is sufficient to interconnect all processors. However, when the number of nodes is larger than what a single router can connect to, routers must be interconnected to each other in a certain formation referred to as the topology of the network.

The choice of topologies is an important one as it determines the latency, bandwidth, and cost characteristics of the network. Several metrics are useful for evaluating the characteristics of latency and bandwidth of an interconnection network topology. One useful metric is the network **diameter**, defined as the longest distance between any pair of nodes, measured in the number of **network hops** (i.e., the number of links that the message must travel between two nodes). Another useful metric is **average distance**, computed by dividing the sum of distances between all pairs of nodes by the number of node pairs. Thus, the average distance of a network is the expected distance between two randomly selected nodes. Another useful metric is **bisection bandwidth**, which is the minimum number of links that must be cut in order to partition the network into two equal halves. As a proxy of cost, two metrics are often used. One metric is the number of links required to form the interconnection network. Another metric is the number of in/out links connecting to each router, which is called the **degree**.

The significance of the diameter is that it resembles the worst case latency that can occur in the network (assuming a minimum routing). While the average communication latency in the network can be controlled through placement of threads in such a way to minimize the distances that occur during communication, for global (one-to-all, all-to-one, and all-to-all) communication, diameter is a limiting factor in the communication latency. The significance of the bisection bandwidth is that it is a single metric that characterizes the amount of bandwidth available in the entire system. Especially, it represents the maximum bandwidth that can sustain global communication in the system. Why does bisection bandwidth matter more than the total/aggregate bandwidth? The aggregate bandwidth is a less representative metric here because it does not account for a set of links that become

■ **Example 12.1** Example of how metrics that characterize an interconnection network are computed.

Suppose we have 6 nodes that are connected through a mesh as shown in this figure.

Assuming each link is bidirectional,

- Compute the diameter and the average distance of the network.
  Answer: The diameter is the maximum distance in the network. The distance between node 1 and node 6, denoted by $d(1,6)$ is one of the largest. Hence, the network diameter is three. To compute the average distance, we need to list all pairs of nodes and their distances, and take their average. The pairs and their distances are:
  d(1,2) = 1, d(1,3) = 2, d(1,4) = 1, d(1,5) = 2, d(1,6) = 3,
  d(2,3) = 1, d(2,4) = 2, d(2,5) = 2, d(2,6) = 2,
  d(3,4) = 2, d(3,5) = 1, d(3,6) = 1,
  d(4,5) = 1, d(4,6) = 2,
  d(5,6) = 1
  The sum of all distances is 24, and the number of pairs is $\binom{6}{2} = 15$, hence the average distance is $\frac{24}{15} = 1.6$.

- Compute the bisection bandwidth of the network.
  answer: We can partition the network in several ways and the bisection bandwidth is the minimum number of links that need to be cut to create the two equal partitions. In this case, since there are 6 nodes, we need to partition them into three nodes each. If we partition them into (1, 2, 3) and (4, 5, 6), we need to cut three links. If we partition them into (1, 2, 4) and (3, 5, 6), we need to cut two links. Therefore, the bisection bandwidth is two times the link bandwidth.

the bandwidth bottleneck in global communication. For example, in a linear array with p nodes, the total link bandwidth is $p-1$ times the link bandwidth, but bisection bandwidth is equal to one link bandwidth. Since global communication must always travel through one link, bisection bandwidth summarizes the bandwidth characteristic of the network better than the aggregate bandwidth.

Several popular network topologies are shown in Figure 12.3. A bus is different than other topologies in that it allows a direct connection from one node

to another, hence it does not require routing. A two-dimensional mesh is related to the linear array in that the linear array can be thought of as a one-dimensional mesh. A cube is essentially a three-dimensional mesh. The same applies to the hypercube, which is a multi-dimensional mesh with two nodes in each dimension. Due to the similarity, such topologies are grouped into a **k-ary d-dimensional mesh** topology family, where d represents the dimensions, and k represents the number of nodes in each dimension. For example, the linear array in the figure is a 5-ary 1-dimensional mesh, the mesh is a 3-ary 2-dimensional mesh, and the cube in the figure is a 3-dimensional mesh with varying arities. The hypercube is always binary (two nodes in each dimension), so it can be referred to as a 2-ary n-dimensional mesh. Similarly, we can always construct a torus from an mesh by adding links between end nodes at each dimension. For example, ring can be constructing by connecting both end nodes of a linear array. The 2-D torus shown in the figure can be constructed from 2-D mesh shown in the figure. Therefore, a related family of topologies is the **k-ary d-dimensional** torus.

The figure also shows a binary tree, which is an example of a **k-ary tree** where $k = 2$. Note that in the tree, nodes are located at the leaves and are not directly connected to each other. Instead, they are connected through intermediate routers shown by the thin rectangles. A tree suffers from a limited bandwidth problem in that root-level links receive more traffic than leaf-level links. In order to avoid the root-level links from becoming the bottleneck in the system, two related topologies can be used. One topology that can be used is a **fat tree**, which has fatter (i.e. having more wires) upper level links compared to lower level links. Another topology that can be used is a **butterfly** network. A butterfly network can be thought of as a tree with replicated upper level intermediate routers and links. Due to the equal number of links and routers at all levels, a butterfly network does not suffer from limited bandwidth at the root level.

What are the diameter and bisection bandwidth of the topologies? For a k-ary d-dimensional mesh, overall there are k^d nodes in the system. The maximum distance in each dimension is one less than the number of nodes in a dimension, i.e. $k - 1$. Since there are d dimensions, the diameter of a k-ary d-dimensional mesh is $d \times (k - 1)$. The bisection bandwidth can be obtained by cutting the network into two equal halves. This cut is performed with a plane one dimension less than the mesh, therefore the bisection bandwidth is k^{d-1}.

For the tree, the maximum distance is when a node reaches another node through the root of the tree. Since the height of a k-ary tree is $log_k p$, then the

Figure 12.3: Various popular network topologies.

diameter of the tree is $2 \times log_k p$. The bisection bandwidth is obtained by cutting the tree into two at the root, hence it is equal to one link. The fat tree has an equivalent diameter but the bisection bandwidth depends on the number of links

at the root level. If the parent link is k-times fatter than child-level links, then the bisection bandwidth is $\frac{p}{2}$.

> ■ *Did you know?*
>
> *A fat tree is difficult to implement because the upper-level links (especially root-level links) must have many wires in order to support a higher bandwidth than lower-level links. In addition, they are often physically longer than lower-level links, reducing the bandwidth they can provide. In recent implementations (e.g. Infiniband), the fat tree use different cables for the upper level links. While lower level links use copper cables, upper-level links use fiber optics, which give several benefits over copper cables: (1) they provide much higher bandwidth than the lower-level links that use copper cables, (2) they provide less bulk and weight for upper-level links and routers, and (3) they provide communication latencies that are low and relatively unaffected by the length of the cable (transmission delay of a copper cable deteriorates quickly with the length of the cable).*

Finally, for the butterfly, the path from a node to any other node is by going from the lowest-level link, up the routers to the uppermost-level link, and to the destination node. Thus, for each pair of nodes, there is only a single path connecting them, and all paths have a distance of $log_2 p$ (equaling its diameter). Its bisection bandwidth is the same with the fat tree, that is $\frac{p}{2}$.

Table 12.1 summarizes the characteristics of different topologies, showing latency-related metric (diameter), bandwidth-related metrics (bisection bandwidth), and the cost-related metrics (number of links and node degrees). The family of k-ary d-dimensional mesh is shown as well as two of its popular instances: 2-D mesh and hypercube.

Let us look at a more specific scenario with regard to the bisection bandwidth and diameter for a subset of the network topologies. Given p nodes, we can choose a low dimensional network with a high number of nodes in each dimension, or a high dimensional network with a low number of nodes in each dimension. To give an illustration of which approach (low dimensional vs. high dimensional) is more scalable, we compare a 2-D mesh versus a hypercube and a butterfly for the same number of nodes. Figure 12.4 shows how they scale in terms of diameter and bisection bandwidth as the number of nodes changes from small (4 nodes) to large (256 nodes). The figure shows that a low dimensional network's diameter

Table 12.1: Comparison of various network topologies.

Topology	Diameter	Bisection BW	#Links	Degree
Linear array	p-1	1	p-1	2
Ring	$\frac{p}{2}$	2	p	2
2-D Mesh	$2(\sqrt{p}-1)$	$\sqrt{p}$	$2\sqrt{p}(\sqrt{p}-1)$	4
Hypercube	$log_2 p$	$\frac{p}{2}$	$\frac{p}{2} \times log_2 p$	$log_2 p$
k-ary d Mesh	$d(k-1)$	k^{d-1}	$dk^{d-1}(k-1)$	$2d$
k-ary Tree	$2 \times log_k p$	1	$k(p-1)$	$k+1$
k-ary Fat Tree	$2 \times log_k p$	$\frac{p}{2}$	$k(p-1)$	$k+1$
Butterfly	$log_2 p$	$\frac{p}{2}$	$2p \times log_2 p$	4

grows quickly as the number of nodes grows, compared to a high dimensional network's diameter. While they start out with an equal diameter on 4 nodes, 2-D mesh's diameter becomes $3.75\times$ that of the hypercube and butterfly (30 vs. 8) for 256 nodes. The figure also shows that a low dimensional network's bisection bandwidth grows much more slowly compared to a high dimensional network as the number of nodes increases. 2-D mesh, hypercube and butterfly start out with an equal bisection bandwidth on 4 nodes, but on 256 nodes, the bisection bandwidth of hypercube and butterfly is $16\times$ that of 2-D mesh (128 vs. 16 links).

The figure shows that the hypercube and butterfly have equal diameter and bisection bandwidth. Hence, their scaling characteristics are similar. The difference between them is that a butterfly has only one path connecting a unique pair of nodes, that goes through $log_2 p$ number of routers, while the hypercube has paths that vary in length depending on the pair of nodes involved in the communication and on the routing algorithm used. Hence, while the hypercube can benefit from mapping threads that frequently communicate to neighboring nodes, a butterfly cannot benefit from thread-to-node mapping.

From performance and scalability perspective, it seems clear that a high dimensional network is preferred compared to a low dimensional network. However, in reality, cost is also an important consideration. For example, in a 2-D mesh, each router is relatively simple and only needs to have a degree of 4. In a hypercube, each router can be quite complex, and needs to have a degree that is

Figure 12.4: Diameter (top) and bisection bandwidth (bottom) comparison for 2-D Mesh (left bar), Hypercube (middle bar), and Butterfly (right bar) for various number of nodes (4, 16, 64, 256).

twice the dimension. For example, for five dimensions, each router has 10 degrees. A router architecture often relies on a crossbar switch to connect input to output ports, so the cost of additional ports increase the switch complexity quadratically. With a butterfly, the degree remains 4, but it requires a lot more routers compared to a hypercube.

12.3 Routing Policies and Algorithms

So far we have discussed the topology of the network. Given a network topology, it remains an open question of how a packet can be sent from one node to another. The relevant issues here are whether a packet transmission is pipelined or not, what paths a packet is allowed to travel, and how path restriction is related to various characteristics of the network.

The first issue deals with when a packet needs to travel for several hops (links), does the packet travel in its entirity from one node to the next, or can it be split across multiple links? In a **store and forward** routing policy, a packet must be fully received (and stored) by a node, before it can be forwarded to the next node. In a **cut-through** routing policy, parts of a packet can be forwarded to the next node even before the packet has fully arrived at the current node. The difference between them is illustrated in Figure 12.5.

Figure 12.5: Illustration of the difference between store and forward vs. cut-through routing.

In store and forward routing (the top part of the figure), all flits of a packet are buffered completely at each node before the first flit is sent out to the next node. In cut-through routing, when each flit is received at a node, it is immediately forwarded to the next node, Thus, a packet may be in transit at multiple links and multiple routers. There are two main advantages of cut-through routing. First, the amount of buffering at each node is reduced because only parts of a packet need to be stored in each node. Buffers are one of the most expensive components in a router, so reducing the buffer space significantly reduces the cost of a router. Secondly, the latency to transmit a packet also decreases. To see why this is the case, assume that the time to transmit one packet over a link is $T_{transmit}$, and the time spent in the router for processing the packet is T_{router}. If a packet travels for h hops, then under store and forward routing the total latency will be the multiplication of the number of hops with the time spent for each hop:

$$T_{stfwd} = h \times (T_{transmit} + T_{router}) \tag{12.1}$$

On the other hand, for a cut-through routing, the time to transmit one packet is the same with the time to transmit the first bit over all the hops, plus the time for the rest of the packet to arrive at the final destination node. This is because the trailing bits of a packet follow the first bit without a lag. The first bit only suffers from the time at each router to route it, while the trailing bits suffer from the transmission delay. Therefore, the time to send a packet on the network is:

$$T_{cutthrough} = h \times T_{router} + T_{transmit} \tag{12.2}$$

Thus, the difference in the latency to send a packet between the store and forward and the cut through routing is $(h-1) \times T_{transmit}$. This implies that the importance of cut-through routing in reducing the packet latency increases when the number of hops is high (such as on a low dimensional network), and also when the transmission time along a link is high (such as when a link is narrow or has a low clock frequency), or when the packet size is large.

A routing algorithm decides the paths a packet can travel from one node to another. There are several categories of algorithms. A routing algorithm is **minimal** if a packet always uses a path that has the least number of hops to the destination. If a packet is allowed to use a longer path (e.g. to avoid traffic) then it is not minimal. A routing algorithm is said to be **deterministic** if it always uses a single path given a pair of sender and destination nodes. Alternatively, a routing algorithm is

adaptive if it can change the path while it is in transit. Finally, a routing algorithm is **deadlock-free** if its routing cannot introduce a deadlock in packet transmission.

> ■ *Did you know?*
>
> *The choice of routing in the interconnection network affects the design of the cache coherence protocol. For example, if deterministic routing is used, then two messages sent from a node to another node will travel on the same path and arrive at the destination in the same order in which they were sent. Consequently, there are fewer protocol races that can occur compared to when non-deterministic routing algorithms are used.*

One popular routing algorithm in a mesh topology family is **dimension-ordered** routing, which requires a packet to travel in one dimension completely before going on another dimension. For example, in a 2-D mesh, a packet must first travel in the west or east direction before it can travel in the north or south direction. Once it travels in the north or south direction, it cannot go back to travel on the west or east direction. On a mesh, a dimension ordered routing algorithm is minimal (because it is guaranteed to use the least number of hops) and deterministic. Such simplicity makes it easy to implement. It is one of the most popular routing algorithms implemented in shared memory multiprocessors that use a mesh topology. However, a dimension-ordered routing policy suffers from several drawbacks. One drawback is that due to its deterministic nature, deterministic routing can lead to contention in certain links. If a link is congested, a packet will wait until the link becomes free, even if there are other possible paths that go through free links. Deterministic routing also leads to the lack of fault tolerance. If link failure occurs, packets that must use that link can no longer be delivered, and the entire system may freeze.

Deadlock Avoidance and Resolution

A deadlock is a situation in which all parties involved are stalled and cannot make any progress. Deadlocks can occur in different components of a multiprocessor, such as in the program due to the order of lock acquisitions, in the coherence protocol due to incorrect handling of protocol races, and in the interconnection network. We will focus on the deadlocks that occur specifically in the interconnection network that manifest in the inability to forward packets. A deadlock in an interconnection network refers to a situation in which packets are stalled and

cannot be sent further on a set of links. It occurs due to limited buffers, and cyclic dependence in buffer space acquisition. Figure 12.6 illustrates such a deadlock. The figure shows four packets being transmitted. Each packet has filled up the input buffer in a channel and needs to be forwarded to the next channel. However, the output buffer in the next channel is already full as it is already occupied by a packet that needs to be transmitted. However, that packet is also stalled because the output buffer in the next node is full, and so on. When such channel acquisition dependence forms a cycle, deadlocks can occur.

Figure 12.6: An illustration of a deadlock situation.

A deadlock can be detected, broken, or avoided by removing the possibility of a cyclic channel acquisition. While there deadlock detection algorithms, they are very expensive to deploy on an interconnection network that requires a very low latency communication between nodes. There are also techniques that can recover from deadlocks such as dropping packets. Dropping one of the packets shown in the figure removes the deadlock and allows all other packets to continue. However, again, such a strategy is not desirable to use for a low latency network because when a packet is dropped, it must be detected (likely via a time-out) and retransmitted. A time-out mechanism and its subsequent retransmission introduce an intolerable delay in a shared memory multiprocessor system in which low latencies are paramount for performance. Thus, an acceptable solution must come either from deadlock avoidance, or from deadlock resolution using redundant resources. We will first look at deadlock avoidance techniques.

One popular deadlock avoidance technique for a mesh-family of topologies is a policy referred to as a **turn restriction** routing algorithm. The basis for the algorithm is based on the observation that in order for a deadlock to occur, all

four anti clockwise turns shown in Figure 12.6 must occur. Similarly, all four clockwise turns can also cause a deadlock. In order to eliminate the possibility of a cycle forming, we should eliminate at least one of the four anti clockwise turns, and at least one of the four clockwise turns. This is why the algorithms are referred to as "turn restriction" routing algorithms.

A dimension-ordered routing naturally restricts all turns from the y-dimension to the x-dimension, hence it is naturally deadlock free. However, it imposes deterministic routing. Relaxing the number of turns that are restricted can still give the deadlock freedom benefit while permitting non-deterministic routing. If we restrict one anti clockwise turn and one clockwise turn, we can come up with $4 \times 4 = 16$ pairs of turns that can be restricted to eliminate the possibility of deadlocks. Figure 12.7 shows three of them. The **west first** routing restricts all turns to the west because the west direction must be taken first before any other directions. The **north last** routing restricts turning to any directions when the current direction is north. Hence, going in the north direction must be taken after all turns are completed. Finally, **negative first** restricts turning to the negative direction in each dimension (i.e., turning west on the x-dimension or turning south on the y-dimension). Any hop in the negative direction must be taken first before taking the first turn.

Turn restriction algorithms not only allow alternative minimum-length paths from one node to another, but they also allow non-minimum length paths between two nodes. Such a property can be used for routing around a link failure as well as for routing around congested links.

Another way to deal with deadlock is to resolve it when it occurs. One way to achieve that is through using redundant channels. For example, suppose that between two adjacent nodes, instead of having one channel, we add a redundant channel. We only use one channel under a normal situation in which there are no deadlocks. When a deadlock occurs, we can utilize the spare channel to route packets that were deadlocked. In order to avoid the possibility of deadlocks from occurring in the spare channels as well, we can put restrictions, such as only allowing one packet to be using the spare channels at a time, or imposing turn restriction on the spare channels, etc. Hence, deadlocks can be resolved without packet dropping.

The key drawback, however, is that spare channels are expensive: they require doubling the number of links, and the buffers in a new channel that corresponds to both ends of a channel. Thus, a router becomes more costly. However, there is

Figure 12.7: Restricting turns to avoid the possibility of deadlocks.

a way to reduce the cost of spare channels. Note that the deadlock occurs due to the buffers in routers becoming full and cannot receive more flits. When such a situation occurs, the flow control at the link level prevents the sender from sending new flits until the buffer becomes free again. But since there is a deadlock, the buffer does not become free. Hence, while buffers are full, links are free and unused in a deadlock. Therefore, spare channels only need to include buffers at the routers but not extra links. Two or more channels have their own buffers but can share a single links. Such channels are referred to as **virtual channels**. Hence, to the network protocols, it is as if there are multiple channels while in fact physically they share common links.

■ **Example 12.2** Example of allowable paths under turn-restriction routing algorithms.

Consider the paths shown in the following 2-D mesh network with bidirectional links:

Determine which paths are allowed under west-first routing and which are allowed under north-last routing.

Answer:
Under *west first* routing: Path 1 is not allowed because it goes to west last, Path 2 and 4 are allowed because they never go west, and Path 3 is allowed because it goes west first before making any turns.
Under *north last* routing: Path 1 and 4 are allowed because they never go north, Path 2 is not allowed because it turns north in the middle and makes a turn east after that, while Path 3 is allowed because the last turn goes north and no more turns are made.

12.4 Router Architecture

So far, we have assumed that given a packet, a router will correctly send the packet to the next channel. But exactly what components are inside a router? Figure 12.8 shows the architecture of a router:

In the figure, the router has four input ports and four output ports corresponding to a total of four physical channels. Each physical channel consists of two virtual channels, with two separate sets of input and output buffers. When a packet is received, based on its header, it is routed to the appropriate virtual channel's buffer. Then, the control logic arbitrates among multiple input virtual channels and allocates one of them to the input of the *crossbar switch*, and allocates the output virtual channel to the output of the switch, depending on the arbitration

Figure 12.8: Router architecture.

policy implemented in the control logic. The control logic also arbitrates among input ports of the switch which want to send flits to the same output port. Once these two arbitrations are completed, the flit can go from the input virtual channel, through the *crossbar switch*, and to the appropriate output virtual channel. A **crossbar switch** is an interconnect that directly connects each input port to all output ports.

Multiple flits (or entire packets if store and forward routing is used) from different virtual channels can be sent simultaneously to the crossbar and to the output ports as long as they do not conflict at various ports. Hence, parallelism is exploited except when the flits suffer from port conflict. In addition, note that routing steps (buffering, virtual channel allocation, switch allocation, and data transfer) allow routing to be pipelined. Hence, for example, while one flit is being transferred, switch allocation for the another flit can be performed, while virtual channel allocation for the third flit can be performed simultaneously.

The cost of a router is a significant fraction of the total cost of an interconnection network. First, buffers take up a lot of die area and consume a lot of power because there are many of them. The die area overhead is equivalent to the number of ports multiplied by the number of virtual channels multiplied by the depth

of each buffer. Secondly, in order to connect any input port to any output port directly, a crossbar switch is often used. A crossbar switch is very expensive when the number of ports is high. In order to connect n input ports to n output ports, the complexity of a crossbar is $\Theta(n^2)$. While the crossbar can be made cheaper by replacing it with a multistage network, the latency increases.

Recall that from performance and scalability perspective, it was clear that a high dimensional network is preferred compared to a low dimensional network. However, the cost of routers is the major impediment to adopting a high dimensional network. Increasing the degree (number of ports) of a router increases the complexity of the control logic for arbitrating and allocating the virtual channels and the switch, and increases the complexity of the switch quadratically.

Interestingly, the cost of links and buffers do not necessarily increase with the higher dimension. While the number of links is higher for a high dimensional network compared to a low dimensional network, the links can be made narrower such that the number of wires is constant (hence the total cost for links can also be made constant). A narrower link also implies a lower link bandwidth, which requires shallower channel buffers. Thus, while the number of buffers increases linearly with the degree of the routers, the total buffer size can be made constant since each buffer becomes smaller.

Overall, in order to allow multiprocessor systems to scale, some of the main challenges in the router architecture is how to keep the cost and latency of the control logic and switches of a router in high dimensional networks.

12.5 Exercise Problems

1. (*30 Points*) **Routing Latency**. Suppose that a multiprocessor system is interconnected using 500-MHz links that are 1 byte wide. We want to measure the maximum latency to transmit a packet between a pair of nodes. Assume that the packet size is 100 byte, and the time spent to route a flit in a switch is 8 clock cycles. Compute the following latencies:

   (a) 16, 64, 256, and 1024-processor systems with 2-D mesh interconnection network topology, using store-and-forward routing.

   (b) 16, 64, 256, and 1024-processor systems with 2-D mesh interconnection network topology, using cut-through routing.

   (c) 16, 64, 256, and 1024-processor systems with hypercube interconnection network topology, using store-and-forward routing.

   (d) 16, 64, 256, and 1024-processor systems with hypercube interconnection network topology, using cut-through routing.

   (e) 16, 64, 256, and 1024-processor systems with butterfly interconnection network topology, using store-and-forward routing.

   (f) 16, 64, 256, and 1024-processor systems with butterfly interconnection network topology, using cut-through routing.

2. (*10 Points*) **Routing Latency**. Suppose that a multiprocessor system is interconnected using 100-MHz links that are 1 byte wide. We want to measure the maximum latency to transmit a packet between a pair of nodes. Assume that the packet size is 1 Kbyte, and the time spent to route a flit in a switch is 20 clock cycles. Compute the following latencies:

   (a) 16, 64, 256, and 1024-processor systems with 2-D mesh interconnection network topology, using store-and-forward routing.

   (b) 16, 64, 256, and 1024-processor systems with 2-D mesh interconnection network topology, using cut-through routing.

   (c) 16, 64, 256, and 1024-processor systems with hypercube interconnection network topology, using store-and-forward routing.

   (d) 16, 64, 256, and 1024-processor systems with hypercube interconnection network topology, using cut-through routing.

(e) 16, 64, 256, and 1024-processor systems with butterfly interconnection network topology, using store-and-forward routing.

(f) 16, 64, 256, and 1024-processor systems with butterfly interconnection network topology, using cut-through routing.

3. (*20 Points*) **Interconnection Network Characteristics**. Compute the (i) switch degree, (ii) network diameter, (iii) number of links, and (iv) bisection bandwidth of:

   (a) Ring network, where the numbers of nodes are 16, 64, and 256.

   (b) 2D-mesh, where the numbers of nodes are 16, 64, and 256.

   (c) Butterfly, where the numbers of nodes are 16, 64, and 256.

   (d) Hypercube, where the numbers of nodes are 16, 64, and 256.

   (e) Fat tree, where the numbers of nodes are 16, 64, and 256.

4. (*15 Points*) **Routing**.

   In the following network, find which routes that are allowed under various routing algorithms:

Allowed in routing?	Route 1	Route 2	Route 3	Route 4
Minimum routing				
X-Y dimension-ordered				
West-First				
North-Last				
Negative-First				

5. (*15 Points*) **Hypercube Topology Characteristics**.

   For a d-dimensional hypercube with $p = 2^d$ number of nodes, prove that:

   (a) Its diameter is $log_2 p$

   (b) Its bisection bandwidth is $\frac{p}{2}$

   (c) Its total number of links (assuming bidirectional links) $\frac{p}{2} \times log_2 p$. (Hint: one helpful approach is to think of how many ways you can cut the network into two partitions with a $(d-1)$-dimensional plane, and how many links are removed by the plane).

   (d) Its router degree is $log_2 p$

6. (*15 Points*) **k-ary d-Mesh Topology Characteristics**.

   For a d-dimensional hypercube with $p = 2^d$ number of nodes, prove that:

   (a) Its diameter is $d(k-1)$

   (b) Its bisection bandwidth is k^{d-1}

   (c) Its total number of links (assuming bidirectional links) $dk^{d-1}(k-1)$. (Hint: this problem is intellectually satisfying for those who are math-inclined. A useful tool to derive this is to use induction. Start by expressing the number of links for a given number of nodes p as a function of the number of links for $\frac{p}{k}$ nodes on a directly smaller dimension. Once you have this function, you can unroll it and express the unrolled function directly as a function of d and k. Enjoy.)

   (d) Its router degree is $2d$

Part III

Designing Multicore Architectures

Chapter 13

Multicore Architecture: Overview and Challenges

Contents

13.1 Measuring Aggregate Multicore Performance 389
13.2 Unique Challenges in Multicore Architectures 394
13.2.1 Cache Organization 395
13.2.2 Performance Volatility 397
13.2.3 Scalability Problems 399
13.3 Exercise Problems 401

The objective of this chapter is to give an overview of multicore architectures, especially on how they differ from traditional (multichip) multiprocessor architectures. A multicore architecture integrates multiple processor cores on a single die. Currently, a multicore architecture is considered the architecture of choice as the number of transistors that can be integrated on a single die has continued to grow at Moore's law speed, roughly doubling every 18 to 24 months. Whereas in the past this increased integration was used to incorporate more components of a single processor core on chip (such as the floating point units), more recently this increased integration is being used to incorporate more cores on a single chip.

At some point in the late 1990s, chipmakers started to realize that improving the performance of a single core was increasingly becoming more difficult, i.e. the performance improvement that can be obtained for a given amount of additional complexity, power consumption, die area, was declining. Consequently, designers needed to find a new approach to improve performance efficiently.

The improvement of performance from a single core system mainly came from two factors: instruction-level parallelism (ILP) and cache memory. With any new technology, there are usually low hanging fruits that can give a large amount of benefits easily. Once the low hanging fruits are taken, it gradually takes more and more complexity to produce the same amount of incremental benefit, a phenomenon referred to as *diminishing return*.

The realization that each additional performance gain comes from an increasing complexity and power consumption leads microprocessor design to shift to a different approach. The multicore architecture provides an attractive approach, as long as performance can scale linearly with the increase in the number of cores. This is predicated upon the ability of software to utilize the number of cores for computation. In the server platform, workloads are often already multithreaded and there are a large number of threads that can keep a large number of cores busy. For example, in a web server, the server spawns a thread to serve every connection from a client. Since there are many client connections for a single web server, there are a large number of threads that can run on the system. In client machines, such as desktops, laptops, and cell phones, there is a dichotomy in the application space. On one hand, many existing applications such as compilers and word processors do not have much thread-level parallelism due to few loop structures in them. However, multimedia applications (image, music, and video processing), applied intelligence (speech, and natural language processing), and data mining exhibit relatively simple computation applied to a large amount of data. Such characteristics result in loop structures with a large number of iterations, which is the type of code that can benefit from loop level parallelization. In general, data parallel applications have a lot of thread level parallelism suitable for execution on a multicore system. Thus, at the time of writing of this chapter, it appears that multicore architecture is a great design approach for both server and some client machines.

Note, however, that with every new technology, there is a limited amount of low hanging fruits. Eventually, as these low hanging fruits are picked up, obtaining the same additional performance becomes increasingly more challenging. In this chapter, we will overview the challenges, and in later chapters, each of these challenges will be discussed in greater details one by one. Note that in contrast to traditional multiprocessors for which the design has relatively matured, multicore architectures are still relatively young. The technology has not matured, and not all design issues and tradeoffs are well understood yet. Thus, the style of writing

will be different than in earlier chapters. Here, our style of writing will be more of presenting challenges and discussing potential solutions of which their relative superiority may not be clear.

The rest of the chapter discusses how aggregate (or system) performance in a multicore architecture should be measured (Section 13.1), and what unique challenges we face when designing a multicore architecture (Section 13.2).

13.1 Measuring Aggregate Multicore Performance

A multicore system can be used to run parallel programs or multiprogrammed workloads in which different cores run different applications with different characteristics. Measuring the system performance for parallel programs can simply be achieved by collecting the wall-clock time. However, how to measure system performance when multiple independent applications run is more challenging.

In the computer architecture community, there is not yet an agreement of what metric should be used to measure the performance of multiprogrammed workloads. A variety of metrics have been used in different studies, such as IPC throughput, weighted speedup [Snavely and Tullsen, 2000], harmonic mean of speedups [Luo et al., 2001], geometric mean of speedups, etc. Their definitions are:

1. **IPC throughput** (IPCT), defined as the sum of instruction per cycles for all programs in the workload, i.e. $IPCT = \sum_{i=1}^{n} IPC_i$, where n is the number of programs in the workload that run together on a multicore system.

2. **Weighted speedup** (WS), defined as the sum of the ratios of IPCs when programs run together (IPC) and when each program runs alone (IPC_{alone}). More specifically, $WS = \sum_{i=1}^{n} \frac{IPC_i}{IPC_{alone,i}}$.

3. **Arithmetic mean of speedup** (Amean), defined as the average of the ratios of IPCs when programs run together and the IPC when each program runs alone, i.e. $Amean = \frac{1}{n} \sum_{i=1}^{n} \frac{IPC_i}{IPC_{alone,i}}$.

4. **Geometric mean of speedup** (Gmean), defined as the nth root of the multiplication of the ratios of IPCs when programs run together and the IPC when each program runs alone, i.e. $Gmean = \sqrt[n]{\prod \frac{IPC_i}{IPC_{alone,i}}}$.

5. **Harmonic mean of speedup** (Hmean), defined as the number of programs divided by the sum of the ratios of IPC when each program runs alone and when programs run together, i.e. $Hmean = \frac{n}{\sum_{i=1}^{n} \frac{IPC_{alone,i}}{IPC_i}}$.

Let us examine the characteristics of these metrics. IPCT is the simplest but may also be a misleading metric to use. In particular, programs with high IPC will have a larger share of the IPCT, and hence these programs disproportionally affects IPCT. An IPC improvement on a high-IPC program carries a disproportionally large impact on IPCT. Likewise, an IPC reduction on a high-IPC program also carries a disproportionally large impact on IPCT.

Weighted speedup normalizes the contribution of each program to the execution time of the program when it runs alone in the system. Hence, it is somewhat less biased than IPCT.

The next three metrics are all various means for summarizing speedup ratios of multiple programs. Arithmetic mean is a regular average of the speedup ratios of all programs. Note that it is identical to the weighted speedup divided by the number of programs. Hence, it tells us the same thing as what weighted speedup tells us. For example, if we compare two techniques and technique A has a better weighted speedup than technique B, it is also true that technique A has a larger arithmetic mean of speedups than technique B. Thus, for comparison purpose, we can treat weighted speedup and arithmetic mean of speedups interchangeably.

Geometric mean and harmonic mean of speedups are different ways to summarize a collection of speedup numbers, as shown in their definitions. Geometric mean takes the nth root of the multiplication of all speedup ratios. Harmonic mean takes the reciprocal of each individual speedup, sums them up, and divides the number of programs by the sum. Geometric mean has been used for reporting SPEC performance numbers, to represent the overall performance of a system against a reference system.

Let us assume a multiprogrammed workload consisting of two programs, as shown in Table 13.1. The table shows the IPC when each program runs alone in the system, the IPC when they run together in the system, and the speedup ratios over the case when each program runs alone. The table shows that initially, the IPCs of App1 and App2 are 2 and 0.5, respectively. When they run together, App1 is slowed down by 50% (its IPC drops to 1), while App2 is sped up by two times (its IPC doubles to 1).

Table 13.2 shows the value of various system performance metrics, when the programs run alone versus after the programs run together. Weighted speedup is

Table 13.1: Example of IPC speedup.

	IPC_{alone}	IPC	$Spdup = \frac{IPC}{IPC_{alone}}$
App1	2	1	0.5
App2	0.5	1	2

skipped since it is already represented by arithmetic mean of speedup. By comparing the value of each metric after programs run together versus before, we can deduce whether running the programs together improves the overall system performance for each metric. The table shows that according to both IPCT and Hmean, the performance is degraded. However, according to Amean, the performance is improved. According to Gmean, the performance is unchanged. Thus, the different metrics support different conclusions. The important question here is which metric is the most reasonable to use?

Table 13.2: Example of various system performance metrics.

Metric	**Value (alone)**	**Value (together)**	**Performance?**
IPCT	2.5	$1 + 1 = 2$	Degraded
Amean	1	$\frac{1}{2}(0.5 + 2) = 1.25$	Improved
Gmean	1	$\sqrt{0.5 \times 2} = 1$	Unchanged
Hmean	1	$\frac{2}{2+0.5} = 0.8$	Degraded

Note that such disagreements are not always the case. If running benchmarks together reduce the IPC of each benchmark, then the difference among the metrics become the magnitude of the performance degradation, rather than the direction of the performance change. However, for the purpose system evaluation, we would like to know which metric best summarizes system performance.

To arrive at a more definitive answer, consider a computer system that executes App1 and App2 during its operation. Suppose that App1 is only run infrequently and executes for only a short time, whereas App2 runs frequently and for a long time each time it runs. It follows that if App1 is slowed down by two times whereas App2 is sped up by two times, then from the perspective of users, the system performance improves. This is an aspect that is not captured in any of the

metrics we have discussed. Consider the following metric: the total number of core-hours used by the execution of the two applications. This metric correlates with the actual resource usage by the applications. Interestingly, whether the system performance improves or degrades when applications run together depends on the original core-hours used by each application when it runs alone. Consider the example in Table 13.3. The table shows two cases. In the first case, both App1 and App2 would run for one core hour alone. When they run together, because of the speedup ratios of 0.5 and 2, respectively, their core hours change to 2 and 0.5, respectively. The total number of core hours now is 2.5, which shows performance degradation compared to when the applications run alone.

Table 13.3: Core-hours of App1 and App2.

	Core-hours (alone)	**Core-hours (together)**	**Performance?**
App1	1	2	
App2	1	0.5	
Total	2	2.5	Degraded
App1	2	4	
App2	8	4	
Total	10	8	Improved

In the second case, App1 and App2 run for 2 and 8 core hours, respectively. When they run together, they both take 4 hours. Hence, the total number of core hours declines from 10 to 8, indicating a reduction in resource usage and hence performance improvement, since it takes less resources to complete both programs compared to the alone case.

Therefore, with the same speedup ratios, the number of core hours can increase or decrease, depending on the execution time when each program runs alone. This is in contrast with the metrics we have discussed which produce one value regardless of the initial execution time of the applications.

The number of core hours can be defined more explicitly as:

$$CH = \sum_{i=1}^{n} W_i \times \frac{IPC_{alone,i}}{IPC_i} \tag{13.1}$$

where W_i is the relative importance of the program, which reflects the original execution time of the program when it runs alone, taking into account the frequency of running the program in the workload.

The difficulty with the core-hour metric is that it requires us to make an assumption on the frequency and the initial execution time of each program that a workload consists of. Such parameters vary from one setting to another, and may vary over time as well. In the situation in which such parameters are unknown, it may be reasonable to give an equal weight to all programs, i.e. assuming that $W_i = 1$. With $W_i = 1$ for all i's, the core-hour metric becomes:

$$CH = \sum_{i=1}^{n} \frac{IPC_{alone,i}}{IPC_i} \tag{13.2}$$

Note that the expression above can be expressed as a function of Hmean, i.e. $\sum_{i=1}^{n} \frac{IPC_{alone,i}}{IPC_i} = \frac{n}{Hmean}$. Based on the assumption that each application takes 1 unit of time to execute, then the core-hour when the applications run alone (CH_{alone}) would be equal to n units of time, whereas the core-hour when the applications run together is $CH = \frac{n}{Hmean}$. If we take the ratio of them, we obtain a speedup of $Spdup_{CH} = \frac{CH_{alone}}{CH} = \frac{n}{\frac{n}{Hmean}} = Hmean$. Thus, the overall speedup ratio based on the core-hour metric, which represents the amount of resources reduced relative to the initial usage of resources to complete the same amount of work, is none other than the harmonic mean of individual application speedups, assuming that the initial execution time of all applications are equal. If the initial execution time of the applications are not equal, the speedup ratio based on the core-hour metric can be computed by:

$$Spdup_{CH} = \frac{\sum_{i=1}^{n} W_i}{\sum_{i=1}^{n} W_i \times \frac{IPC_{alone,i}}{IPC_i}} \tag{13.3}$$

Another metric that also takes into account of relative importance of different programs is job throughput, which is defined as the number of jobs completed per unit time. For a workload consisting of non-uniform jobs, a way to represent job throughput is the *wall-clock time* to fully execute all the jobs, i.e. the difference in the time between the start of the first job and the end of the completion of the last job. Job throughput is a traditional metric that has been used for a long time to evaluate computer system performance. The drawback of job throughput is that the number is meaningful only when there are a large number of jobs. For example, a two-job or a four-job workload is not sufficient to evaluate the performance of a system.

Overall, in order to measure the aggregate improvement in performance, we recommend using either the wall-clock time to complete a collection of applications, or core-hour speedup shown in Equation 13.3. Core-hour speedup is an appealing metric because it does not require a large number of applications before we can compute it meaningfully, and also it corresponds to the reduction in the amount of resources consumed when running the applications individually versus when running the applications together. If the initial execution time for the applications that represent a realistic load is unknown, then we recommend the use of harmonic mean, which is a special case of Equation 13.3. Other metrics, such as the arithmetic and geometric average of speedups, can express the average of a collection of independent speedup numbers, however this average does not directly correspond to the amount of resources needed to run a workload.

13.2 Unique Challenges in Multicore Architectures

A multicore architecture has multiple processor cores implemented on a single die, in contrast to traditional multiprocessor architectures that implement processors on different chips. There are several aspects of a multicore system that make it unique in comparison to traditional multiprocessor architectures, which lead to several interesting questions. For example, how should on-chip caches be organized? Should a cache be private to each core or shared among cores? At what level should we allow cache sharing? Another set of questions deal with fine-grain contention that occur between cores, be it at the shared cache capacity or at the off-chip bandwidth. What are the impacts of such contention on performance? How should contention be managed? Finally, a set of questions deals with identifying problems that restrict the scalability of multicore architecture. In contrast to traditional multiprocessor system where adding more processors implies adding more memory bandwidth, in a multicore architecture, adding more cores on a chip has the opposite effect, that is the amount of off-chip bandwidth available to each core declines. This is referred to as the *bandwidth wall* problem. To what extent does the bandwidth wall problem restrict future scaling of the number of cores in a multicore system? What techniques can mitigate it, and how well do they mitigate it relative to one another? These are challenges that are unique to a multicore architecture, and will be the focus of the next few chapters. In the following subsections, we will overview these challenges at a high level.

13.2.1 Cache Organization

In a multicore architecture, cores are located on a single die. This allows a tighter integration between cores that is impossible to achieve in traditional (multi-chip) multiprocessors. For example, it is not feasible, or at least not practical, to have multiple processors on different chips share on-chip resources such as the L1, L2, or L3 caches. However, it is feasible to have multiple cores on the same die to share some of the on-chip caches in the memory hierarchy, especially lower level caches.

Figure 13.1 shows various possibilities of how the caches are organized with respect to different cores, assuming that there are two levels of caches on chip. Our discussion at this point is on how the caches are *physically organized*, while the logical organization will be discussed later. Physical organization determines the physical proximity of the cache with cores.

One possible cache organization is for the cores to physically share the L1 cache. Such a design is not used in current multicore architectures because L1 data and instruction caches are tightly integrated with a core due to their strict access time requirement. For example, instruction scheduling in the core relies on the L1 cache access time to be low, and may speculatively wake up and issue instructions that depend on a load a fixed number of cycles after the load instruction was issued, corresponding to the L1 data cache access time. Hence, a slight increase in the L1 cache access time has a big impact on the instruction scheduling and the degree of instruction level parallelism that can be exploited. As a result, each core in current architectures is tightly integrated with its L1 caches. In fact, the L1 instruction and data caches are considered parts of the core.

The next possibility is to have physically private L1 caches but a common shared L2 cache. This configuration was used by early multicore architectures, such as the IBM Power4 [IBM, 2002], and is still used by current architectures, such as Sun Niagara [McGhan, 2006]. Note that the bandwidth requirement to the L2 cache is much lower than the bandwidth requirements from cores to the L1 caches since the traffic to the L2 cache has been filtered by the L1 caches. Hence, it is easier to design the interconnection compared to the case where the L1 caches are shared by cores. Typically, the L2 cache is too large to implement as a single bank, so it makes sense to break it up into several cache banks, where each bank has its own read/write port, decoder, and sense amplifier. In order to allow multiple accesses to different L2 cache banks to be serviced in parallel, one can use a crossbar interconnect between the cores and the cache banks. The

Figure 13.1: Various ways to organize the memory hierarchy in a multicore architecture: shared L1 cache (a), shared L2 cache (b), or all-private L2 caches (c). Fatter lines represent higher bandwidth requirements.

complexity of such a crossbar interconnect will be related to $C \times B$ where C is the number of cores, and B is the number of cache banks. Hence, it is straightforward to see that a crossbar interconnection cannot scale to more than a few cores and a few cache banks. While other types of interconnections can be used, they have their own problems as well. For example, a bus only allows accesses to be serviced sequentially, even though they go to different cache banks. Using point-to-point interconnection network is possible, but will make the access time to various banks to differ highly, depending on how many network hops are there between a core and a bank.

Therefore, as the number of cores grows, a physically shared L2 cache becomes less and less attractive. The interconnection to support a very large shared cache is highly expensive, especially in a multicore system where the increase in die area allocated for the interconnection implies that less die area is available for cores or caches. The level of memory hierarchy that is interconnected will thus need to move down to lower levels. As in traditional multiprocessor architecture,

interconnecting a lower level memory hierarchy is simpler because (1) the demand for bandwidth is smaller in lower level caches, and (2) a lower level cache already can tolerate a higher cache access time. Hence, there are several alternative solutions. First, physical sharing can be localized to only a cluster of cores rather than all cores sharing a cache. Going further, we can remove sharing completely by breaking up the cache into separate banks and assign each bank to a core. Finally, we can move sharing to a lower level cache, such as the L3 cache, of which its latency is not as critical to program performance. In addition since accesses to the L3 cache have been filtered by both the L1 and L2 caches, accesses from different cores rarely conflict in the L3 cache, allowing the use of a simpler interconnection.

Overall, while a multicore architecture allows cores to be integrated on a single chip, there many ways of how to interconnect them, depending on the memory hierarchy level that is shared or kept private among cores. Note that we have mostly discussed physical organization of caches. In reality, just like DSM providing the abstraction of shared memory over distributed memory modules, the logical organization of caches may or may not correspond to the physical organization. For example, we can have distributed L2 caches but form an abstraction of a shared L2 cache. On the other hand, we can have a physically shared L2 cache but partition it in such a way that each partition becomes a private L2 cache to the core to which it is assigned. This physical and logical organization of caches will be further discussed in Chapter 14.

13.2.2 Performance Volatility

Sharing on-chip resources also has an implication to the performance of applications running on a multicore system. For example, two applications running on different cores that share an L2 cache may compete for the L2 cache capacity, and some will win in getting more capacity than others. Applications that tend to lose in competing for cache capacity may suffer from a significant **performance volatility**, a situation in which an application's performance varies a lot, depending on which other applications run on the multicore chip simultaneously.

Performance volatility also exists in traditional multiprocessor systems when different applications that share various physical resources, such as the main memory, network I/O bandwidth, and the processor itself (through time sharing). For example, allowing several applications with a large memory footprint to timeshare a processor may result in main memory thrashing, a situation in which system performance (as well as individual application performance) degrades because

the system spends a disproportionate amount of time just performing page swap to/from the disk. Applications that timeshare a processor with small time quanta may also suffer from a large number of cache misses due to them competing for cache capacity to keep their working sets fresh in the cache. We have demonstrated that the number of context switch misses can be quite high and even exceed other types of misses when time quanta for applications that time share a processor is small (Section 6.3). However, contention for such hardware resources is relatively coarse grain in nature, and hence the impact on performance volatility is relatively small for most applications in most situations, and is high only for certain situations. For example, a large number of context switch misses only occur when small time quanta is used. Main memory thrashing only occurs when enough number of applications with a high memory footprint run together in the system. In a low to medium load situation, main memory thrashing is not common. In contrast, the degree of performance volatility due to sharing fine-grain resources, such as on-chip caches and off-chip bandwidth, can be much higher and affects a large majority of applications in a wide range of situations. One reason for this is that on-chip caches are accessed a lot more frequently than coarser grain resources such as the main memory or network I/O bandwidth. In addition, the performance of many applications depends on fine-grain resources to a larger degree than coarse-grain resources.

In addition to hardware resource contention, another source for performance volatility is the contention for software resources, such as locks. When applications run together on the system, they may invoke system calls or library calls, in which the OS or the library protect global variables or data structures with locks. One example is the memory allocation library. If threads of an application perform a lot of memory allocations, in order to keep the main memory management data structure consistent, many memory allocation libraries will enter a critical section when they modify key data structures such as the free lists. Such a critical section makes the library thread-safe, but also introduces serialization that can hurt the performance of parallel applications. To avoid such serialization, a parallel memory allocation library needs to be used. There are many other functions that are protected with critical sections and introduce serialization, such as the OS virtual memory management functions, random number generation functions, etc. Contention for such functions may introduce performance volatility. However, this is not a unique problem to the multicore architecture. To some extent, the problem already exists on traditional shared memory multiprocessor

systems, although its magnitude may be worse in a multicore architecture due to the potentially higher number of threads that run simultaneously in the system. In Chapter 15, our discussion focuses on multicore-specific performance volatility that is caused by sharing of fine-grain resources such as the on-chip cache and off-chip bandwidth.

13.2.3 Scalability Problems

Some on-chip resources, such as the off-chip pin bandwidth, are unavoidably shared among cores. Such off-chip bandwidth sharing is a new problem in multicore architectures. Off-chip bandwidth sharing not only introduces performance volatility, but also presents a scalability problem for multicore systems. In traditional multiprocessor architecture, especially Distributed Shared Memory (DSM) systems, the more processors are added into the system, the higher the aggregate memory bandwidth becomes, since the additional processors come with their own local memories and bandwidth to those local memories. In contrast, adding more processor cores on die does not increase the number of pins used for communicating to components outside the chip, hence increasing the number of cores corresponding to reducing the off-chip bandwidth available to each core. While pin counts and the clock frequency of each pin grow over time, their growth rate is a lot lower than the growth rate of transistors that can be implemented on a die. The projected rate of growth in the number of transistors that can fit on a chip (hence the number of cores) in general is much higher than the projected rate of growth of bandwidth to off-chip memory, roughly 50–70% versus 10–15% per year [ITRS, 2005]. Hence, a unique challenge of a multicore system is how to scale despite the declining off-chip bandwidth per core.

There are other scalability challenges as well. For example, adding more cores on a chip requires at least a proportional increase in the size of the main memory to keep the size of main memory per core constant. However, the main memory size may need to grow faster than Moore's law, due to the continuing increase in software complexity, which employs more layers of abstraction and more modular composition. As a result of the increased complexity, we can expect applications to have larger working sets that need to be kept in a larger main memory. The combination of main memory capacity per DRAM chip growing at Moore's law speed while the demand for main memory capacity growing faster than Moore's law speed will cause the cost of the main memory as a fraction of the cost of the entire system to increase in the future. This makes the main memory design an

increasingly important issue.

There are power concerns as well. While we may be able to continue to shrink the size of transistors at Moore's law speed, the supply voltage cannot be reduced enough to compensate for the increased density. As a result, power density (power consumption divided by unit area) increases to the extent that temperature may rise beyond a safe level. Due to this reason, it is likely that in future multicore systems, not all parts of the chip can be turned on all the time. Cores and cache tiles may be turned on and off alternately. Another concern is that the fraction of power consumption that arises due to leakage current, referred to as static power consumption as opposed to dynamic power consumption due to transistor switching from 0/1 to 1/0, increases. This trend makes it more attractive to increase the utilization of cores as much as possible below the safe temperature threshold, rather than cutting off the clock signal to the cores. However, an increase in core utilization exacerbates the performance volatility problem.

Reliability concerns will also play a significant role in the future. First, soft error rate may increase as transistors become smaller. A soft error is a change in bit value due to a collision of a storage cell with various particles in the atmosphere. The storage cell is still functional but its value has changed. Protecting memory arrays against soft errors is relatively simple (although costly), for example by using error detection and correction code. Protecting logic, such as latches, is more difficult. Secondly, as the size of transistors shrink, the variability in process manufacturing has a magnified impact to transistors. In the future, some parts of the chip may have a shorter life span than others, or cannot tolerate as high clock frequency as others.

Overall, there are many challenges going forward in chip design. However, many of them, such as power, energy, and reliability concerns, are caused by progress in process technology, but not a direct consequence of integrating cores on a single chip. Consistent with our goal of covering only multicore-specific design issues, we choose to discuss only the impact of the bandwidth wall problem in limiting core scaling in Chapter 16.

13.3 Exercise Problems

1. **System performance metrics**. Assume that a workload consists of two applications A and B, which run for x unit time if they run alone in a multicore system. When they run together, application A is unaffected and still takes x unit time to complete. However, application B now takes $k \times x$ amount of time to complete.

   (a) Compute the arithmetic, geometric, and harmonic means of speedups if $k = 2$.

   (b) Compute the arithmetic, geometric, and harmonic means of speedups if $k = 5$.

   (c) Compute the arithmetic, geometric, and harmonic means of speedups if $k = 20$.

   (d) Make observations as to the difference between them for various values of k.

2. **System performance metrics**. Suppose that the speedup ratios of applications A and B when they run together versus when they run alone are denoted by x and y.

   (a) Prove that the arithmetic mean of speedups is always equal or larger than the geometric mean of speedups.

   (b) In what case is the arithmetic mean of speedups equal to the geometric mean of speedups?

   (c) Prove that the geometric mean of speedups is always larger than the harmonic mean of speedups.

3. **System performance metrics**. Assume that a workload consists of two applications A and B, which run for x and $k \times x$ unit time if they run alone in a multicore system, respectively. When they run together, both application A and B now take the same amount of time to complete, at x unit time.

   (a) Compute the arithmetic mean of speedups, geometric mean of speedups, and core-hour speedup if $k = 2$.

   (b) Compute the arithmetic mean of speedups, geometric mean of speedups, and core-hour speedup if $k = 5$.

(c) Compute the arithmetic mean of speedups, geometric mean of speedups, and core-hour speedup if $k = 20$.

(d) Make observations as to the difference between them for various values of k.

Chapter 14

Multicore Memory Hierarchy Organization

Contents

14.1 Physical Cache Organization 405
14.1.1 Physically Shared Cache 405
14.1.2 Physically Private Caches 409
14.1.3 Hybrid Private+Shared Caches 409
14.2 Logical Cache Organization 413
14.2.1 Mapping Function 419
14.2.2 Improving Distance Locality of Shared Cache 421
14.2.3 Capacity Sharing in the Private Cache Configuration . 422
14.3 Keeping Cache Tiles Coherent 426
14.3.1 Broadcast/Snooping Coherence Protocol on Multicore 427
14.3.2 Directory-Based Coherence Protocol 434
14.4 Exercise Problems 438

The goal of this chapter is to overview various memory hierarchy organization choices for multicore architecture. It is important to distinguish between physical versus logical organization of caches.

Physical organization refers to how caches are organized on the die, i.e. whether a core and a cache are directly connected physically. If the cache can be directly accessed by all cores, the cache is said to be **physically shared**. If the cache is physically sliced up into parts where each part is assigned to a core, and a core can only directly access the part of the cache that is assigned to it, the cache parts

are said to be **physically private**. Logical organization refers to which caches a core is allowed to access, i.e. whether a core is allowed to access all caches, its own local cache, or a part of the cache. If all parts of the cache are allowed to be accessed by any core, then the cache is said to be **logically shared**. Physically private caches can be aggregated to form a logically shared, for example by allowing a core to access any part of the cache even when the cache is not directly connected to the core. If only a part of the cache is allowed to be accessed by a core, then the cache is said to be **logically private**. A logically private cache configuration can be built on top of physically private caches by allowing a core to access only the cache that is directly connected to it. A logically private cache configuration can also be built on top of a physically shared cache by dividing the cache into partitions where a core is allowed to access only a certain partition. The distinction between physical and logical organization is quite important to avoid ambiguity when discussing various cache organization choices for a multicore architecture.

We will first discuss the physical organization of caches in Section 14.1. We will discuss how physically shared cache is a suitable design for a small-scale multicore, but is increasingly difficult to scale to a larger scale multicore, due to the cost of directly interconnecting a large number of cores and a large number of cache banks. In Section 14.2, we will discuss various logical organization choices for physically shared caches. In one extreme, we can let the logical organization to exactly follow the physical organization, which is a simple approach. In the other extreme, we can aggregate the entire caches to form a single large shared cache. We introduce a concept called tile associativity that defines which tiles a block address can map to. We will show how the arrangement of tiles in forming tile associativity determines the physical proximity of data from cores. We will also discuss an alternative configuration that falls in between physical and shared logical configuration. This hybrid configuration has several attractive features. We will also discuss some of the research efforts to alleviate the main drawback of a logically shared cache, i.e. the large distance between data and cores, by allowing blocks to be replicated. On the other hand, some research efforts have also attempted to alleviate the main drawback of logically private caches, i.e. the lack of capacity sharing, by allowing neighboring caches to share their cache capacity.

Finally, in Section 14.3, we will discuss challenges in designing cache coherence protocols in a multicore system. We start by discussing broadcast/snoopy protocol over point-to-point interconnection network, using a 2-D mesh network as an example. Due to the nature of broadcast and snooping, such protocols cannot

scale to a large multicore system. Hence, we also describe a directory protocol that can be built over tiled multicore. We discuss that one of the main challenges of such protocols is the physical proximity of the directory information from cores, and describe some research efforts that attempt to alleviate such a problem.

14.1 Physical Cache Organization

14.1.1 Physically Shared Cache

Early multicore designs relied on a physically shared cache organization. For example, IBM's first dual-core architecture, IBM Power4, connects two processor cores with a shared L2 cache. At the time of writing of this book, a shared L2 cache is still a popular implementation in multicore systems, such as in Intel Xeon, and Sun Niagara processors. Other systems, such as AMD quad core and Intel Nehalem, use private L2 caches backed up by a large shared L3 cache.

In the IBM Power4 chip, the L2 cache is fairly large, so it is split into three banks, with each bank having the capacity of 0.5 MB. The L2 cache is physically shared, hence each core can access any cache bank. To achieve fast access for all cores to all cache banks, a crossbar is used to interconnect cores with cache banks. A crossbar has a set of buses that directly connect each core with all cache banks. It allows cores to access different banks simultaneously. Only when two cores access the same bank, their accesses collide and have to be serialized.

With a shared L2 cache, there is no coherence protocol needed at the L2 cache level since each data item stored in the L2 cache can only have one copy. However, coherence still needs to be maintained between the L2 cache and the L1 caches. A coherence protocol for multi-level caches can be implemented, as discussed in Section 8.4.2. To ensure write propagation from one L1 cache to another, when an L1 cache wishes to write to a block, it must notify the L2 cache, which in turn invalidates all copies of the block in other L1 caches. The L2 cache may send invalidation to all other L1 caches, or may send invalidation to a subset of L1 caches if it knows which L1 caches cannot possibly have a copy of a block. The latter strategy can be supported by keeping a duplicate L1 cache tag that the L2 cache can check. Alternatively, a cheaper implementation using snoop filtering is also possible (Section 8.4.3).

To handle an L1 cache read miss, the L2 cache can supply the block if the L2 cache has a fresh copy of the block. If the L1 caches implement a write through policy, the L2 cache always has the latest value of all blocks that it stores. It can

Figure 14.1: High-level diagram of a dual-core chip architecture similar to IBM Power4.

always respond to an L1 cache miss by supplying data. If, however, the L1 caches implement a write back policy, the L2 cache must know whether any of the L1 caches currently stores the most recent value of the block. To support this, the L2 cache can keep keep a duplicate of the tag of each L1 cache. Alternatively, it can keep a special state that indicates which block is dirty at the L1 cache. Overall, keeping L1 caches coherent is relatively simple and inexpensive, if there are not too many L1 caches that share a single L2 cache.

Finally, requests from the L1 cache that involve a single block must be serialized. In this case, since all these requests go to the L2 cache, the L2 cache is a natural serializing point for these requests. It can ensure that all writes and reads from different L1 caches are serialized by ordering their processing. For example, before supplying data for a read request from an L1 cache, if there is a prior write request made by another L1 cache, it must first send invalidation to all L1 caches, and incorporate the new, written-through value to complete the write request pro-

Table 14.1: Illustration of protocol operation for a multicore with three cores, private L1 caches, and a shared L2 cache.

	Request	P1.L1	P2.L1	P3.L1	L2	Data Supplier
0	Initially	–	–	–	–	–
1	R1	V	–	–	E	Mem
2	W1	V	–	–	M	–
3	R3	V	–	V	M	L2 Cache
4	W3	I	–	V	M	–
5	R1	V	–	V	M	L2 Cache
6	R3	V	–	V	M	–
7	R2	V	V	V	M	L2 Cache
8	EvictL2	I	I	I	I	–

cessing. Then, the new value must be supplied to the L1 cache from which the read request originates.

To illustrate how such a cache coherence protocol works in a 3-core system with a physically shared L2 cache, Table 14.1 shows an example. The example assumes that the L1 caches, private to each core, use a write through policy, and keep two states for each block: Valid and Invalid. The L2 cache, which is shared by all cores, on the other hand, uses a write back policy and keeps MESI states.

On a read from Core 1 (the first request), the block is fetched from off-chip memory and placed in the L1 cache of Core 1 in a Valid state, as well as in the L2 cache in an Exclusive state. When Core 1 later writes to the block, since the L1 cache is a write through cache, the write request is sent to the L2 cache. The L2 cache may send invalidation to the L1 caches of Core 2 and Core 3. However, if it keeps duplicate L1 cache tags of all cores, it only needs to check the tags to see if any other L1 caches have a copy of the block, and only sends invalidation to the L1 caches that may have a copy of the block. The write request also writes the new value to the block that resides in the L2 cache, and the state at the L2 cache changes to Modified.

On a read request from Core 3, the L2 cache has the latest value of the data block so it directly supplies the block to the L1 cache of Core 3. When later Core

3 wants to write to the block, the request is sent to the L2 cache, which in turn sends invalidation to Core 1's L1 cache. Core 3 can then proceed with its write, while the block is invalidated at the L1 cache of Core 1. The newly written block value is also propagated down to the L2 cache. Later, when Core 1 wants to read the block, it suffers an L1 cache miss, and the block is supplied by the L2 cache. A read by Core 3, on the other hand, is a cache hit. When Core 2 wants to read the block, the block is again supplied by the L2 cache as the L2 cache has the latest value of the block.

Finally, if the block is ever evicted from the L2 cache, either due to the replacement policy making room for a new block, or due to invalidation received from an external source (such as from another chip), to keep the inclusion property between the L2 cache and L1 caches, invalidation is sent to all L1 caches.

To implement a crossbar needed in the shared cache configuration shown in Figure 14.1, multiple buses can be used. We can use one address bus per core that connects to all banks, one data bus per core that connects to all banks to send a block value from a core's L1 cache to the L2 cache, and one data bus for each L2 cache bank that connects to all cores' L1 caches to send a block value from the L2 cache to an L1 cache. Hence, the total number of buses scales with the sum of the number of cores and the number of L2 cache banks. In addition, the length of each bus also scales with the number of cores or number of L2 cache banks. Hence, the complexity of a crossbar scales quadratically with the number of cores and L2 cache banks.

Most of the wires in a crossbar's buses can be implemented in the metal layers of the chip, possibly on top of memory arrays. Thus, some components of the crossbar do not necessarily take much die area overhead. However, some components must still be implemented on the silicon. For example, each bus has an arbiter, a queue, and possibly repeaters. These components are implemented in silicon, and they take up a larger die area when the buses become longer Thus, a unique characteristic of a multicore architecture is that the cores, caches, and interconnection compete for the same die area (for quantification of this aspect, readers are referred to [Kumar et al., 2005b]). Scaling the crossbar to a large number of cores will essentially reduce the die area available to implement the cores themselves and the cache.

In addition the complexity of providing a crossbar interconnection, a large shared cache suffers from a high access latency, due to the physical length of wires that form the cache structure (e.g. word lines, bit lines). Even for a single

processor system, it has been demonstrated that the performance of applications improves when the cache is broken into smaller banks, allowing the applications to access cache banks that are close by with a low access latency; only accesses to far-away cache banks incur a high access latency [Kim et al., 2002].

Overall, the high-access latency of a large shared cache and the complexity of interconnecting a large number of cores and a shared cache put a limitation on the scalability of a physically shared cache. This leads to an alternative physical organization of private caches.

14.1.2 Physically Private Caches

In light of the inherent non-scalability of a physically shared cache, future multicore architecture will likely not rely on a physically shared L2 cache architecture. As we discuss in Chapter 13, a convenient and logical way to scale multiprocessor system is to move the interconnection down the memory hierarchy. In this case, multicore architectures will likely implement physically private L2 caches that are connected together using an interconnection network, or backed up by a shared L3 cache.

The interconnection of choice depends on the number of cores that are implemented on a single chip. Similar to the scalability issues facing traditional multichip multiprocessors, some medium-scale interconnection network such as rings can be used when the number of cores is still relatively small but is larger than what can be supported using a bus. A larger scale multicore system is likely to use some low-dimensional networks, or high-dimensional networks if the number of cores continues to grow. The configuration in which a multicore system uses physically private caches and use point-to-point interconnection is often referred to tiled multicore architecture because of its regular tile-like layout on the die. Figure 14.2 shows examples of a tile-based multicore architecture with ring and 2-D mesh interconnection. The "R" box in the picture denotes a router. From this point on, we will refer to a section of the L2 cache that is associated with a core as a **cache tile**.

14.1.3 Hybrid Private+Shared Caches

There are possible physical cache organizations that fall in between private and shared cache configurations. For example, we may have a group of cores sharing a cache or having private caches at the L2 level but the L3 cache is shared (Figure 14.3). Backing up private L2 caches with a shared L3 cache moves the

3 wants to write to the block, the request is sent to the L2 cache, which in turn sends invalidation to Core 1's L1 cache. Core 3 can then proceed with its write, while the block is invalidated at the L1 cache of Core 1. The newly written block value is also propagated down to the L2 cache. Later, when Core 1 wants to read the block, it suffers an L1 cache miss, and the block is supplied by the L2 cache. A read by Core 3, on the other hand, is a cache hit. When Core 2 wants to read the block, the block is again supplied by the L2 cache as the L2 cache has the latest value of the block.

Finally, if the block is ever evicted from the L2 cache, either due to the replacement policy making room for a new block, or due to invalidation received from an external source (such as from another chip), to keep the inclusion property between the L2 cache and L1 caches, invalidation is sent to all L1 caches.

To implement a crossbar needed in the shared cache configuration shown in Figure 14.1, multiple buses can be used. We can use one address bus per core that connects to all banks, one data bus per core that connects to all banks to send a block value from a core's L1 cache to the L2 cache, and one data bus for each L2 cache bank that connects to all cores' L1 caches to send a block value from the L2 cache to an L1 cache. Hence, the total number of buses scales with the sum of the number of cores and the number of L2 cache banks. In addition, the length of each bus also scales with the number of cores or number of L2 cache banks. Hence, the complexity of a crossbar scales quadratically with the number of cores and L2 cache banks.

Most of the wires in a crossbar's buses can be implemented in the metal layers of the chip, possibly on top of memory arrays. Thus, some components of the crossbar do not necessarily take much die area overhead. However, some components must still be implemented on the silicon. For example, each bus has an arbiter, a queue, and possibly repeaters. These components are implemented in silicon, and they take up a larger die area when the buses become longer Thus, a unique characteristic of a multicore architecture is that the cores, caches, and interconnection compete for the same die area (for quantification of this aspect, readers are referred to [Kumar et al., 2005b]). Scaling the crossbar to a large number of cores will essentially reduce the die area available to implement the cores themselves and the cache.

In addition the complexity of providing a crossbar interconnection, a large shared cache suffers from a high access latency, due to the physical length of wires that form the cache structure (e.g. word lines, bit lines). Even for a single

processor system, it has been demonstrated that the performance of applications improves when the cache is broken into smaller banks, allowing the applications to access cache banks that are close by with a low access latency; only accesses to far-away cache banks incur a high access latency [Kim et al., 2002].

Overall, the high-access latency of a large shared cache and the complexity of interconnecting a large number of cores and a shared cache put a limitation on the scalability of a physically shared cache. This leads to an alternative physical organization of private caches.

14.1.2 Physically Private Caches

In light of the inherent non-scalability of a physically shared cache, future multicore architecture will likely not rely on a physically shared L2 cache architecture. As we discuss in Chapter 13, a convenient and logical way to scale multiprocessor system is to move the interconnection down the memory hierarchy. In this case, multicore architectures will likely implement physically private L2 caches that are connected together using an interconnection network, or backed up by a shared L3 cache.

The interconnection of choice depends on the number of cores that are implemented on a single chip. Similar to the scalability issues facing traditional multichip multiprocessors, some medium-scale interconnection network such as rings can be used when the number of cores is still relatively small but is larger than what can be supported using a bus. A larger scale multicore system is likely to use some low-dimensional networks, or high-dimensional networks if the number of cores continues to grow. The configuration in which a multicore system uses physically private caches and use point-to-point interconnection is often referred to tiled multicore architecture because of its regular tile-like layout on the die. Figure 14.2 shows examples of a tile-based multicore architecture with ring and 2-D mesh interconnection. The "R" box in the picture denotes a router. From this point on, we will refer to a section of the L2 cache that is associated with a core as a **cache tile**.

14.1.3 Hybrid Private+Shared Caches

There are possible physical cache organizations that fall in between private and shared cache configurations. For example, we may have a group of cores sharing a cache or having private caches at the L2 level but the L3 cache is shared (Figure 14.3). Backing up private L2 caches with a shared L3 cache moves the

Figure 14.2: Examples of a tiled multicore with a ring (top) and a 2D mesh (bottom) interconnection.

complexity of the interconnection down to the L3 cache that is less frequently accessed than the L2 cache. However, a single large cache will be uniformly slow for all cores to access. In addition, the shared L3 cache must now handle cache coherence across L2 caches, so it is relatively complex to design. The L3 cache will act to sequence all requests from the L2 caches and broadcast intervention or invalidation requests to other L2 caches. The L3 cache must also be kept coherent with other L3 caches in other chips, unless it is a memory-side cache, in which it is only allowed to store blocks from the local memory node, and the coherence is

kept between L2 caches in a chip and L2 caches of other chips. Overall, a shared large L3 cache is a reasonable compromise design, but it still suffers from the scalability and complexity drawbacks of a shared cache design.

Figure 14.3: Hybrid physical configuration of L2 caches (top) and private L2 caches backed up by a shared L3 cache (bottom).

An alternative design will be the hybrid private+shared physical cache shown in Figure 14.3. Such an organization is attractive for several reasons. First, physical sharing is limited to only a few cores, hence the design of each shared cache is relatively simple. In addition, since there are fewer caches, fewer routers and links are needed to interconnect them, allowing data to be sent over the interconnection with a smaller latency. However, there is a limit to this approach. As we include more cores to share an L2 cache, the L2 cache becomes too large to offer small access latencies to the cores that share it. Eventually, if the L2 cache is too

large, the overall performance is degraded. The reason for this is shown in Figure 14.4. The figure plots the overall cache access latency as a function of cache size. The x-axis starts with a very small shared cache on the left, thus many of such shared caches are needed for a given number of cores on a chip. As we go to the right along the x-axis, the shared cache size increases, thus each cache is shared by more cores, and fewer of them are needed to achieve the same aggregate cache capacity. There are three curves in the figure. One curve, labeled *access latency*, shows the time to access one shared cache. As the cache becomes bigger, the access time increases due to the large structure of the cache. Another curve shows *routing latency*, representing the time to route request and response on the interconnection network. As each shared cache becomes larger, fewer of them are needed for the same aggregate capacity, which reduces the number of routers in the interconnection network. As the number of routers is reduced, the average distance from a random core to a random shared cache, in terms of the number of network hops, declines as well, which reduces the total routing latency. The total cache access latency is the addition of each cache's access latency and the routing latency. The figure shows that the total cache access latency declines then increases as the size of each cache increases. Thus, there is an optimum cache size beyond which the total cache access latency degrades.

Figure 14.4: Total cache access time as a function of the size of a cache tile in a tiled multicore.

To summarize, a single large cache that is physically shared by all cores is unlikely to be the organization of choice as the number of cores integrated on a chip increases in the future, due to the increasing complexity, die area overhead,

and a high access time. Physically private caches are more amenable to scaling. It is important to size the cache in such a way to optimize for the total cache latency, which includes access latency of each cache tile as well as the routing latency. This may lead to a hybrid private+shared cache in which each cache is shared by several cores and the caches are kept coherent.

14.2 Logical Cache Organization

On multicore configurations in which the L2 cache is not physically shared and is sliced into tiles that are associated with different cores, several logical organization choices are possible: shared, private, or hybrid. In a logically shared configuration, the cache tiles are treated as components of a single large cache. In this case, each tile is accessible by any core, and can be used to store data for any core. Such a shared cache organization was not feasible to implement in traditional multiprocessor systems because caches are located in different chips and hence accessing a remote cache is very expensive. In a multicore architecture, on the other hand, accessing a remote (non-local) cache tile can be performed relatively quickly as all the cache tiles are located on a single die.

There are many ways to aggregate cache tiles into a logically shared cache, as illustrated in Figure 14.5. The figure shows a 4×4-tiled multicore, physically with private L2 cache tiles assigned to different cores. One possible logical cache organization is to keep them private, in line with the physical organization. With a private cache configuration, each core can only place blocks that it fetches on its own cache tile. A block may be replicated at other cache tiles if other cores access the block too, and their values are kept consistent by the cache coherence protocol.

Alternatively, although the L2 cache tiles are physically private, they can be aggregated to form a large shared L2 cache in various ways. For example, we can organize the tiles to form a 1-way tile-associative shared cache. With this configuration, given an address, a mapping function determines which tile the address maps to. This mapping function is static, so to find a particular block, a core knows exactly which tile to look up. The lookup request is routed through the interconnection network. Note that an important concept to consider is how close the data can be found relative to the core that accesses the data, which is referred to in literature as **distance locality**. The drawback of a logically shared configuration is that the distance locality is quite poor for a large multicore system.

large, the overall performance is degraded. The reason for this is shown in Figure 14.4. The figure plots the overall cache access latency as a function of cache size. The x-axis starts with a very small shared cache on the left, thus many of such shared caches are needed for a given number of cores on a chip. As we go to the right along the x-axis, the shared cache size increases, thus each cache is shared by more cores, and fewer of them are needed to achieve the same aggregate cache capacity. There are three curves in the figure. One curve, labeled *access latency*, shows the time to access one shared cache. As the cache becomes bigger, the access time increases due to the large structure of the cache. Another curve shows *routing latency*, representing the time to route request and response on the interconnection network. As each shared cache becomes larger, fewer of them are needed for the same aggregate capacity, which reduces the number of routers in the interconnection network. As the number of routers is reduced, the average distance from a random core to a random shared cache, in terms of the number of network hops, declines as well, which reduces the total routing latency. The total cache access latency is the addition of each cache's access latency and the routing latency. The figure shows that the total cache access latency declines then increases as the size of each cache increases. Thus, there is an optimum cache size beyond which the total cache access latency degrades.

Figure 14.4: Total cache access time as a function of the size of a cache tile in a tiled multicore.

To summarize, a single large cache that is physically shared by all cores is unlikely to be the organization of choice as the number of cores integrated on a chip increases in the future, due to the increasing complexity, die area overhead,

and a high access time. Physically private caches are more amenable to scaling. It is important to size the cache in such a way to optimize for the total cache latency, which includes access latency of each cache tile as well as the routing latency. This may lead to a hybrid private+shared cache in which each cache is shared by several cores and the caches are kept coherent.

14.2 Logical Cache Organization

On multicore configurations in which the L2 cache is not physically shared and is sliced into tiles that are associated with different cores, several logical organization choices are possible: shared, private, or hybrid. In a logically shared configuration, the cache tiles are treated as components of a single large cache. In this case, each tile is accessible by any core, and can be used to store data for any core. Such a shared cache organization was not feasible to implement in traditional multiprocessor systems because caches are located in different chips and hence accessing a remote cache is very expensive. In a multicore architecture, on the other hand, accessing a remote (non-local) cache tile can be performed relatively quickly as all the cache tiles are located on a single die.

There are many ways to aggregate cache tiles into a logically shared cache, as illustrated in Figure 14.5. The figure shows a 4×4-tiled multicore, physically with private L2 cache tiles assigned to different cores. One possible logical cache organization is to keep them private, in line with the physical organization. With a private cache configuration, each core can only place blocks that it fetches on its own cache tile. A block may be replicated at other cache tiles if other cores access the block too, and their values are kept consistent by the cache coherence protocol.

Alternatively, although the L2 cache tiles are physically private, they can be aggregated to form a large shared L2 cache in various ways. For example, we can organize the tiles to form a 1-way tile-associative shared cache. With this configuration, given an address, a mapping function determines which tile the address maps to. This mapping function is static, so to find a particular block, a core knows exactly which tile to look up. The lookup request is routed through the interconnection network. Note that an important concept to consider is how close the data can be found relative to the core that accesses the data, which is referred to in literature as **distance locality**. The drawback of a logically shared configuration is that the distance locality is quite poor for a large multicore system.

For example, in the figure, if the core that accesses a data block is core 0, and the data block maps to tile 15, the network distance (assuming a two-dimensional mesh) is 6 hops. Note that to get a message across, each router likely introduces at least 3-cycle delay, corresponding to the number of pipeline stages implemented in the router. The transmission time depends on the link width, which is likely 1 to 2 cycles for a cache block worth of data. Hence 6 network hops will take at least $6 \times 3 + 1 = 19$ additional cycles (assuming cut-through routing), and can be as high as $6 \times 5 + 2 = 32$ additional cycles if the routing delay is 5 cycles. Accessing a remote cache is expensive in a large multicore with a logically shared cache configuration.

Figure 14.5: Several logical organization choices for physically-private cache tiles.

Another possible configuration is to organize the tiles with a higher tile associativity, as illustrated in the figure with a 2 and 4 tile-associative configurations. In this case, a block can map to any of two or four tiles. For example, a block with physical address 0 can map to tile 0 or tile 1, with a 2 tile-associative configuration, or can map to tile 0, 1, 4, or 5 with a 4 tile-associative configuration.

One advantage of a tile-associative configuration is an extension of the set-associativity each tile has, which tends to reduce the number of conflict misses. However, if the number of set associativities in each tile is already high, then the benefit in terms of the reduction in the number of conflict misses is likely small. Another potential benefit is that since a block can be placed in any of the two tiles in a tile set, the placement policy can be tailored in such a way to allocate the block closest to the core that accesses the block. For example, if Core 0 brings a block from the main memory that can map to tile 14 or 15, the block can be placed at tile 14, which has a lower hop count (5) compared to tile 15 (hop count of 6). Similarly, with a 4 tile-associativity, the block that can map to tile 10, 11, 14, or 15, can be placed at tile 10 instead which has a hop count of 4, versus at tile 15 which has a hop count of 6.

Tile associativity can be used to improve the distance locality of a logically shared cache further with a more careful spatial arrangement of tile sets. Figure 14.6 illustrates two ways of arranging tile sets. The top part of the figure shows a regular arrangement in which two neighboring tiles form a tile set. The bottom part of the figure shows a different arrangement in which a tile set is spread out over non-neighboring tiles. For example, tile 0 and tile 10 form a tile set, tile 1 and tile 15 form another tile set, tile 2 and tile 12 form a tile set, etc.

The benefit of spreading out a tile set is that a core has a shorter distance to reach at least one tile in any tile set. In the arrangement shown in the figure, any core can access at least one tile in any tile set with at most three network hops. Since a block that maps to a tile set can be cached in any tile in the tile set, the block can be stored in the tile that is the closest to the core that accesses the block. For example, if core 1 accesses a block that maps to a tile set consisting of tile 2 and tile 12, the block can be placed in tile 2 instead of tile 12, ensuring that core 1 can access the block in just one network hop. However, if core 14 accesses a block that maps to the same tile set, it is better to place the block in tile 12 instead, so that the distance is two network hops, instead of three network hops away if the block was stored in tile 2. The decision of which tile a block should be placed in a tile set can be made at placement time, after a cache hit, or even after several cache hits by the same core. For example, if after a block is placed in a cache tile, it is repeatedly accessed by a core that is far away, the block can be migrated to another tile in the same tile set that is closer to the core. There are definitely many other possible policies that can be used.

However, there are drawbacks to tile associativity. The higher the associa-

Figure 14.6: Arranging tile sets in order to improve distance locality.

tivity, the more tiles must be checked simultaneously to locate a data block. This consumes an additional energy, as well as incurs a higher network traffic compared to a single tile-associativity. Such drawbacks can be mitigated to some extent by requiring a core to look up the closest tile in a tile set first, and look up farther tiles only when the block is not found on the closest tile. However, an L2 cache miss will still have caused multiple lookups. In addition, even with a higher tile associativity or with spread-out tile arrangement, good distance locality is still difficult to achieve in a parallel program in which many threads access a single data item. Wherever the data item is placed will be quite far away from some of the threads. In such a case, good distance locality can only be achieved by replicating a block.

Note that it is possible to have a hybrid private and shared cache configuration, as illustrated in Figure 14.5. In the figure, each group of four tiles is logically

shared. However, each group is private, and coherence is maintained between groups. With coherence, a block can be replicated in different tile groups. However, within a group, there can only be one copy of the block.

Table 14.2 summarizes the differences between various logical cache organizations in a multicore architecture: private cache tiles, shared cache with 1-way tile-associativity (Shared-1), shared cache with A-way tile-associativity where $A > 1$ (Shared-A), and hybrid of shared within group and private among groups. In terms of how close a core is to data (distance locality), the private cache configuration is the best as data is always located in the local cache tile, assuming that the working set fits in the local cache tile. The Shared-1 configuration is the worst in terms of distance locality as data is distributed across all cache tiles. If there are n tiles, only $\frac{1}{n}$ of data will be located in the local cache tile, while the majority ($\frac{n-1}{n}$) data is located in other (remote) cache tiles. With Shared-A configuration, data can be placed in a closer tile among tiles that form a tile-set, hence distance locality improves. Distance locality can be further improved by a careful spatial arrangement of tile sets. Distance locality is even better in a hybrid organization, since a block can always be replicated in a tile of a tile-group in which a core belongs to.

In terms of the total cache capacity, the private configuration has the lowest capacity because in the worst case, a block that is accessed by all cores may be replicated in all cache tiles. Hence, a single data block may occupy as many cache lines as the number of tiles, reducing cache capacity available to store other data blocks. Limiting replication increases the aggregate cache capacity. Shared-1 and Shared-A do not allow replication at all, hence have the largest cache capacity. Private+Shared limits replication to the number of groups, hence its cache capacity falls in between the private and shared configurations.

Table 14.2: Comparing logical cache organization choices in a multicore. * indicates that the magnitude can be tuned.

Aspect	Private	Shared-1	Shared-A	Private+Shared
Distance locality	Best	Worst	Bad*	Good*
Total cache capacity	Lowest	Largest	Largest	Medium
Capacity sharing	No	Yes	Yes	Within a group

Another aspect to consider is the flexibility in sharing cache capacity beyond

a tile. The private configuration does not allow capacity sharing between cores, hence it suffers from cache capacity fragmentation. If an application running on a core has a working set that exceeds the local cache tile size, the application will suffer from a high number of capacity misses. At the same time, there may be excess cache capacity at other tiles for various reasons, for example there may not be any threads running there, or there are threads running there but they have a small working set. Such excess cache capacity cannot be donated to other cores in the private cache configuration. Shared-1 and Shared-A allow a full capacity sharing of all cache tiles between all cores. On the other hand, Private+Shared allows capacity sharing between cores that are in the same tile group, but not between cache tiles in different tile groups.

For a small tiled multicore having 4-16 cores, the poor distance locality of a logical shared cache configuration may still be tolerable. However, for a larger multicore with, say 64 cores or more, shared cache configuration is no longer a good design, for several reasons. First, most data will be located in remote tiles, very little of it will be found in the local tile. Secondly, having a large fraction of cache accesses going to remote cache tiles is wasteful in terms of the average cache hit latency, high network traffic, and high energy use due to the high traffic. Hence, it is likely that the logical cache configuration for a multicore system with a large number of cores is not going to be a shared cache, regardless of the number of tile associativities. However, the private cache configuration, while a feasible choice, suffers from several fundamental limitations. The lack of capacity sharing will cause cores that run applications with a large working set to suffer from a lot of capacity misses. As we will discuss in Chapter 16, the off-chip bandwidth in future multicore architectures is a very precious resource that must be conserved as much as possible. Considering that at any given time there will be capacity crunch in some cache tiles but excess capacity in other tiles, the lack of capacity sharing is a serious drawback. In addition, keeping a large number of cache tiles coherent requires expensive coherence protocol overheads, both storage-wise and latency-wise. Therefore, a hybrid private and shared configuration is likely to emerge as a relatively attractive alternative compared to fully private or fully shared configurations. Some capacity sharing is allowed within a group, removing the much of the capacity fragmentation problem. At the same time, if a group is relatively small, e.g. 4 tiles, the distance locality will still be good, as data is never far away from any core in a group, and can be found at up to two network hops away.

14.2.1 Mapping Function

For a logical shared cache configuration, the purpose of the mapping function is to map a block address to a cache tile, or to one cache tile in a tile set. The mapping function needs to be simple and easy to generate, and at the same time should be absence of uneven distribution of blocks into tiles. Readers may recall that a similar mapping requirement is used for deciding where block address maps to memory in a Distributed Shared Memory architecture. In that mapping function, we discussed block interleaving and page interleaving. However, in the case of tiled multicore with a shared L2 cache, both block and page interleaving should not be used, due to some pathological cases in which some sets are utilized but others are not. Figure 14.7 illustrates an example.

Figure 14.7: Bad tile mapping function due to block interleaving (a) and page interleaving (b), an improved mapping function (c), and an improved mapping function with bit randomization (d).

Figure 14.7(a) shows that with block interleaving, four bits in the address are used to index a tile in the 16-core system. Unfortunately, the same bits are also

used for indexing a set in a tile. Suppose four bits for indexing a cache tile have a value of "0100". In the tile itself, the same bits are used to index a set, hence only a certain group of sets are indexable with an index value containing "0100". Other sets are not indexable at all. Similarly, page interleaving may produce a pathological mapping as well (Figure 14.7(b)), depending on the number of sets on a cache tile versus the size of a page. For example, if a cache tile has a 512KB size, 8-way associativity, and 64B block size, and a page has 4KB size, then the set index contains $\frac{512KB}{8\times64} = 10$ bits beyond the block offset bits (i.e., bit 6 to bit 15), whereas the tile index contains 4 bits beyond the page offset bits (i.e., bit 12 to bit 15). Hence, they overlap by three bits.

Figure 14.7(c) illustrates the selection of higher order bits from the address that avoids the overlap problem. However, it is well known that higher order bits do not vary as much as lower order bits because a single value in higher order bits corresponds to a large address region. Hence, it is possible that there are only a few distinct values used by a program if we use higher order bits as the tile index, which implies non-uniform use of tiles. Thus, it may be beneficial to spread them out further by XOR-ing the higher order bits with lower order bits in order to produce more combination of tile index values (Figure 14.7(d)).

■ *Did you know?*

Note that if the address-to-tile mapping function is exposed to the operating system (OS), the OS may be able to improve distance locality by allocating pages that are used by a core to page frames that map to the tiles that are near to the core. Such an approach relies on page coloring in which page frames are assigned colors where each color corresponds all page frames that map to a particular tile. The OS page allocator can be modified to allocate a new page on a page frame that has the same color as the tile from which the page fault occurs [Cho and Jin, 2006]. This approach can significantly improve the distance locality of a logically shared cache. It can also be used to manage the cache capacity assigned to a core, by spreading data across several neighboring cache tiles. One drawback to this approach is that if for some reason a thread migrates to a different tile, the distance locality degrades, unless all pages of the thread are moved to frames that have a different color. However, moving a page in the main memory is a very expensive operation as it involves a large amount of memory copying, page table modification, and TLB shootdown.

14.2.2 Improving Distance Locality of Shared Cache

We have discussed how shared cache configuration in a large multicore suffers from the distance locality problem. We have also discussed how a hybrid private+shared cache can improve the distance locality quite significantly. However, there are more advanced solutions targeted to improve the distance locality of a shared cache configuration in a tiled multicore. We will discuss a few of them to expose readers to some possible solutions. Obviously this is an active area of research with many ongoing studies. Our goal here is briefly discuss the main idea of a few of them.

One observation for a shared cache is that distance locality is the outcome of not being able to replicate a block. Hence, if a block maps to a cache tile that is far away from a core, the core must retrieve the block from that faraway tile every time it misses on the block. If a core is allowed to replicate the block in its local cache tile, then distance locality is greatly improved. While replication is automatic in a private cache configuration as it is allowed by the cache coherence protocol, allowing replication in a shared cache requires us to support a new mechanism. This mechanism can come in different forms.

One example mechanism to replicate data in a shared cache in order to improve distance locality is *victim replication* (VR) [Zhang and Asanovic, 2005]. In VR, the local L2 cache tile is allowed to act as the victim tile of a core's L1 cache. A block that is fetched from a remote L2 cache tile is stored in the core's L1 cache. If the block is evicted from the L1 cache, it is stored in the local L2 cache as a victim. This is performed without the knowledge of the *home* tile, i.e. the tile to which a block address maps. When later the core accesses the block again, it will suffer from an L1 cache miss. At that time, it should first check if the local L2 cache keeps a replica of the block. If so, there is no need to fetch the block from the home tile. It can just retrieve the block from its local L2 cache, invalidate it from the L2 cache, and place the block in the L1 cache.

As usual, replication can introduce a data coherence problem that has to be handled correctly. For example, write propagation must be ensured when the evicted L1 cache block is dirty. In this case, the block can still be replicated at the local L2 cache, but the copy at the home tile must be updated with the new value. Write propagation due to a remote writer must also be ensured. For example, when a remote core wants to write to a block, it sends the request to the home tile, which then sends an invalidation message to all L1 caches that may have a copy of the block. When an L1 cache receives this invalidation message, it must

not just check its own tag. Since it is possible that the block is replicated at the local cache, it must also check the local L2 cache tag to invalidate the block.

There is also an issue of how to manage the use of L2 cache space between local blocks versus replica. VR uses a policy in which to make room for a new block (either real or replica), the victim block is selected from the local L2 cache based on the following priority order: (1) an invalid block, (2) a global block that is not being shared by cores, and (3) another replica.

One drawback of allowing a victim block to be replicated in the local L2 cache is that there is competition for L2 cache capacity between victim blocks and regular data blocks. Such capacity contention likely results in an increased local L2 cache capacity miss rate. Thus, VR attempts to reduce cache hit latency (by converting remote hits to local hits) at the expense of increased cache miss rate. Since a cache miss latency is usually much more expensive than a remote hit latency, VR must be able to convert many remote hits into local hits for each additional cache miss that it incurs. Due to this property, one may think of strategies to distinguish between different types of data in order to assess which data gives the most benefit from distance locality with the least drawback from increased capacity misses.

An example study that proposed selective replication *adaptive selective replication* (ASR) [Beckmann et al., 2006] found that for commercial workloads, there is a type of data that is shared by many threads but is mostly read only. Such read-only data was found to have a small capacity footprint but causes a disproportionally large number of misses. Thus, shared read-only blocks should be the most important candidate for replication, in comparison to shared read-write blocks.

Note that replication techniques for a shared cache configuration allow a shared cache to evolve to absorb some of the advantages of the private cache configuration, namely good distance locality. However, at the same time, some of the disadvantages of the private cache configuration are also absorbed. For example, replication invariably reduces the aggregate cache capacity as in the private cache configuration.

14.2.3 Capacity Sharing in the Private Cache Configuration

In the previous subsection, we discussed how an important limitation of a logically shared cache configuration, namely poor distance locality, could be addressed by selective replication. In this subsection, we will discuss ideas to address main limitations of the private cache configuration, namely cache capacity fragmentation due to the lack of capacity sharing. Obviously, this is an active area of research

with many ongoing studies. Our goal here is briefly discuss the main idea of a few of them.

The lack of capacity sharing can cause a significant system performance loss in a multicore system, especially when a diverse mix of sequential programs run on different cores. Some programs may over-utilize the local L2 cache (due to their large working set), while others may under-utilize the local L2 cache (due to their small working set). Recent trends in enterprise IT toward service-oriented computing, server consolidation, and virtualization, help create an environment in which increasingly a diverse set of applications run on a single CMP.

Such uneven cache utilization can result in a significant loss in the overall performance, due to the inability of the programs that require a large cache capacity from utilizing the excess cache capacity that may exist at the neighboring tiles. For example, with 512KB private L2 caches, several SPEC CPU2006 benchmarks (e.g. namd, milc, povray, libquantum) suffer almost no decrease in performance when half of their private L2 cache capacity is taken away, indicating a significant amount of excess cache capacity. Without capacity sharing, such excess cache capacity cannot be donated to other cores which need more cache capacity.

Recognizing the need to provide capacity sharing in private caches, prior work, such as, Cooperative Caching (CC) [Chang and Sohi, 2006] and Dynamic Spill Receive (DSR) [Qureshi, 2009], have proposed capacity sharing mechanisms. CC allows each core to spill an evicted block to any other private cache on-chip, if the block is not already replicated. While this allows capacity sharing, allowing any core to spill into other caches regardless of the temporal locality of the application running in the core may produce an unwanted effect of polluting other caches without providing much performance improvement to the application, when the application has little temporal locality. DSR attempts to remove this drawback by distinguishing between applications that can benefit from extra cache capacity beyond that provided by their local L2, versus applications that can tolerate a reduction in L2 cache capacity without much performance loss. The former applications are referred to as Spillers and the latter, as Receivers. In DSR, only cache blocks evicted by a Spiller are allowed to spill into caches belonging to Receivers, while Receivers, which cannot benefit from additional cache capacity, are not allowed to spill their blocks. In a sense, Receiver-caches have a new role of being the victim caches of Spiller-caches.

Both CC and DSR allow remote caches to be treated as the victim cache of the local cache. Is it the best policy for managing capacity sharing? Not neces-

sarily. The optimality of treating remote caches as a large victim cache rely on the premise that victim blocks are less likely to be reused by the core than blocks that are currently in the local cache. Interestingly, this premise is not always true, and is often false for Spiller applications. This is due to a behavior common in Spiller applications in which blocks that were less recently used are more likely to be accessed in the near future. In other words, Spillers often exhibit anti-LRU behavior.

Figure 14.8: Stack distance profiles for several SPEC 2006 benchmarks, showing Spillers vs. Receivers and perfect LRU vs. anti-LRU behavior.

To illustrate this anti-LRU temporal reuse pattern behavior, Figure 14.8 shows the stack distance profile for eight SPEC CPU2006 benchmark applications. The x-axis shows the cache ways sorted based on their recency of accesses, in a 2MB 32-way L2 cache, and the y-axis shows the percentage of cache accesses that hit in a given way. Thus, the leftmost point represents the percentage of accesses to the most recently used blocks. As we go to the right along the x-axis, the figure shows the percentage of accesses to less and less recently used blocks. Finally, the right-most point on the x-axis accounts for all the misses which the 32-way, 2MB cache was not able to avoid. Note that if an application has a perfect LRU temporal reuse behavior, i.e. blocks that were accessed more recently have a higher likelihood of being accessed again, the stack distance profile will show monotonically decreasing bars.

The figure organizes applications into two rows. The bottom row shows applications with perfect LRU temporal reuse behavior (monotonically decreasing bars), while the top row shows applications with imperfect (or anti-) LRU tem-

poral reuse behavior (bars may rise after they decline). The extent of anti-LRU temporal reuse behavior varies across applications. For example, omnetpp has less than 1% hits in the first 8 ways, but 57% hits in the next 24 ways. Other applications in the top row show a mixture of LRU and anti-LRU behavior, generally with LRU behavior in the first few ways, but anti-LRU in latter ways.

In addition, the first five applications in the figure (xalancbmk, omnetpp, soplex, hmmer and bzip2) have a significant number of hits beyond the 8th (and even the 16th) way. These applications can benefit from extra cache capacity beyond a 512KB (and even beyond a 1MB) private L2 cache. Hence, we can categorize such applications as Spillers. On the other hand, as shown, other applications do not depict this behavior - namd, milc and povray indicate negligible hits beyond the 3rd, 2nd and 5th ways, respectively. These applications do not gain performance with more cache space, and may even donate some large portions of their cache space with almost no performance degradation. We can categorize such applications as Receivers. Notice that four of five Spillers shown in the figure exhibit anti-LRU behavior, while all three Receivers exhibit perfect LRU behavior. Therefore, in general, Spillers usually exhibit some anti-LRU temporal locality behavior, while Receivers usually exhibit perfect LRU temporal locality behavior.

Note that if each core has a 512KB private L2 cache that is used to store all new incoming blocks (as in DSR), then the first 8 most recently used ways in the stack distance profile will correspond to the local 512KB cache. Because DSR uses remote caches to store victim blocks evicted from the local L2 cache, remote L2 caches strictly hold blocks that are farther away in the recency order than the local L2 cache. This implies that the 24 ways beyond the first 8 in the figure correspond to the total size of three remote L2 caches used to store victim blocks of the local L2 cache (assuming a 4-core CMP). Since DSR treats remote caches as a victim cache, it is guaranteed that blocks in the last 24 ways are going to be located in the remote L2 caches. This creates a situation in which, for Spillers, there is going to be a large number of remote hits.

A large number of remote L2 cache hits is costly from both performance and an energy point of view. A remote hit has a significantly longer latency compared to a local hit due to the coherence action needed for the remote cache to source the block into the local cache. In addition, it also incurs more energy consumption due to the coherence request posted on the interconnection, snooping and tag checking activities in all remote caches, and data transfer across the interconnection. Thus, it is very important for performance and energy consumption reasons to maximize

the local hits and minimize the remote hits.

Unfortunately, with DSR, since all incoming blocks are always placed in the local L2 cache, anti-LRU temporal locality behavior common in Spillers cause these blocks not to be accessed during their lifetime in the local L2. The lines would have moved out to a remote L2 before they are accessed the second time. For example, for omnetpp, this will cause 57% remote hits versus only 1% local hits. Even when the local L2 cache size is increased to 1MB, the anti-LRU behavior will still cause a significant number of remote cache hits. Other applications are less extreme compared to omnetpp, but they also exhibit anti-LRU behavior to some extent. For example, in xalancbmk and hmmer, there is a fraction of blocks that exhibit the LRU behavior, and a fraction of blocks which exhibit anti-LRU behavior. Therefore, deciding the placement of blocks in local vs. remote caches only when a block is evicted from the local L2 cache is inherently unable to reduce the number of remote hits, especially for Spiller applications. In order to improve the number of local hits, we must *selectively* decide to place blocks locally vs. remotely when they are brought into the L2 cache from the lower level memory hierarchy. A recent study [Samih et al., 2009] investigates how such selective placement policy can be implemented in hardware with reasonable costs.

14.3 Keeping Cache Tiles Coherent

As with traditional multichip multiprocessors, the choice of cache coherence protocol highly depends on the size of the multicore, i.e. the number of caches that need to be kept coherent. A small number of cores can benefit from a bus-based interconnection with broadcast coherence. However, since a bus is not scalable due to its physical constraints (clock skew, central arbitration limit, etc.), for a medium- or large-scale multicore, a different alternative is needed. For a medium-scale multicore, a more suitable alternative is to use point-to-point interconnection, but still with a broadcast/snoopy protocol. Eventually, however, broadcast/snoopy protocol cannot scale to a large-scale multicore. For large-scale multicore systems, the only choice to keep cache tiles coherent is to use a directory protocol; it is the only scalable protocol for a large number of caches. However, as in multichip systems, a directory protocol has its own drawbacks. One drawback is that a directory protocol involves request indirection, in which a request always has to inquire the directory at home before it can locate which caches have a copy of the block. Secondly, directory states incur space overheads. We will discuss

both broadcast/snoopy and directory protocols in more details.

14.3.1 Broadcast/Snooping Coherence Protocol on Multicore

Recent 8-core systems, such Intel Larrabee [Seiler et al., 2008] use point-to-point interconnection instead of a bus, to keep eight L2 cache tiles coherent. The interconnection of choice is a bidirectional ring. Each link in the ring is 512-bit wide in each direction, hence it only takes one clock cycle to send one block of data over a link. A nice property of a ring is that each router only has a degree of two. A packet received in one direction either ends at the receiving router, or is forwarded in the same direction. Since there is a 1-to-1 mapping between an input and and output port, there is no need to implement an arbitration for competing input channels, or for packets that collide at the same output channel. Hence, the router can be greatly simplified with a relatively small routing delay. However, a ring interconnection network cannot scale to a high number of caches since the diameter and average latency scales proportionally to the number of caches. To reduce the diameter of the network, an intuitive alternative is a higher dimensional interconnection network, such as a mesh.

This section will discuss the coherence protocol mechanism for medium-scale multicore architecture in which a broadcast/snoopy protocol is implemented over point-to-point interconnected cache tiles.

To recap from the discussion of coherence protocol in previous chapters, to implement a coherence protocol correctly, we need to provide the following components:

1. Write propagation through invalidation and maintaining cache coherence states

2. Request serialization through (a) a global sequencing of all write and read coherence requests that affect a single block address, and (b) handling of races by avoiding an overlap in the processing of requests, or by giving the illusion of no overlap.

Ensuring write propagation is performed by keeping track of all sharers in a directory-based protocol, or by broadcasting invalidation to all caches in a broadcast protocol. In this regard, broadcast coherence over a point-to-point network relies on the same strategy as a broadcast protocol over a shared bus.

Request serialization in a bus-based broadcast protocol is achieved by ensuring that all processors obey the order in which requests are posted on the shared

bus. For example, if a processor wants to write but snoops an invalidation request before it has a chance to place its upgrade request on the bus, the processor has to cancel its write, invalidate its cache block, and reissue the write request as a read exclusive request. In a directory-based protocol, the directory serializes the requests. Whichever request arrives at the directory first, among competing requests, wins and is serviced before other requests can be serviced. Hence, the directory acts as the **global sequencer** (ordering point).

Avoiding overlap in processing of multiple requests that are already sequenced is trivial in a shared bus as long as snooped requests are serviced based on the order in which they appear on the bus. Avoiding overlap in processing of multiple requests in a directory protocol requires the use of transient states in the directory to indicate that the current request processing is not yet completed. In addition, it requires buffering of each new request of which its processing is deferred until the current request processing is completed, or a negative acknowledgment to tell the requestor to retry the request. This requires the directory to detect the completion of request processing. Some overlap can be allowed if the outcome of the overlap can be guaranteed to be the same as without overlapping the processing. For example, this is achievable using requestor-assisted scheme (Section 11.4.2).

Let us now assess the situation with regard to achieving a broadcast protocol over point-to-point interconnection network, such as a ring or a mesh. Write propagation can be ensured by guaranteeing that invalidation will reach all tiles, and all tiles will react to it. Such guarantee is naturally provided by a broadcast protocol and through snooping.

Globally sequencing requests is more challenging due to the absence of a natural sequencer since there is no directory or a shared bus. In general, there are two alternatives to achieve global sequencing of requests. The first approach is to assign a sequencer whose role is to provide an ordering point for colliding requests and ensuring that requests are processed and seen by all cores in that order. The second approach is to design the protocol to work in a distributed fashion, without having to assign a single sequencer.

An example of the first approach (assigning a global sequencer) is the AMD Opteron [AMD, 2005] system, described in [Marty, 2007]. The system connects multiple cores and the memory controller with an interconnection network, where the memory controller acts as the global sequencer. All requests are sent to the memory controller, and the order in which they are serviced follows the same order in which they are received by the memory controller. For example, a write

request that arrives at the memory controller will be broadcast to all cores, which respond with an invalidation acknowledgment to the memory controller. A read request that arrives at the memory controller will also be broadcast to all cores, since there is no directory to tell the memory controller where the block may be cached. If a cache holds the block in a dirty state, it will supply the block. When the requestor obtains the block, it sends acknowledgment to the memory controller, which then considers the request processing complete. During the time request processing is still outstanding, new requests that arrive at the memory controller will be buffered and serviced at a later time when the current request is complete. Note that this approach locks up the memory controller until the current request is fully processed. As discussed in Chapter 11 in Section 11.4.2, such an approach reduces concurrency for request processing. However, for a small number of caches, this is not a serious problem.

In a more general design, the global sequencer for competing requests does not have to be centralized at the memory controller. It can be distributed across tiles. For example, we can assign each address a home tile, and the home tile of an address acts as the sequencer. Each home tile acts as the sequencer for a subset of all addresses that map to it. This is similar to the role performed by the directory in a directory protocol, however without the home keeping directory states.

Illustration of Global Sequencing

Figure 14.9 illustrates how a cache coherence protocol that relies on a global sequencer works. We do not place any assumption as to what type of device is assigned the role of the sequencer (AMD Opteron assigns that role to the memory controller). Suppose that node A and B simultaneously want to write to a block that they already cached in a clean state. Without sequencing the writes, we cannot guarantee that we achieve write serialization. Hence, suppose that node S is assigned the role of sequencing requests for the block. Hence, both node A and B send the write request to node S (part (a) of the figure). Suppose that the request from node A arrives at node S before the request from node B, and hence S needs to ensure that the write by A is seen by all cores as happening before the write by B. To achieve that, S serves the request from A, but replies to B with a negative acknowledgment so B can retry its request at a later time, or buffers the request by B and serves it when the current request processing is completed. To process the write request from A, S can broadcast an invalidation message to all nodes (b). All nodes receive the invalidation message, invalidate the copy of the block

if the block is found in the local cache, and send invalidation acknowledgment to node A (c). Node A waits for these invalidation acknowledgment messages until all of them are collected. At that time, A knows it is safe to write to the block, and it sends a notice of completion to the sequencer S (d). Upon receiving the notice, the sequencer knows that the write by A is completed, so now it can serve the next request.

Figure 14.9: Illustration of a broadcast coherence protocol over point-to-point interconnection with a global sequencer.

Figure 14.10 illustrates another case of how the global sequencer works, this time with a read request. Suppose that node A and B simultaneously want to read to a block that they do not find in their local caches. Both node A and B send the read request to node S (part (a) of the figure). However, the request from node A arrives earlier in node S, and hence it is served first by S. Since S does not keep a directory, it has no idea whether the block is cached somewhere in a dirty state, or it is not cached anywhere. In this case, S needs to be conservative and assumes that there is a dirty copy of the block somewhere, and broadcasts an intervention message to all nodes (b). Each node that receives the intervention message reacts by checking its cache tag. If it finds a valid copy of the block, it sends the block to the requestor (node A). Otherwise, it sends out a different type of message to node A indicating that the block is not found (c). A will collect all these response messages. If it receives data block in one of the responses, it closes out the request

by sending a message to the sequencer S (d). However, if it does not receive any response messages containing data (e), then it knows it must ask the sequencer to fetch the data from the lower level memory hierarchy (f).

Figure 14.10: Illustration of a broadcast coherence protocol over point-to-point interconnection with a global sequencer.

In the illustration, data fetch from the lower level hierarchy is not started until after all snoop responses are collected, which is very late. An alternative to this is to issue the fetch from the lower level hierarchy speculatively, as soon as the sequencer receives a read request. Hopefully, by the time the snoop responses are collected, if none of the responses contains data, the data is already fetched from the lower level hierarchy by the sequencer, which can now supply data to the requestor. If the snoop responses contain data, however, the version fetched from the lower level hierarchy may be stale and needs to be discarded. Such a strategy

hides memory latency at the expense of increased off-chip bandwidth usage and higher energy consumption.

Global Sequencing with Overlapped Request Processing

The protocol discussed in the previous section suffers from not being able to overlap request processing completely. This may cause an excessively high coherence transaction latency due to the high waiting time as requests from different nodes for a single address collide at the sequencer.

One way to allow overlapped request processing while still giving the appearance that requests are not overlapped is to use an ordered network. The trick of using an ordered network to overlap request processing has been used in some DSM system [Gharachorloo et al., 2000], and can be applied to a multicore system as well. The basic idea is to impose or use the property of the network to ensure that all nodes see the order of transactions in the same way as the sequencer. For example, we can define a deterministic path from the sequencer to every other node. Different destination nodes may share a path or have their own paths. All messages sent by the sequencer to a node must follow this pre-determined path. Hence, messages sent by the sequencer are guaranteed to arrive at the destination node in the same order as the sequencer sent them.

Figure 14.11 illustrates how this request overlapping works. Suppose node A sends a write request to S and in the mean time B sends a read request to S. Suppose the write request from A arrives at the sequencer before the request from B arrives (a). Upon receiving the write request, the sequencer sends out invalidation messages to all other nodes, while sending an acknowledgment message to the requestor node A indicating that the request has been accepted by the sequencer (b). The messages may use different paths as shown in the figure, but for each destination node, only one path is allowed. While this is still going on, the sequencer assumes that this process is already completed. When it receives the read request from node B, it forwards intervention messages to all other nodes, and sends an acknowledgment to node B indicating that the request has been accepted by the sequencer (c). Now, because the network message delivery is ordered, it is guaranteed that each node will first receive the invalidation message before it receives the intervention message. Each node will have invalidated its own copy of the block by the time it receives the intervention message (d). Thus, they would respond indicating that they do not find the block in their cache. This guarantees that it will be node A who will supply the data to B.

Figure 14.11: Overlapping request processing using an ordered network.

Note that A will receive the acknowledgment message from the sequencer before the intervention message. Hence, A knows that its request is accepted by the sequencer earlier than the read request by node B. However, node A may receive the intervention message before it has collected all invalidation acknowledgments. It will be incorrect to supply data to B until after all the invalidation acknowledgments are collected and its write completed. Therefore, similar to the requestor-assisted protocol for DSM, node A must wait until its own request processing is complete before it reacts to other requests. In the mean time, node A can buffer the intervention message or negative acknowledge it until it has collected all invalidation acknowledgments and has performed the write to the data. After that occurs, it can supply the newly written data directly to node B (e). Similarly, node B must wait until it receives data before it can service other requests that it receives after the arrival of the acknowledgment message from the sequencer.

To summarize, we have discussed a broadcast/snoopy cache coherence protocol over point-to-point interconnection that relies on a global sequencer. We have discussed a variant of that protocol that does not overlap request processing at all, and another variant that overlaps request processing by exploiting the property of an ordered network.

■ *Did you know?*

The protocol discussed earlier can also be implemented for cores that are interconnected with a ring topology. However, they will not be very efficient because the ring only allows one path from a node to another node. Hence, a request may reach the sequencer in one rotation, while the invalidation or intervention broadcast takes another rotation. For a ring topology, there are other protocols that do not rely on a global sequencer, and a request can potentially succeed in just one rotation. More detailed discussion on such protocols can be found in [Marty, 2007]. A ring can also be logically constructed on top of other interconnection topologies, such as the mesh, and a coherence protocol constructed on such "embedded ring" has a mixture of characteristics of a ring protocol and a more general broadcast protocol [Strauss et al., 2007].

14.3.2 Directory-Based Coherence Protocol

Although point-to-point interconnection networks can easily handle a large number of cores, the broadcast/snoopy protocol itself cannot easily scale due to the high traffic incurred by broadcasting each request to all cores. As we have covered in Chapter 11, usually a block is shared by only a few caches. This presents an opportunity to reduce the traffic requirement, if we keep track of which caches may store a copy of a block. Tracking such information requires the use of the directory. As with DSM systems, a directory protocol can also be implemented on a multicore system in a similar fashion.

Just like in a DSM system, each block address is assigned a home. In the case of a tiled multicore, a home is not a memory node, but a tile. We can add directory information to each tile for all addresses that map to the tile. A major difference between DSM and directory-based multicore is that in DSM, the block is permanently stored in the home's memory. When there is a request for the block sent to the home, if the block is clean, it can be fetched directly from the home node. In the case of directory-based multicore, the home tile does not actually

have the data, as it only keeps the directory information for the block. Thus, after looking up the directory, if the block is found to be clean, and no cached copy exists, the home tile must fetch the block from the off-chip memory. If, on the other hand, a cached copy exists, the home may attempt to ask the tile that has a copy to supply the block to the requestor, which can supply the block with a smaller latency than fetching the block from off-chip memory.

Another important consideration in a directory-based multicore is the distance locality issue. Data may be cached in a nearby tile but the directory may be located in a faraway tile. The lack of distance locality is due to address-to-directory mapping that does not consider the distance between a core that uses a data block and the tile in which the directory information for the data resides. Thus, accesses to the directory in the remote tile incurs a high delay and high on-chip network traffic. This problem is illustrated in Figure 14.12. Suppose that currently, a data block is cached in tile P0, and its directory information is kept in tile P15. When P4 wants to read block X, it has to inquire the directory at a faraway tile P15 (circle 1). P15 discovers that the block is currently cached at tile P0, so it forwards an intervention request to P0 (circle 2), and P0 responds by sending a copy of block X to P4 (circle 3). Thus, although data is close by, with the directory being far away, distance locality suffers.

Figure 14.12: The lack of distance locality in a directory coherence protocol in a tiled multicore architecture.

In order to improve distance locality for a directory protocol, several approaches are possible. One approach is to use a hierarchical directory protocol, such as one referred to as the Virtual Hierarchy (VH) [Marty and Hill, 2007]. In the VH protocol, multiple tiles that are spatially close to one another are put together to form a coherence group. Each coherence group employs a (first-level) directory protocol to keep private caches coherent. Caches among groups are also kept coherent using a second-level directory protocol. Membership of a group can be changed at configuration time. In order to assign home tile to an address in a group, a mapping function needs to be used. To provide maximum flexibility, VH uses a configuration table in each tile. Address bits are selected to index the table as input. Each entry in the table points to the ID of the tile that acts as a home for that address.

In VH, on a cache miss, the request is sent to the home tile in the group, using the mapping function defined in the configuration table. Since a group is (hopefully) small and consists of neighboring tiles, the home is not far away from the requestor. The home checks whether the block can be supplied locally, i.e. whether there is a dirty copy of the block in one of the tiles in a group. Optionally, it can also check whether there is a clean copy of the block in one of the tiles in a group, since obtaining the block from the group will likely takes smaller latency than obtaining it from another group, or from the off-chip memory. If it finds that the block can be supplied locally, the home tile sends a message to the local tile that has a copy of the block, which then supplies the block to the requestor. If, on the other hand, the block cannot be supplied locally, the second-level directory protocol is involved. The request is sent to the second-level directory. The second-level directory does not keep bit vector of which caches keep a copy of the block. Instead, it keeps the information of which tiles act as the home tiles for the block address in different groups. When a request is received by the second level directory, the directory forwards the request to the home tile in another group which may have the block. The home tile of another group then handles the request locally in its own group, then forwards the block to the second level directory, which then further forwards the block to the requesting group.

While VH achieves good distance locality, it is quite complex, as it has to support two levels of protocols. A simpler way to improve distance locality of the directory is to use a hybrid private+shared configuration, which was shown in Figure 14.5. Each group uses a logically shared cache configuration. In a group, blocks are not replicated, hence there is no coherence problem in a group. A

group does not have many tiles, and consists of tiles that are spatially close to each other. Hence, distance locality within a group will be quite good. We can reserve the directory protocol to keep coherence only between groups. Hence, only one level of directory protocol is needed. Furthermore, since most data sharing among threads will occur within a group, the directory protocol that keeps coherence among groups, while having a poor distance locality, is involved only infrequently.

14.4 Exercise Problems

1. **Tile mapping function**. Suppose that a tile multicore system has 16 tiles, where each tile contains a core, L1 caches, and an L2 cache bank that has a size of 256KB, and block size of 128 bytes. Suppose also that the page size is 4KB. The cache tiles are physically private but are logically shared to form a large aggregate L2 cache.

   (a) Does block or page interleaving result in a situation in which some sets in a cache bank unused?

   (b) What does the associativity of each cache bank need to be in order for page interleaving to utilize all cache sets in all cache banks?

2. **Tile set arrangement**. Suppose that a 16-tile multicore is organized in a 4×4 fashion with a 2-D mesh interconnection, and the cache tiles are physically private but logically organized as a large aggregate shared L2 cache with 4-tile associativity.

   (a) Find one arrangement of tiles that form tile set in such a way that the distance of any core to at least one tile in each tile set is never more than two network hops away.

   (b) Find one arrangement of tiles that form tile set in such a way that the distance of some core to all tiles in a tile set is more than two network hops away.

3. **Shared vs. private caches**. Suppose that an application runs on a 4×4 multicore interconnected with a 2D mesh, as illustrated in the following figure:

Each cache tile has a size of 256KB. Each cache tile has an access time of 10 clock cycles. Accessing a remote cache tile takes an additional 5 clock cycles per network hop. The application has a cache sensitivity parameter (α) of 0.5. That is, doubling the cache size it uses reduces the application's miss rate by a factor of $\sqrt{2}$. If the application uses only 256KB cache, its miss rate is 30%. A cache miss takes 300 cycles to satisfy. Compare the average acces time (taking into account both cache hits and cache misses) for the following cases:

(a) The cache tiles are logically private, hence each cache tile can only be accessed by the core on the same tile. The application runs on one tile, and can only access its private cache tile.

(b) The cache tiles are logically shared to form an aggregate size of 4MB. The application runs on tile 0 at the north west corner of the chip. (*Hint*: first, calculate the average distance of tile 0 from all tiles. Use this average distance to compute the average access time for a cache hit.)

(c) The cache tiles are logically shared to form an aggregate size of 4MB. The application runs on tile 5 near the center of the chip. (*Hint*: first, calculate the average distance of tile 5 from all tiles. Use this average distance to compute the average access time for a cache hit.)

(d) Compare the results from parts (a), (b), and (c). Make qualitative observation regarding the advantages and disadvantages of the various configurations in parts (a), (b), and (c).

4. **Spilling into neighboring caches**. Suppose that an application runs on a 4×4 multicore interconnected with a 2D mesh, just like in the previous question. However, each cache tile has a size of 512KB. Each cache tile has an access time of 10 clock cycles. The temporal reuse of the application is shown in the following table:

Stack position	Percent of accesses
Most recently used 512KB	30%
Second most recently used 512KB	50%
Miss rate beyond 1MB cache size	20%

A cache miss takes 400 cycles to satisfy. Compare the average acces time

(taking into account both cache hits and cache misses) for the following cases:

(a) The cache tiles are logically private, hence each cache tile can only be accessed by the core on the same tile. The application runs on tile 5, and can only access its own cache tile.

(b) The cache tiles are logically private. The application runs on tile 5, but it is allowed to spill its victim blocks into cache tile 6 (network distance of one hop).

(c) The cache tiles are logically private. The application runs on tile 5, but it brings all new blocks into cache tile 6. Blocks that are replaced from cache tile 6 are placed in cache tile 5. If a block to be accessed by the core is found on tile 5, it is moved into tile 6. In other words, cache tile 5 acts as the victim cache for cache tile 6.

5. **Overlapped processing**. Suppose a 4×4-multicore system uses a broadcast/snoopy protocol over a 2D mesh network, to keep logically private caches coherent. Suppose a parallel program with four threads run on tile 0, 1, 2, and 3. The caches maintain MESI states. Assume that cost of a network transaction is determined by the number of network hops involved in the transaction, and each hop has a latency of 5 cycles. Accessing each cache takes 10 cycles. Suppose that the parallel program incurs the following accesses to a single block address: r0, r1, w2, and r3, where r indicates a read request, w indicates a write request, and the number indicates the thread issuing the request. Tile 5 is assigned as the global sequencer for this particular block address. Its distance from tile 0, 1, 2, and 3, are 2, 1, 2, 3 hops, respectively. Suppose that the requests are received almost simultaneously but in the following order by the sequencer: r0, r1, w2, r3. The sequencer must ensure correctness from processing these competing requests. Assume that the sequencer takes only 10 cycles to send intervention or invalidation requests as a reaction to an incoming request, but only after it is safe to respond. Assume also that all snoopers need only 10 cycles to react and send a response message. Initially, the block is cached on tile 3. How many clock cycles are needed to finish the processing of these requests if:

(a) No overlap in request processing is allowed at all.

(b) Request processing is overlapped as much as possible assuming the network is ordered, where a message follows X-Y dimension-ordered routing paths.

Chapter 15

Performance Volatility

Contents

15.1 Empirical Observation **444**
15.2 Factors Causing Performance Volatility **456**
15.2.1 L2 Cache Space Contention 456
15.3 Dealing with Performance Volatility **458**
15.3.1 Cache Partitioning 458
15.3.2 Off-chip Bandwidth Partitioning 461
15.3.3 Interaction Between Cache and Bandwidth Partitioning 465
15.3.4 System Performance versus Performance Volatility . . 468
15.4 Exercise Problems **469**

The goal of this chapter is to discuss the impact of resource sharing by cores on a chip that occurs in multicore architectures. For example, in current multicore architectures, the last level cache is shared by all cores. The off-chip bandwidth is invariably shared by all cores, regardless of the cache organization. Such chip resource sharing introduces *performance volatility* to applications, where an application's performance highly depends on what other applications are running together with it. We will discuss various aspects of the performance volatility problem.

Performance volatility has various system-level implications. When the number of applications sharing on-chip resources becomes large enough, *thrashing* can occur in that system, a situation in which system performance (as well as individual application performance) degrades significantly because the system spends a disproportionate amount of time just accessing the shared resources. In addition,

system performance is likely to be lower than the maximum performance that the system can deliver, typically because the performance of some applications is reduced by a larger amount than the increase in performance of other applications. Furthermore, in the past, OS relied on assigning time quanta that are determined based on the priority level of applications, relying on a premise that the size of time quantum correlates linearly with the throughput of an application. This may no longer be the case in a multicore system, as on-chip cache capacity has become a dominant factor in determining performance. This may lead to the appearance of *priority inversion*, a situation in which a lower priority thread, when scheduled together with a higher priority thread, appears to make faster execution progress than the higher priority thread. Finally, in the extreme, some threads may appear *starved* if they cannot compete to get enough cache capacity to make meaningful progress.

Section 15.1 will discuss empirical results showing the extent of the performance volatility problem on a relatively recent multicore platform. While there are many shared resources that can produce performance volatility (as discussed in Chapter 13), we focus on resource sharing that arises specifically due to the integration of multiple processor cores on a single chip, i.e. the on-chip cache capacity and off-chip bandwidth that are shared by multiple cores. Section 15.2 discusses what factors or application characteristics cause contention-related performance volatility. Finally, Section 15.3 discusses potential solutions that can be applied to reduce the performance volatility problem.

15.1 Empirical Observation

In this section, we will look at the degree of the contention problem on a recent multicore platform, a Dell Precision 690 system. We will measure the severity of the contention problem with two workloads. The first workload is a multiprogrammed combination of sequential applications. The second workload is a parallel program with multiple threads. The reason for why we use two workloads is that they respond to L2 cache sharing differently. The multiprogrammed workload consists of independent applications that have practically no data sharing, hence L2 cache capacity sharing always produces contention and degrades the performance of each application, in comparison to if the application runs alone in the system. In contrast, the parallel program workload consists of threads that may have a substantial data sharing, hence they may actually benefit from L2

cache sharing. For example, one thread may fetch data into the cache that will be needed by another thread, avoiding the cache miss that the other thread would suffer otherwise. Hence, L2 cache sharing for a parallel program may be net harmful, if the contention impact is stronger than the data sharing impact, or net beneficial, if the contention impact is weaker than the data sharing impact.

The Dell Precision 690 system has a two processor chips connected by a bus, and each processor chip is an Intel Xeon chip with two cores. Each core is a 64-bit x86 processor with a clock frequency of 3.2 GHz. Each core has private L1 instruction and data caches, and both cores on a chip share a 2MB L2 cache (Figure 15.1). Each core also has two hardware thread contexts, implementing simultaneous multi-threading (SMT) technology, which Intel refers to as hyper-threading. SMT is informally labeled as a poor man's multiprocessor because although two threads can run simultaneously, they share and compete for processor core resources such as the branch predictor, out of order pipeline, physical registers, functional units, and the L1 caches. MESI coherence protocol is implemented to keep the L2 caches coherent between the two chips.

Figure 15.1: A high level diagram of the Intel Xeon system used in the empirical evaluation.

To the Linux OS that run on the system, the cores and thread contexts appear as regular processors, so there are a total of eight processors visible to the OS (Figure 15.1). In order to test the degree of contention at different resources, we pin applications to different combinations of processor IDs.

In our experiment, we run a subset of memory intensive benchmarks from the SPEC 2006 benchmark suite [Standard Performance Evaluation Corporation, 2006].

SPEC 2006 benchmark suite consists of sequential (non-parallel) benchmarks that were chosen for testing the performance of a processor and its cache memories. It contains 12 integer-intensive benchmarks and 17 floating-point intensive benchmarks. The benchmarks were not designed to stress main memory capacity or I/O performance. Details of the benchmarks are shown in Table 15.1.

Table 15.1: List of SPEC 2006 benchmarks which include floating-point intensive benchmarks (FP) and integer-intensive benchmarks (Int).

Benchmark	Type and Domain	Language
bwaves	FP, fluid dynamics	Fortran
gamess	FP, quantum chemistry	Fortran
milc	FP, physics/quantum chromodynamics	C
zeusmp	FP, physics/CFD	Fortran
gromacs	FP, biochemistry/molecular dynamics	C, Fortran
cactusADM	FP, physics/general relativity	C, Fortran
leslie3d	FP, fluid dynamics	Fortran
namd	FP, biology/molecular dynamics	C++
dealll	FP, finite element analysis	C++
soplex	FP, linear programming, optimization	C++
povray	FP, image ray-tracing	C++
calculix	FP, structural mechanics	C, Fortran
gemsFDTD	FP, computational electromagnetics	Fortran
tonto	FP, quantum chemistry	Fortran
lbm	FP, fluid dynamics	C
wrf	FP, weather	C, Fortran
sphinx3	FP, speech recognition	C
perlbench	Int, programming language	C
bzip2	Int, compression	C
gcc	Int, C compiler	C
mcf	Int, combinatorial optimization	C
gobmk	Int, artificial intelligence	C
hmmer	Int, search gene sequence	C
sjeng	Int, artificial intelligence, chess	C
libquantum	Int, physics/quantum computing	C
h264ref	Int, video compression	C
omnetpp	Int, discrete event simulation	C++
astar	Int, path-finding algorithms	C++
xalancbmk	Int, XML processing	C++

We run two applications at a time using the combination shown in Table 15.2. If the two applications run on two cores on different chips, they only contend for off-chip resources such as the system bus, the memory controller, and the main memory. The contention mostly manifests in longer memory access latencies due to collision between accesses that come from the two chips. If the two applications run on different cores on the same chip, there are extra sources for contention from the off-chip bandwidth and from the L2 cache capacity. The contention for accesses to off-chip bandwidth manifests in the increase of memory access time. However, the contention for L2 cache capacity manifests in the increase of the number of L2 cache misses for both applications. Finally, if the two applications run on different hardware contexts on the same core, they suffer from additional contention due to sharing the L1 caches and core resources such as the branch predictor, out of order pipeline, instruction decoders, physical registers, and functional units. The contention for L1 caches will manifest in higher L1 miss rates for both applications (increase in the number of L2 cache accesses). The contention for core resources will manifest in various ways such as increased branch misprediction rates and higher incidents of stalls due to various pipeline hazards.

Table 15.2: Where applications are run and the source of resource contention they suffer from.

Where Apps Run	Sources of contention
2 cores on different chips	system bus, memory controller, main memory
2 cores on the same chip	L2 cache capacity, off-chip bandwidth
2 hw contexts on 1 core	core resources, L1 caches

The execution time results for various combinations of two applications running together are shown in Figure 15.2. Each combination of two applications is represented by a pair of bars. The height of the bar is the total execution time of each application normalized to the case in which the application runs alone in the system. Each bar has several components added to the execution time in the "Alone" case, which is at 100%. The component of a bar immediately above 100% is the additional execution time incurred when two applications run simultaneously on two cores on different chips. In this setting, the additional execution time is caused by contention between the two threads for the system bus, memory controller, and the main memory. The next component up in a bar is the addi-

tional execution time incurred when two applications run simultaneously on two cores on the same chip. In this setting, the additional execution time is caused by contention between the two threads for L2 cache capacity and the off-chip bandwidth. Finally, the topmost component of a bar is the additional execution time incurred when two applications run simultaneously on two different hardware thread contexts on the same core. The additional execution time is due to the contention between the two threads for core resources and the L1 cache ports and cache capacity. The total height of a bar is the normalized execution time when the applications run on different thread contexts on a single core. The bar to the far right represents the average for all applications.

Figure 15.2: Execution time impact of resource contention on two sequential applications running together on separate chips, separate cores on a chip, and separate hardware contexts on a core.

There are several observations that we can make. First, the slowdown due to contention for off-chip components is relatively small across all applications,

on average producing a 2% slowdown, and in the worst case producing a 12% slowdown. The most significant factors that contribute to the slowdown are (1) the contention for core resources and the L1 caches (27% on average and 58% in the worst case), and (2) the contention for L2 cache capacity and off-chip bandwidth (13% on average and 40% in the worst case). The overall average and maximum slowdown are 42% and 79%, respectively.

Note that it would be incorrect to only look at the average and conclude that contention for core resources is a much more severe problem than the contention for L2 cache capacity and off-chip bandwidth. First, there are applications that suffer very little from the contention for core resources, but suffer significantly from L2 cache capacity and off-chip bandwidth contention. For example, soplex in soplex+milc combination suffers from a 12% slowdown from contention for core resources, but a 39% slowdown from contention for L2 cache and off-chip bandwidth. Likewise, lbm suffers from a slowdown of 5% from contention for core resources, but a 26–38% slowdown from contention for L2 cache and off-chip bandwidth, in lbm+mcf and omnetpp+lbm combinations. The reason for the variation in the magnitude of slowdown is caused by the temporal reuse pattern of both applications that run together (more discussion in Section 15.2).

In addition, the contention level for core and the L1 cache resources has reached its maximum for the Xeon system since each processor core can only run two threads simultaneously. In contrast, the L2 cache and off-chip bandwidth can be shared by four threads simultaneously, but the result only represents the case where two threads from different applications run simultaneously. Under a full load where four threads run in each chip, the slowdown due to L2 cache and off-chip bandwidth contention will increase from the level shown in the figure.

To illustrate this point, Figure 15.3 shows a case in which four benchmarks run together on the hardware thread contexts and cores on a single chip, and on separate cores across two chips. Each bar has two components above 100%: the middle component represents the additional execution time due to running the benchmarks on separate cores across two chips, while the top component represents the additional execution time due to running the benchmarks on all hardware thread contexts across two cores on a single chip. The figure shows that the average slowdown is now 126% when the benchmarks run on a single chip, which is much higher than the case in which only two benchmarks run on a single chip. In the case of mcf, it suffers from a 131% slowdown, which is much higher than the 79% slowdown it suffers when it runs with only one benchmark (lbm) in Fig-

ure 15.2. If future multicore architectures continue to adopt a shared L2 cache design for a higher number of cores, the contention for the L2 cache capacity will increase beyond the current contention level. In addition, the contention level for off-chip bandwidth will increase beyond the current level, regardless of whether the L2 cache is shared or not, when more cores are implemented on a single chip.

Figure 15.3: Execution time impact of resource contention on four sequential applications running together across two chips and on a single chip.

Another observation that we can make is that although overall applications are slowed down significantly when they share resources, the overall system performance still increases, even in the case when they contend for all resources. For example, when we run two applications sequentially (one after another), if we suppose each application takes T time to complete, then it will take $2T$ time to run two applications. However, when we run them together, both are slowed down by 42% or 15% if they run on the same core or separate cores, respectively. Thus, it only takes $1.42T$ or $1.15T$ to complete the execution of two applications. Thus, overall, there is a core-hour speedup of $\frac{2T}{1.42T} = 1.41\times$ in system performance when two applications run on a single core or $\frac{2T}{1.15T} = 1.74\times$ when they run on separate cores, pointing to the benefit of multicore and SMT for improving system performance. For the four simultaneous application case, the core-hour speedup is $\frac{4T}{2.26T} = 1.76\times$. If the number of cores on a chip is increased faster than the what

can be supported by a shared cache and the off-chip bandwidth, at some point we may reach a turning point in which running more threads will cause thrashing, which degrades the overall system performance.

Finally, an application may suffer from significant performance volatility, depending on which other application it runs with. Figure 15.4 shows the normalized execution time for two such applications, astar and lbm, when each runs together with various other applications on different cores, also including when they run alone.

Figure 15.4: Performance volatility of astar and lbm when running with another application.

The figure shows that astar's execution time is not slowed down much when it runs with gcc, but is slowed down by 72% when it runs with lbm. It also shows lbm's execution time is only slowed down by 11% when it runs with astar, but is slowed down by 46% when it runs with omnetpp. Such performance volatility is significant, and it is very likely to increase if there are more applications running in the system, and when there are more cores on a chip. Obviously, a performance volatility of this magnitude cannot be left unmanaged. In Section 15.2 and Section 15.3, we will discuss what factors cause different applications to suffer from different degrees of performance volatility, and discuss potential solutions

for the performance volatility problem. Before that, however, we will look at the contention problem for threads from parallel programs running on different cores.

Figure 15.5 shows the execution time of PARSEC parallel benchmarks with two parallel threads (top) and with four parallel threads (bottom). PARSEC 2.0, which stands for Princeton Application Repository for Shared-memory Computers, is a collection of shared memory parallel benchmarks released in 2009 that consists of primarily emerging non-scientific computing applications (listed in Table 15.3).

Table 15.3: List of benchmarks in PARSEC version 2.0 suite, showing the application domain and the dominant parallelization method.

Benchmark	Application Domain	Parallelization
Blackscholes	financial analysis	data parallel
Bodytrack	computer vision	pipeline
Canneal	engineering	data parallel
Dedup	enterprise storage	pipeline
Facesim	animation	data parallel
Ferret	similarity search	pipeline
Fluidanimate	animation	data parallel
Freqmine	data mining	data parallel
Raytrace	visualization	data parallel
Streamcluster	data mining	data parallel
Swaptions	financial analysis	data parallel
Vips	media processing	data parallel
x264	media processing	pipeline

All the bars in the figure are normalized to the case in which only a single thread runs. For the top figure, the ideal speedup from running with two threads should put the execution time at half of the single-threaded case. There are four different bars: the first one representing the single-thread execution time, the next one to the right representing the case in which the threads run on cores on different chips, the next one representing the case in which the threads run on different cores

but on the same chip, and the last bar representing the case in which the threads run on different hardware thread contexts on a single core.

Figure 15.5: Normalized execution time of PARSEC benchmarks for two threads (top) and four threads (bottom).

The figure shows that there is virtually no difference in the execution time between whether the two threads run on two cores on different chips or on the same chip. This is in contrast to the observation we made with SPEC 2006 applications, where each application is sequential and there is no data sharing between applications. Since PARSEC applications are parallel, the two threads have some data sharing. Running the two threads on the same chip allow them to share the L2 cache, eliminating the cache coherence overheads that would be incurred if they run on different chips. On the other hand, running the threads on separate chips

means that each thread can use a full 2MB L2 cache on the chips, whereas running them on the same chip means that they have to compete for the capacity of a single 2MB L2 cache. A possible explanation for why it does not seem to matter whether the two threads run on a single chip or across two chips is that the 2MB L2 cache may be large enough for the working sets of both threads. Hence the threads do not suffer from any meaningful capacity contention, and the off-chip bandwidth is also plentiful enough that they do not suffer from any meaningful contention. Another possible explanation is that these two effects are approximately equal in magnitude and neutralize each other. In other words, running the two parallel threads on the same chip, while producing a higher degree of contention for the L2 cache capacity and off-chip bandwidth, also eliminates cache coherence overheads, which offsets the performance impact of contention. Considering PARSEC benchmarks are relatively well optimized in terms of minimizing data sharing between threads (especially ones that rely on data parallelism), the first explanation probably carries more weight than the second explanation.

However, when the two threads run on different hardware thread contexts on the same core, they are slowed down significantly, by 49% on average over the cases in which the threads run on separate cores (either on the same chip or different chips). The average speedup ratio over the single-thread execution time is only $1.2\times$. The magnitude of slowdown is somewhat larger than what SPEC 2006 applications experience when they share the same core. This is likely caused by the harmful impact of two threads sharing core resources outweighing the beneficial impact that comes from sharing core resources. When two threads of a single parallel program share core resources, they may produce a beneficial effect in resources that are used as predictors. For example, the branch predictor trained by one thread may reduce the branch misprediction rate of another thread, since threads tend to execute similar computation. However, many parallel programs are loop intensive, and have relatively few hard-to-predict branches. Hence, even without the cross-thread branch prediction training, the branch prediction accuracy is probably already very high, leaving little room for improvement.

On the other hand, two threads sharing core resources likely suffer from structural hazards due to contending for limited core resources. Two threads from a single parallel program have a lot in common with one another in terms of their instruction mixes. Different SPEC 2006 applications typically have different instruction mixes; hence they do not always stress the same type of core resources. For example, a floating-point intensive application uses floating-point units and

registers heavily, while an integer-intensive application uses integer arithmetic and logic units and integer registers heavily. When they run together on the same core, they may not compete for the same core resources. However, a PARSEC application's threads likely exhibit similar instruction mixes and compete for the same core resources.

For the bottom figure, each application runs with four threads, hence the ideal speedup put the execution time at a quarter (0.25) of that of the single-threaded case. There are three bars for each benchmark: one representing sequential execution of the benchmark using a single thread, one representing the case in which the threads run on different cores (two threads on two cores on a chip and two other threads on two cores on another chip), and the next one representing the case in which the threads run on a single chip (two threads on one core and two other threads on another core). As before, we can expect the single-chip case to suffer from more contention but avoid cache coherence protocol overheads. The figure shows that the execution time is significantly faster when the threads run across two chips. This indicates that with four threads, the performance drawback due to L2 cache capacity and off-chip bandwidth contention can no longer be offset by the removal of cache coherence overheads that come from running the threads on a single chip. The loss of performance due to the L2 cache capacity and off-chip bandwidth contention is very significant. On average, the execution time with a single chip is 43% longer than that with two chips, and can be as high as 62% longer (for x264). In terms of speedup ratios, the effect is more pronounced, corresponding to a reduction of average speedup ratio from 3.2 to 2.2. In essence, when we run PARSEC benchmarks on a single chip, the speedup ratio peaks at two threads, and running with four threads no longer reduces the execution time of the benchmarks, whereas when we run the benchmarks with four threads across two chips, the execution time and the speedup ratio still improve over parallel execution with two threads.

Overall, we can conclude that contention for chip resources in the Intel Xeon system poses several significant problems: (1) performance degradation for sequential applications, (2) performance volatility for an application depending on what other applications are running along with it, and (3) reduction in scalability potential for parallel applications – maximum speedup is reduced and achieved with fewer threads. It is likely that these problems will increase in severity in future multicore architectures as there will be more cores integrated on a single chip. Note that the contention for the L2 cache capacity also exacerbates the con-

tention for the off-chip bandwidth because of the increased in the number of L2 cache contention-induced cache misses. We discuss this problem in more details in Chapter 16.

15.2 Factors Causing Performance Volatility

15.2.1 L2 Cache Space Contention

The performance volatility suffered by an application is affected by the temporal reuse patterns of the application and other applications that run together with it. To illustrate the relationship between an application's temporal reuse pattern and the application's vulnerability to performance volatility due to contention for cache capacity that arises due to cache sharing, consider two threads X and Y that run on different cores and share the cache. Both threads generate accesses to the cache, as illustrated in Figure 15.6. The figure assumes a 4-way associative cache, and thread X accesses blocks A, B, and then A again, while thread Y accesses blocks U, V, V, and W. All the blocks map to a single cache set. There are many possible interleaving cases between the accesses from thread X and Y. The figure shows two possible interleaving cases. In the first case, accesses to blocks U, V, and V from thread Y occur within the accesses to blocks A, B, and A from thread X. From the LRU replacement policy behavior, we know that since there are only three distinct addresses (U, B, and V) between the first and the last access to block A, the last access to A is a cache hit. However, in the second case, accesses to blocks U, V, V, and W from thread Y occurs during the accesses to blocks A, B, and A from thread X. Therefore, there are four distinct addresses (U, B, V, and W) between the accesses to A, which is equal to the cache associativity. By the time the second access to block A occurs, block A is no longer in the LRU stack since it has been replaced from the cache, resulting in a cache miss for the last access to A.

An analytical model developed in [Chandra et al., 2005] went to great lengths to estimate the probabilities of various interleaving cases. However, the illustration alone gives us sufficient information to make several qualitative observations. First, an application that reuses its cache blocks more frequently is less vulnerable to contention-related misses. For example, in Figure 15.6, if thread X reuses block A quickly, i.e. before thread B accesses block W, thread B would have evicted its own block U when it brings block W into the cache. Secondly, an application that misses frequently will also bring more blocks into the cache at a fast rate, and

Figure 15.6: How various cases of access interleaving by two threads that share a single cache produce a different cache hit/miss outcome.

this increases the likelihood that the new blocks replace old blocks that belong to another application. This is illustrated in the difference between second and first case for thread Y in the figure. Thirdly, an application with a flat stack distance profile [1], or with a stack distance profile concentrated near the LRU position, tend to be more vulnerable because a small decline in cache space can cause a large increase in the number of cache misses. In contrast, an application with a stack distance profile concentrated near the MRU position is much less vulnerable, since only a large decrease in cache capacity causes many additional cache misses. The synthesis of these factors determines the level of performance volatility an application suffers from when it runs together with other applications. Note that miss rate on its own has little to do with how vulnerable an application is. This is because what matters more is the number of reuses and misses per unit time rather than the number of misses per number of accesses.

Unfortunately, one implication of the observations we just discussed is that reducing performance volatility for an application requires changing the application's temporal reuse pattern. This cannot be easily achieved without changing the algorithm and data structure used by the application. Hence, we need architecture mechanisms to deal with the performance volatility problem.

[1] Readers are referred to Chapter 6 for the definition and illustration of a stack distance profile

15.3 Dealing with Performance Volatility

The contention for shared cache capacity, as well as for off-chip bandwidth, has produced a large performance volatility. In this subsection, we will discuss possible solutions to mitigate this situation.

One solution is to use private caches, instead of a shared cache. A physically shared cache is not scalable due to its large structure. Hence, large-scale multicore systems will likely rely more on physically private caches, where each core is associated with its own cache bank. However, this does not fully solve the problem if the physically separate caches are allowed to be logically shared. That is, if a core is allowed to use not just its private cache but also neighboring caches, then it will compete for cache space with other applications as well.

15.3.1 Cache Partitioning

Another solution is to provide a mechanism to partition the cache such that each core is assigned a cache partition that cannot be taken away by other cores. A simple mechanism to achieve this is to modify the replacement policy so that it is aware of allocation that should be made to different cores [Suh et al., 2002]. Figure 15.7 illustrates the mechanism of such a replacement policy on a cache that has four sets and four-way associativity. Each cache block is tagged with the core that brings the block into the cache. Blocks brought into the cache by Core 0 are shown in grey, while blocks brought into the cache by Core 1 are shown in white. Each core keeps two registers, one that shows how many blocks in the cache that it brought in (current allocation), and another that shows how many blocks that it should have in the cache (target allocation).

To enforce the partitioning, on a cache miss, the replacement policy is modified. Instead of always replacing the least recently used (LRU) block, an under-allocated core is allowed to bring a new block to replace a block from an over-allocated core (the top part of the figure). In the figure, the new block that Core 0 brings in from a lower level in the memory hierarchy replaces a block that belongs to Core 1 instead of replacing the LRU block. After that, the current allocation registers are updated by incrementing Core 0's register and decrementing Core 1's register (the bottom part of the figure). Since now the current allocation matches the target allocation, from here on, each block that is brought in by a core is only allowed to replace a block that belongs to the core itself. This ensures that the partition is maintained.

Figure 15.7: Illustration of a modified LRU replacement policy that performs cache partitioning over time.

The cache partitioning mechanism just discussed has several desirable characteristics: (1) changing partition sizes is simply a matter of writing new values to the target allocation registers, (2) when the partition changes, the cache operates normally without any pause, and the partition converges to the target over time. However, there are several issues that need to be considered. First, a thread may migrate from one core to another, due to the OS thread scheduler assigning the thread to an idle core, typically after the thread was pre-empted for various reasons (interrupt, using up time quantum, blocking for I/O, etc.). When a thread migrates to another core, we want to retain the allocation registers along with the information of which blocks were brought in by the thread. For this, it is better to tag blocks not by the ID of the core that brought them into the cache, but by some

15.3 Dealing with Performance Volatility

The contention for shared cache capacity, as well as for off-chip bandwidth, has produced a large performance volatility. In this subsection, we will discuss possible solutions to mitigate this situation.

One solution is to use private caches, instead of a shared cache. A physically shared cache is not scalable due to its large structure. Hence, large-scale multicore systems will likely rely more on physically private caches, where each core is associated with its own cache bank. However, this does not fully solve the problem if the physically separate caches are allowed to be logically shared. That is, if a core is allowed to use not just its private cache but also neighboring caches, then it will compete for cache space with other applications as well.

15.3.1 Cache Partitioning

Another solution is to provide a mechanism to partition the cache such that each core is assigned a cache partition that cannot be taken away by other cores. A simple mechanism to achieve this is to modify the replacement policy so that it is aware of allocation that should be made to different cores [Suh et al., 2002]. Figure 15.7 illustrates the mechanism of such a replacement policy on a cache that has four sets and four-way associativity. Each cache block is tagged with the core that brings the block into the cache. Blocks brought into the cache by Core 0 are shown in grey, while blocks brought into the cache by Core 1 are shown in white. Each core keeps two registers, one that shows how many blocks in the cache that it brought in (current allocation), and another that shows how many blocks that it should have in the cache (target allocation).

To enforce the partitioning, on a cache miss, the replacement policy is modified. Instead of always replacing the least recently used (LRU) block, an under-allocated core is allowed to bring a new block to replace a block from an over-allocated core (the top part of the figure). In the figure, the new block that Core 0 brings in from a lower level in the memory hierarchy replaces a block that belongs to Core 1 instead of replacing the LRU block. After that, the current allocation registers are updated by incrementing Core 0's register and decrementing Core 1's register (the bottom part of the figure). Since now the current allocation matches the target allocation, from here on, each block that is brought in by a core is only allowed to replace a block that belongs to the core itself. This ensures that the partition is maintained.

Figure 15.7: Illustration of a modified LRU replacement policy that performs cache partitioning over time.

The cache partitioning mechanism just discussed has several desirable characteristics: (1) changing partition sizes is simply a matter of writing new values to the target allocation registers, (2) when the partition changes, the cache operates normally without any pause, and the partition converges to the target over time. However, there are several issues that need to be considered. First, a thread may migrate from one core to another, due to the OS thread scheduler assigning the thread to an idle core, typically after the thread was pre-empted for various reasons (interrupt, using up time quantum, blocking for I/O, etc.). When a thread migrates to another core, we want to retain the allocation registers along with the information of which blocks were brought in by the thread. For this, it is better to tag blocks not by the ID of the core that brought them into the cache, but by some

hardware id associated with the thread. When the thread migrates, the blocks' tags do not need to be changed. In addition, we should treat the target allocation register as a part of the thread's context so we can save and restore it.

Another issue is that the partitioning mechanism shown in the figure is rigid once the target partition is achieved. At that point, each set will have constant numbers of blocks that belong to different cores, regardless of whether the threads that run on them change their access patterns and no longer utilize cache sets that they used to utilize heavily. This can cause an undesirable situation when a thread that utilizes some sets heavily moves to utilize other sets, its performance suffers since the old sets have many blocks allocated to the thread but the new sets have few blocks allocated to it. Hence, the replacement policy must allow perturbation to avoid this situation, either by periodically fluctuating the target allocation slightly, or by allowing the current allocations to deviate from the target allocations slightly. Alternatively, we can assign target and current allocation registers to each set so that each set can enforce its own partitioning, at the expense of the storage and management overheads of keeping many registers.

Finally, while the cache partitioning mechanism can enforce partition sizes, the unresolved question is how to determine the partition sizes that should be assigned to different cores. Some studies have proposed to partition the caches so as to improve the overall throughput or fairness. Suh et al. proposed a cache partitioning policy that minimizes the total number of cache misses [Suh et al., 2002], Kim et al. proposed a cache partitioning technique to optimize for fairness (uniform slowdown) to applications that share the L2 cache [Kim et al., 2004], while Qureshi et al. proposed utility-based cache partitioning that attempt to optimize for system performance [Qureshi and Patt, 2006]. Hsu et al. studied the impact of various optimization goals in guiding how cache partitions are allocated in a CMP architecture [Hsu et al., 2006]. The optimization goals include maximizing the overall performance metric, such as the instructions per cycle (IPC) or miss rate, and the overall fairness. Chang and Sohi proposed a cooperative cache partitioning algorithm that optimizes for several metrics, such as the sum of per-thread slowdowns as well as the harmonic mean of per-thread speedups over an equal-partition cache baseline [Chang and Sohi, 2007]. The focus of those studies is to optimize for a single metric.

Typically, to optimize for a given metric, we can set up fixed-length intervals. At the end of each interval, cache partition sizes are re-evaluated to see if they could have achieved a better metric value if they had been partitioned differently.

For example, if the goal is to minimize the total number of cache misses, the number of cache misses in the last interval is compared against the estimated number of misses that would have occurred with different partition sizes. If the number of misses would have been lower with different partition sizes, the partition sizes are changed and applied for the next time interval. How can we know the number of cache misses under hypothetical partition sizes that were not applied in the previous interval? We can collect the stack distance profile of each application, so that the miss rate curve (i.e. the miss rate as a function of cache size) can be collected to tell us the number of misses that would have been under different partition sizes. Alternatively, we can use additional cache tag arrays that keep track of the hypothetical content of the cache, and collect the number of cache hits/misses under a hypothetical scenario that we want to compare against.

Note, however, that optimizing a single metric, be it maximizing weighted speedup, IPC throughput, or fairness, does not eliminate the performance volatility problem. The performance volatility problem can only be addressed with a partition size that is fixed for a given application. Naturally, this leads to a concept of **Quality of Service** (QoS), where an application's performance level is guaranteed by the system, regardless of what other applications run in the system. For this goal, some studies have proposed to provide an abstract machine model to the application, containing a certain number of cores, a certain cache size, off-chip bandwidth rate, memory size, etc [Nesbit et al., 2007, Guo et al., 2007]. The computer system then implements mechanisms to ensure that the application runs at the same performance level as if it runs on this abstract machine. For example, the cache and off-chip bandwidth are partitioned according to the abstract machine. The partition sizes can be significantly reduced in some cases without slowing down the application by a certain capped amount, allowing the system to reallocate the excess partitions for improving the throughput of the system, for example by accepting new applications and assigning them the excess partitions [Guo et al., 2007].

15.3.2 Off-chip Bandwidth Partitioning

In this section, we will discuss a technique for allocating fractions of off-chip bandwidth among cores. Note that naively allocating fixed fractions of off-chip bandwidth to different cores is risky because it can degrade, instead of improve, system performance for two reasons. First, if a thread is allocated a larger fraction of bandwidth than it needs, there will be bandwidth fragmentation in which

it underutilizes the bandwidth fraction, while at the same time other threads may be penalized due to insufficient bandwidth fractions. Secondly, bandwidth usage is often bursty, hence a fixed partition can significantly hurt performance during bursts of memory requests, and suffer from fragmentation during a period of sparse memory requests.

Figure 15.8: A multicore system without off-chip bandwidth partitioning (a), versus with off-chip bandwidth partitioning using a token bucket algorithm (b).

To allocate fractions of off-chip bandwidth to different cores while tolerating the bursty nature of memory requests, we can use the *token bucket algorithm* (Figure 15.8(b)), a bandwidth partitioning algorithm borrowed from computer networking. With a token bucket algorithm, each off-chip memory request is allowed to go to the off-chip interface only when it has a matching token in the respective bucket. There is a token generator that distributes tokens to different buckets at rates proportional to the fractions of bandwidth allocated to different cores. Unused tokens are accumulated in the buckets and are consumed in the future during a period of bursty memory requests. Thus, the depth of the buckets determines the extent of how much burstiness can be tolerated. When tokens for a core are discarded because the bucket is full, they are lost forever, hence the core will never reach the allocated bandwidth fraction. Hence, the selection of the bucket size is

quite important.

If several memory controllers are integrated on chip, each memory controller has its own channel to the off-chip memory. In this case, the token bucket must be implemented for each memory controller, but the off-chip bandwidth rates allocated for a core at the memory controllers must be coordinated so that the total rates in the controllers match the intended total fraction of off-chip bandwidth.

As in cache partitioning, the ability to enforce bandwidth partitions is just one part of the solution to manage bandwidth usage. Another part that is just as important is how to determine the size of the partitions. Again, this is related to what the goal of bandwidth usage management. There are various possibilities: maximize the overall system performance, ensuring fairness of bandwidth usage, providing quality of service, or a combination of them.

Let us assume that the goal is to optimize for system performance, as expressed in weighted speedup. In this section, we discuss briefly the findings from Liu et al's study [Liu et al., 2009]. First, the study finds that without managed bandwidth usage, the bandwidth is naturally shared by applications that run on different cores, and the bandwidth share is proportional to the share of miss (including writeback) frequencies coming from the core divided by the total miss frequencies. Specifically, assuming P number of cores, if a core runs thread i that generates miss rate M_i and access frequency A_i (the number of accesses per unit time), its natural bandwidth share (β_i) can be expressed as:

$$\beta_i = \frac{M_i A_i}{\sum_{j=1}^{P} M_j A_j} \tag{15.1}$$

For example, if a core generates misses at a frequency that is twice the miss frequency of another core, then the first core will naturally use two thirds of the available bandwidth compared to the second core.

Using a constrained optimization method, the study shows that the bandwidth partitions that optimize for weighted speedup can be expressed as:

$$\beta_{opt,i} = \frac{(M_i A_i)^{2/3} (\frac{CPI_{alone,i}}{CPI_{L2\infty,i}})^{1/3}}{\sum_{j=1}^{P} (M_j A_j)^{2/3} (\frac{CPI_{alone,j}}{CPI_{L2\infty,j}})^{1/3}} \tag{15.2}$$

where $CPI_{alone,i}$ expresses the CPI of thread i when it runs alone in the system, having access to the entire available bandwidth. $CPI_{L2\infty,i}$ expresses the CPI of thread i when it runs alone with an infinite L2 cache size, which implies that it suffers from zero capacity and conflict misses.

Let us compare the natural versus optimum bandwidth shares. They differ in two respects. First, the optimum bandwidth shares are affected by CPIs of threads, whereas the natural bandwidth shares are not. The reason for this is because to optimize weighted speedup, threads that are more sensitive to extra memory latency due to colliding requests to access the off-chip bandwidth can be given a larger bandwidth share, whereas threads that are less sensitive can be allocated a smaller share. This results in the increase of performance in delay-sensitive threads in excess of the decrease of performance in delay-insensitive threads. The sensitivity of threads to extra memory latency is captured by the two CPIs.

Secondly, the miss frequencies are exponentiated differently in the two cases, with the exponent of $\frac{2}{3}$ for the optimum bandwidth share, but with the exponent of 1 for the natural bandwidth share. The effect of this is that the optimum bandwidth share for an application that incurs a very high relative miss frequency to be slightly smaller than its natural bandwidth share.

Comparing the two factors, the miss frequencies are potentially the stronger contributor for several reasons. First, the CPIs contribute as a ratio, hence the CPI bias of an application (very high or very low CPI) is somewhat neutralized when we take a ratio. Second, the CPI ratio is exponentiated by a smaller power ($\frac{1}{3}$) versus a larger power ($\frac{2}{3}$) in miss frequencies. Finally, when there is a high number of cores, the size of the cache when a thread runs alone in the system is many times larger than when a thread is coscheduled with others. As a result, $CPI_{alone,i}$ will approach $CPI_{L2\infty,i}$, causing their ratio ($\frac{CPI_{alone,i}}{CPI_{L2\infty,i}}$) to approach 1.

Suppose that we have a two-core CMP with two threads running on them. Suppose also that we ignore the contribution of the CPI ratios on the optimum bandwidth shares, i.e. $\frac{CPI_{alone,1}}{CPI_{L2\infty,1}} = \frac{CPI_{alone,2}}{CPI_{L2\infty,2}} = 1$. Since the optimum bandwidth is now a function of miss frequencies of both threads, Figure 15.9 plots the optimum bandwidth vs. natural bandwidth share for thread 2 as a function of the ratio of miss frequencies of thread 2 to thread 1. From the figure, we could see that when the miss frequency ratio is 1, the natural bandwidth share for thread 2 stands at 50%, equal to the optimum bandwidth partition. As the miss frequencies between the two applications diverge, the natural bandwidth share vs. optimum bandwidth partition diverges as well, especially when the miss frequencies differ a lot (e.g., $\frac{M_2A_2}{M_1A_1} < 0.5$ or $\frac{M_2A_2}{M_1A_1} > 2$). The figure shows that an application with a larger miss frequency tends to obtain a larger bandwidth share when no parti-

tioning is applied, compared to the optimum bandwidth partition size. Therefore, compared to unmanaged bandwidth usage, the optimum bandwidth partitioning tends to slightly (but not overly) constraint applications with high miss frequencies as if to prevent them from dominating the bandwidth usage and starving the applications with lower miss frequencies.

Figure 15.9: Natural vs. optimum fractions of bandwidth allocated to thread 2 as a function of miss frequency ratio of thread 2 to thread 1.

15.3.3 Interaction Between Cache and Bandwidth Partitioning

In this section, we would like to understand the relative importance of L2 cache capacity versus off-chip bandwidth contention. It is not possible to separate and compare them on a real system, hence we rely on a simulation study instead.

Comparing cache capacity and off-chip bandwidth contention, bandwidth contention affects system performance differently. Whereas cache capacity contention results in an increased number of cache misses, bandwidth contention instead results in additional queuing delays for memory requests that collide with others waiting for the bus. On the surface, bandwidth contention seems to have only a secondary effect on performance. For example, by partitioning the cache space such that the total number of cache misses is minimized, both the cache contention and bandwidth contention levels are reduced. The hypothesis certainly makes sense because while cache partitioning can significantly reduce the total number of cache misses of applications that share the cache, resulting in a reduction of total off-chip memory traffic, bandwidth partitioning, on the other hand,

cannot reduce the total off-chip memory traffic, since it merely prioritizes misses or write back requests from some cores over others. Hence, it is intuitive to guess that the performance impact of bandwidth partitioning is likely secondary to cache partitioning.

Taking it one step further, one may further hypothesize that when cache partitioning is employed, bandwidth partitioning may be unnecessary because it has little additional performance benefit compared to that of cache partitioning. This hypothesis also makes sense in that when off-chip bandwidth is plentiful, bandwidth partitioning is unnecessary, while cache partitioning can still improve performance because of the reduction in the number of cache misses. When off-chip bandwidth is scarce, bandwidth partitioning cannot alleviate the scarcity because it does not reduce the number of memory requests, while cache partitioning can.

Figure 15.10: Weighted speedup of four applications running on a 4-core CMP with a 2MB shared L2 cache, assuming various peak off-chip bandwidths.

Interestingly, however, both hypotheses are not necessarily correct. Figure 15.10 shows a simulation result showing the weighted speedup of four sequential SPEC 2006 applications (*libquantum*, *milc*, *hmmer*, and *gcc*) running on a 4-core CMP system with a 2MB shared L2 cache, with different amounts of available peak off-chip bandwidth. The four bars for each peak bandwidth correspond to when no partitioning is applied, only cache partitioning is applied, only bandwidth partitioning is applied, and both cache and bandwidth partitioning applied. The cache partitioning scheme attempts to minimize the total number of cache misses, the

bandwidth partitioning scheme maximizes the weighted speedup of applications using Equation 15.2. Both cache and bandwidth partitioning decisions are evaluated and re-applied every one million of clock cycles, based on parameter values collected in the prior one-million cycle interval. The simulation models a 4-core system where each core has a scalar in-order issue pipeline with a 3.2GHz clock frequency, a 2-way associative 16KB L1 instruction cache and L1 data cache. The L2 cache is shared among four cores, and has 2MB size and 16-way associativity. All caches have a block size of 64-byte, implement write-back policy, and use the LRU replacement policy. The bus to off-chip memory is a split transaction bus, with a peak bandwidth varied from 1.6GB/s to 25.6GB/s (with a base case of 6.4 GByte/s). The main memory is 2GB in size with a 240-cycle access latency. The CMP runs Linux Operating System that comes with Fedora Core 4 distribution.

The figure shows that bandwidth partitioning can improve the overall system performance. It also shows that bandwidth partitioning can outperform cache partitioning in some cases (1.6 GB/s and 3.2 GB/s). Finally, combined cache and bandwidth partitioning can outperform the product of speedup ratios of individual partitioning techniques, implying that there sometimes is a synergistic interaction between them.

To understand more about the interaction between cache and bandwidth partitioning, let us revisit Equation 15.2 which shows partition sizes that maximize weighted speedup. By analyzing the equation, e can deduce that there are two ways that cache and bandwidth partitioning interact. The first way is that since cache partitioning reduces the total number of cache misses, it has the effect of relieving certain off-chip bandwidth contention. More abundant bandwidth results in smaller system performance improvement due to bandwidth partitioning. The second way is cache partitioning may change miss frequency difference between two threads. If the miss frequency discrepancy between threads increases, then the difference between bandwidth partition sizes that optimize for performance versus natural bandwidth partition sizes increases. This implies that the performance improvement that bandwidth partitioning can deliver over unmanaged bandwidth sharing increases as well. If, on the other hand, the miss frequency discrepancy decreases, then the performance improvement that bandwidth partitioning can deliver decreases as well.

Overall, cache partitioning may reduce the impact of bandwidth partitioning on system performance by (1) reducing the bandwidth pressure through reducing the total number of cache misses and (2) reducing the difference in miss frequen-

cannot reduce the total off-chip memory traffic, since it merely prioritizes misses or write back requests from some cores over others. Hence, it is intuitive to guess that the performance impact of bandwidth partitioning is likely secondary to cache partitioning.

Taking it one step further, one may further hypothesize that when cache partitioning is employed, bandwidth partitioning may be unnecessary because it has little additional performance benefit compared to that of cache partitioning. This hypothesis also makes sense in that when off-chip bandwidth is plentiful, bandwidth partitioning is unnecessary, while cache partitioning can still improve performance because of the reduction in the number of cache misses. When off-chip bandwidth is scarce, bandwidth partitioning cannot alleviate the scarcity because it does not reduce the number of memory requests, while cache partitioning can.

Figure 15.10: Weighted speedup of four applications running on a 4-core CMP with a 2MB shared L2 cache, assuming various peak off-chip bandwidths.

Interestingly, however, both hypotheses are not necessarily correct. Figure 15.10 shows a simulation result showing the weighted speedup of four sequential SPEC 2006 applications (*libquantum*, *milc*, *hmmer*, and *gcc*) running on a 4-core CMP system with a 2MB shared L2 cache, with different amounts of available peak off-chip bandwidth. The four bars for each peak bandwidth correspond to when no partitioning is applied, only cache partitioning is applied, only bandwidth partitioning is applied, and both cache and bandwidth partitioning applied. The cache partitioning scheme attempts to minimize the total number of cache misses, the

bandwidth partitioning scheme maximizes the weighted speedup of applications using Equation 15.2. Both cache and bandwidth partitioning decisions are evaluated and re-applied every one million of clock cycles, based on parameter values collected in the prior one-million cycle interval. The simulation models a 4-core system where each core has a scalar in-order issue pipeline with a 3.2GHz clock frequency, a 2-way associative 16KB L1 instruction cache and L1 data cache. The L2 cache is shared among four cores, and has 2MB size and 16-way associativity. All caches have a block size of 64-byte, implement write-back policy, and use the LRU replacement policy. The bus to off-chip memory is a split transaction bus, with a peak bandwidth varied from 1.6GB/s to 25.6GB/s (with a base case of 6.4 GByte/s). The main memory is 2GB in size with a 240-cycle access latency. The CMP runs Linux Operating System that comes with Fedora Core 4 distribution.

The figure shows that bandwidth partitioning can improve the overall system performance. It also shows that bandwidth partitioning can outperform cache partitioning in some cases (1.6 GB/s and 3.2 GB/s). Finally, combined cache and bandwidth partitioning can outperform the product of speedup ratios of individual partitioning techniques, implying that there sometimes is a synergistic interaction between them.

To understand more about the interaction between cache and bandwidth partitioning, let us revisit Equation 15.2 which shows partition sizes that maximize weighted speedup. By analyzing the equation, e can deduce that there are two ways that cache and bandwidth partitioning interact. The first way is that since cache partitioning reduces the total number of cache misses, it has the effect of relieving certain off-chip bandwidth contention. More abundant bandwidth results in smaller system performance improvement due to bandwidth partitioning. The second way is cache partitioning may change miss frequency difference between two threads. If the miss frequency discrepancy between threads increases, then the difference between bandwidth partition sizes that optimize for performance versus natural bandwidth partition sizes increases. This implies that the performance improvement that bandwidth partitioning can deliver over unmanaged bandwidth sharing increases as well. If, on the other hand, the miss frequency discrepancy decreases, then the performance improvement that bandwidth partitioning can deliver decreases as well.

Overall, cache partitioning may reduce the impact of bandwidth partitioning on system performance by (1) reducing the bandwidth pressure through reducing the total number of cache misses and (2) reducing the difference in miss frequen-

cies between threads. On the other hand, cache partitioning may increase the impact of bandwidth partitioning on system performance by increasing the difference in miss frequencies between threads. The synthesis of these factors leads to an interesting situation in which the system performance of some mixes of applications cannot benefit from cache partitioning, but nonetheless cache partitioning can be applied to increase the discrepancy of miss frequencies such that the performance improvement from bandwidth partitioning is boosted.

15.3.4 System Performance versus Performance Volatility

Overall, however, the issue remains unsolved in that achieving a maximum system performance does not fully address the performance volatility problem, while addressing the performance volatility by providing quality of service may prevent us from achieving a maximum system performance. In the end, it is likely that the right solution depends on the characteristics of the target environment and workloads for the multicore architecture.

15.4 Exercise Problems

1. **Cache capacity contention**. Suppose that a multicore has two cores that share a single L2 cache that is 4-way associative, uses LRU replacement policy, and is initially empty. Two threads, X and Y, run on these cores simultaneously. Suppose that X has the following blocks that it accesses on the L2 cache: A B A C B D E A A F C, while Y has the following blocks that it accesses on the L2 cache: U V W X U V W X U V W X ... Suppose that all the accesses map to a single cache set. Determine how many cache space contention misses that X suffers from given the following relative speeds between accesses by X and by Y:

   (a) X's and Y's accesses are interleaved one to one, e.g. one access by thread X, then one access by thread Y, then one access by thread X, etc.

   (b) X's and Y's accesses are interleaved one to two, e.g. one access by thread X, then two accesses by thread Y, then one access by thread X, etc.

   (c) X's and Y's accesses are interleaved two to one, e.g. two accesses by thread X, then one access by thread Y, then two accesses by thread X, etc.

   (d) Compare all the above cases, and state your observations with regard to what behavior makes thread X vulnerable to performance volatility compared to if it runs alone in the system (i.e. without Y running simultaneously).

2. **Cache partitioning**. Assume a core with 3 GHz clock frequency, having a private L1 caches but a shared L2 cache. The caches have a block size of 64 bytes. The core runs an application which has a cache miss rate of 10%, L2 cache access frequency of 0.1 accesses/cycle. Suppose that the core's current partition in the L2 cache is 512KB, which we want to increase to 1MB. How long at the minimum it will take for the L2 cache partition to converge to the new partition size?

3. **Bandwidth partitioning**. Assume that two cores (Core0 and Core1) are generating memory requests at the rate of x GBs/sec and $k \times x$ GBs/sec, respectively.

(a) What are the cores' natural off-chip bandwidth shares for the following values of k? 0.3, 0.5, 0.7

(b) What are the cores' weighted speedup-optimum off-chip bandwidth partition sizes for the following values of k? 0.3, 0.5, 0.7. Assume that the applications on both cores have an equal infinite-L2 CPI and CPI when they run alone in the system.

Chapter 16

Bandwidth Wall Bottleneck

Contents

16.1 Assumptions, Scope, and Limitations of Study **473**
16.2 Modeling CMP Memory Traffic **474**
16.3 Bandwidth Wall Impact on CMP Scaling **476**
16.3.1 Bandwidth Wall Impact on CMP Scaling 476
16.3.2 CMP Scaling and Die Area Allocation 479
16.4 Analyzing Memory Traffic Reduction Techniques **480**
16.4.1 Indirect Techniques 482
16.4.2 Direct Techniques 489
16.4.3 Dual Techniques 491
16.4.4 Discussion and Recommendations 495
16.5 Concluding Remarks **500**
16.6 Exercise Problems **502**

In traditional multiprocessor systems, adding more processors typically involve adding more nodes, and since a node consists of a processor and its memory, the aggregate bandwidth to main memory increases with more processors. The opposite situation occurs in multicore multiprocessor systems. The projected rate of growth in the number of transistors that can fit on a chip (hence the number of cores) in general is much higher than the projected rate of growth of bandwidth to off-chip memory, roughly 50–70% versus 10–15% per year [ITRS, 2005]. This creates a situation in which over time, the amount of off-chip bandwidth each core can use declines. This causes the performance of a multicore architecture to be increasingly limited by the amount of off-chip bandwidth that a core can use.

This problem is referred to as the *bandwidth wall* problem. This chapter is based on a recent study of the bandwidth wall [Rogers et al., 2009].

In general, doubling the number of cores and the amount of cache in a CMP to utilize the growing transistor counts results in a corresponding doubling of off-chip memory traffic. This implies that the rate at which memory requests must be serviced also needs to double to maintain a balanced design. If the provided off-chip memory bandwidth cannot sustain the rate at which memory requests are generated, then the extra queuing delay for memory requests will force the performance of the cores to decline until the rate of memory requests matches the available off-chip bandwidth. At that point, adding more cores to the chip no longer yields any additional throughput or performance.

The net result is that the rate at which memory traffic is generated by an increasing number of cores is growing faster than the rate at which it can be serviced. Similar to the memory wall and power wall problems faced by previous generations of systems, today's designers also face a *bandwidth wall*; total system performance and throughput are increasingly limited by the amount available off-chip bandwidth.

The goal of this chapter is to answer two fundamental questions: (1) *to what extent does the bandwidth wall problem restrict future multicore scaling*, and (2) *to what extent are various bandwidth conservation techniques able to mitigate the bandwidth wall problem.* To answer these questions, we turn to a simple, yet powerful analytical model that projects how the generated memory traffic is affected by the workload characteristics, number of cores on a chip, the total cache size, and the allocation of die area for cores vs. caches. Then, we extend the model to incorporate the impact of various bandwidth conservation techniques on the memory traffic generated by the cores, and we evaluate the effectiveness of each technique in allowing multicore scaling in future systems. The model allows us to arrive at several interesting findings:

- Without bandwidth conservation techniques, the bandwidth wall problem is multiplying at an astounding rate. For example, if the starting configuration has eight cores and 50% die area allocated to cores and 50% to caches, in four technology generations the allocation for caches must grow to 90% (vs. 10% for cores) so that the caches become large enough to reduce the total memory traffic per core such that the total traffic for all cores remains unchanged. Consequently, the number of cores would increase from 8 to only 24 (3× increase), versus 128 cores (16× increase) under "proportional"

scaling.

- Bandwidth conservation techniques vary greatly in their ability to reduce the memory traffic requirement. For example, using DRAM caches allows the number of cores to increase to 47 in four technology generations, whereas leveraging smaller sized cores to gain more chip area for caches has a much more limited benefit to core scaling.

- Techniques which have a direct effect on the memory traffic itself offer significantly more benefit than techniques that reduce memory traffic indirectly by reducing cache miss rates. For example, in four technology generations, link compression can enable 38 cores while cache compression can enable only 30.

- When several bandwidth conservation techniques are combined, some of them allow super-proportional scaling. For example, using the combined techniques of 3D-stacked cache, DRAM caches, cache+link compression, and ideally-sized cache lines, in four technology generations, we can increase the number of cores on a chip to 183 (71% of the die area) while still maintaining the bandwidth envelope. Considering this, it is likely that the bandwidth wall problem can be delayed by several technology generations as long as multiple bandwidth conservation techniques are combined and applied effectively.

16.1 Assumptions, Scope, and Limitations of Study

Before we discuss the model and results, there are several simplifying assumptions that we would like to make. First, we restrict our study to CMPs with uniform cores. A heterogeneous CMP has the potential of being more area efficient overall, and this allows caches to be larger and generates less memory traffic from cache misses and write backs. However, the design space for heterogeneous CMPs is too large for us to include in our model.

We also assume single-threaded cores. The consequence of this assumption is that our study tends to underestimate the severity of the bandwidth wall problem compared to a system with multithreaded cores. This is because multiple threads running on a multi-threaded core tend to keep the core less idle, and hence it is likely to generate more memory traffic per unit time than a simpler core.

In addition, we assume that we always have threads or applications that can run on all cores in the CMP, regardless of the number of cores on a chip. We also assume that characteristics of workload do not change across technology generations. While it is possible that future workloads are more bandwidth efficient (i.e. they generate fewer cache misses), past trends point to the contrary, as the working set of the average workload has been increasing due to the increasing complexity of software systems. This means that the future severity of the bandwidth wall problem may be worse than what we project in our study.

In addition to these assumptions, there are configurations that we use as the base, and apply throughout the study, unless mentioned otherwise. For example, we assume a configuration in which each core has a private L2 cache and threads that run on different cores do not share data with each other, but we relax this assumption when we discuss the impact of data sharing across threads. Another example is that we assume the size of cores, in terms of number of transistors, remains unchanged across technology generations, but we relax this assumption when we discuss the impact of using smaller cores in future generations.

For the scope of this study, we do not evaluate the power implications of various CMP configurations, and only qualitatively discuss the implementation complexity of the bandwidth conservation techniques we study.

Finally, the projection obtained by our analytical model is the memory traffic generated for different configurations for a constant amount of *computation work*, rather than for a constant unit of time (i.e. bandwidth). Both metrics are equal if the rate at which memory traffic is generated does not change. However, many factors affect the latter metric (bandwidth), including the features of the core, cache configuration, as well as queuing delay at the off-chip interconnect. Predicting the actual bandwidth demand complicates our model with no additional insights in the bandwidth saturated scenario that we assume. Hence, we focus on predicting the total memory traffic generated instead.

16.2 Modeling CMP Memory Traffic

At the basic level, our analytical model must relate memory traffic to the workload characteristics, the number of cores on a chip, the total cache size, and the allocation of die area for cores vs. caches. Recall the power law model for a single processor system that we discussed in Section 6.3.1. The power law states that if m_0 is the miss rate of a workload for a baseline cache size C_0, the miss rate (m)

for a new cache size C can be expressed as:

$$m = m_0 \cdot \left(\frac{C}{C_0}\right)^{-\alpha} \tag{16.1}$$

where α is a measure of how sensitive the workload is to changes in cache size.

For our CMP memory traffic model, we extend the power law model to include the effect of additional memory traffic due to write backs, and multiple cores on a chip. We find that for a given application, the number of write backs tends to be an application-specific constant fraction of its number of cache misses, across different cache sizes. If such a fraction is denoted as r_{wb}, then the total memory traffic becomes $M = m \times (1 + r_{wb})$, where m is the miss rate. Substituting it into Equation 16.2, the $(1 + r_{wb})$ term appears in both the numerator and denominator, and cancel out each other, leaving us with:

$$M = M_0 \cdot \left(\frac{C}{C_0}\right)^{-\alpha} \tag{16.2}$$

which states that the power law holds for the memory traffic as well.

To take into account multiple cores on a chip, we first assume that the chip's die area is divided into some number of Core Equivalent Areas (CEAs). One CEA is equal to the area occupied by one processor core and its L1 caches. Additionally, we assume that on-chip components other than cores and caches occupy a constant fraction of the die area regardless of the process technology generation.

Table 16.1: CMP system variables used in our model

CEA	Core Equivalent Area (die area for 1 core)
P	# of CEAs for cores(equivalent to # cores)
C	# of CEAs for on-chip cache
N	= P+C, total chip die area in terms of CEAs
S	= C/P, amount of on-chip cache per core

Table 16.1 shows the major parameters of our model which capture the fact that the die area in a CMP is allocated between cores and caches. Assuming that threads do not share data (Section 16.1), all cores in a CMP generate cache miss and write back traffic independently. Thus, the total memory traffic of the entire multicore chip is equal to the number of cores multiplied by the memory traffic

generated by each core:

$$M = P \cdot M_0 \cdot \left(\frac{S}{S_0}\right)^{-\alpha} \tag{16.3}$$

where $S = C/P$. In order to compare the memory traffic from different CMP configurations, suppose that we have a baseline configuration with P_1 cores and S_1 amount of cache per core, and a new configuration with P_2 cores and S_2 cache per core. Dividing the expressions from Equation 16.3 for both configurations, we obtain:

$$\frac{M_2}{M_1} = \frac{P_2 \cdot M_0 \cdot \left(\frac{S_2}{S_0}\right)^{-\alpha}}{P_1 \cdot M_0 \cdot \left(\frac{S_1}{S_0}\right)^{-\alpha}} = \frac{P_2}{P_1} \cdot \left(\frac{S_2}{S_1}\right)^{-\alpha} \tag{16.4}$$

$$M_2 = \left(\frac{P_2}{P_1}\right) \cdot \left(\frac{S_2}{S_1}\right)^{-\alpha} \cdot M_1 \tag{16.5}$$

That is, the memory traffic of two different configurations depend on two terms: $\frac{P2}{P1}$, which accounts for the difference in the number of cores, and $\left(\frac{S2}{S1}\right)^{-\alpha}$, which depends on the change in the cache space that is allocated to each core. Example 16.1 shows how the equation can be used to estimate off-chip traffic demand under various core vs. cache configurations.

16.3 Bandwidth Wall Impact on CMP Scaling

The previous section described an analytical model for memory traffic as a function of die allocation to cores versus cache in the same processor generation. In this section, we will look at how the memory traffic requirement changes in future technology generations.

16.3.1 Bandwidth Wall Impact on CMP Scaling

To investigate the impact of the bandwidth wall on CMP scaling, we start with a baseline CMP configuration in the current process technology generation, with parameters chosen to be similar to that of Sun Niagara2 [McGhan, 2006]. In this baseline, we assume a balanced configuration with 8 cores and 8 CEAs allocated for on-chip L2 cache (roughly corresponding to 4MB in capacity). The parameters corresponding to this baseline configuration are $N_1 = 16$, $P_1 = 8$, $C_1 = 8$, and $S_1 = 1$. We will also assume that $\alpha = 0.5$, representing the characteristic of

■ **Example 16.1** An illustration of how off-chip traffic demand for a particular core vs. cache configuration can be estimated.

Suppose that a workload has $\alpha = 0.5$, and a baseline CMP configuration with 8 cores and 1 CEA of cache per core, for a total of 16 CEAs. If we reallocate 4 CEAs from caches to make room for four extra cores, how much does the off-chip traffic demand increase, compared to the case prior to reallocation?

Answer: The initial configuration has $P1 = 8$ and $C1 = 8$. In the new configuration, $P2 = 8 + 4 = 12$ and $S2 = \frac{8-4}{8+4} = \frac{1}{3}$. Using Equation 16.5, we find that

$$\begin{aligned} M_2 &= \left(\frac{12}{8}\right) \cdot \left(\frac{\frac{1}{3}}{1}\right)^{-0.5} \cdot M_1 \\ &= 1.5 \cdot 1.73 \cdot M_1 \\ &= 2.6 \cdot M_1 \end{aligned} \tag{16.6}$$

Thus, the new configuration yields memory traffic demand of $2.6\times$ more than the baseline configuration, of which $1.5\times$ is due to the increase in the number of cores, and $1.73\times$ is due to the increase in the memory traffic per core because of the reduced cache size per core.

an average commercial workload. We also assume that in future generations, we wish to keep the memory traffic the same as in the baseline CMP configuration.

Now, let us suppose that twice as many transistors are available in the following technology generation, and so we now have 32 CEAs of available area (assuming each core uses the same number of transistors as before). We can allocate P_2 cores, leaving $C_2 = 32 - P_2$ CEAs for cache, resulting in a cache-to-core ratio of $S_2 = \frac{32-P_2}{P_2}$. Substituting P_2 values ranging from 1 to 28 into Equation 16.5, we can determine how memory traffic demand changes as a function of the number of cores on a chip, normalized to the baseline system (Figure 16.1).

The figure shows that as we increase the number of processor cores (and reduce the area allocated for caches), the memory traffic relative to the baseline configuration grows super-linearly. If we wish to double the number of cores from the baseline to 16, then the memory traffic increases by a factor of 2. Unfortunately, off-chip pin bandwidth will not likely keep up as pin count only grows a projected 10% per year [ITRS, 2005]. If we assume that memory traffic should remain constant in the next technology generation, then the new CMP configuration can only support 11 cores (the intersection of the two curves in Figure 16.1),

which is equivalent to an increase of only 37.5%. Even when we assume that the bandwidth envelope can grow by an optimistic 50% in the next generation, the number of cores can only increase by 62.5% to 13. Of course, it is always possible to build more cores on a chip if one is willing to sacrifice a balanced design. Unfortunately, adding more cores beyond the bandwidth envelope will force total chip performance to decline until the rate of memory requests matches the available off-chip bandwidth. Even worse is the fact that the area taken up by excess cores could have been allocated for more productive use such as for extra cache capacity.

Figure 16.1: Memory traffic as the number of CMP cores varies in the next technology generation.

Another way of looking at CMP scaling is to look at how fast the cache size per core must grow in order to reduce the memory traffic per core sufficiently to compensate for the "proportional" increase in the number of cores (i.e. doubling the core count every process generation). To arrive at the answer, let us first assume that memory bandwidth grows by a factor of B in each technology generation. This causes the aggregate memory traffic of the CMP to also grow by a factor of B, i.e., $M_2 = B \cdot M_1$. Incorporating this into Equation 16.5, we can express:

$$\left(\frac{P_2}{P_1}\right) \cdot \left(\frac{S_2}{S_1}\right)^{-\alpha} = \frac{M_2}{M_1} = B \tag{16.7}$$

which can be rearranged to explicitly express the required cache size per core (S_2) to support the desired number of cores (P_2):

$$S_2 = S_1 \cdot \left(\frac{P_2}{B \cdot P_1} \right)^{\frac{1}{\alpha}} \tag{16.8}$$

Figure 16.2 shows the amount of cache area per core required to sustain a proportional CMP scaling where the number of cores continually doubles and memory bandwidth scaled by a different factor ranging from $1\times$ to $2\times$ per technology generation. The axes in the figure are shown in log-scale. The figure shows that proportional CMP scaling requires a staggering growth in the required cache size per core, even assuming a highly-optimistic bandwidth growth of $1.4\times$ per generation. For example, by the time 1024 cores can fit on chip, the total cache size must increase by more than $100\times$, and the cache size per core will have to be more than $100\times$ bigger than the core itself. Overall, the figure illustrates the enormity of the bandwidth wall problem.

Figure 16.2: Cache area per core to sustain desired CMP scaling.

16.3.2 CMP Scaling and Die Area Allocation

One useful angle for chip designers to look into the bandwidth wall problem is to determine the fraction of chip area that should be allocated to cores versus cache to achieve a balanced CMP design in a future technology generation. To arrive at this answer, suppose that memory bandwidth grows by a factor of B in each technology generation, allowing the aggregate memory traffic of a next-generation CMP to grow by a factor of B relative to a current design, i.e., $M_2 = B \cdot M_1$. Incorporating this into Equation 16.5, we can express:

$$\left(\frac{P_2}{P_1}\right) \cdot \left(\frac{S_2}{S_1}\right)^{-\alpha} = \frac{M_2}{M_1} = B \tag{16.9}$$

Next, we rework Equation 16.9 by expressing S_2 in terms of N_2, which is known given our fixed chip area constraints, and P_2, which is what we want to find, i.e. $S_2 = \frac{N_2 - P_2}{P_2}$. Substituting this expression for S_2 in Equation 16.10, we obtain the following expression:

$$\left(\frac{P_2}{P_1}\right) \cdot \left(\frac{\left(\frac{N_2 - P_2}{P_2}\right)}{S_1}\right)^{-\alpha} = B \tag{16.10}$$

Solving for P_2 numerically, we obtain the results shown in Figure 16.3.

Figure 16.3 reveals that if we keep the memory traffic constant across process generations, the fraction of the die area allocable to cores declines rapidly. At $16\times$ scaling (four process generations from now), only 10% of the die area can be allocated for cores, corresponding to 24 cores (versus $16 \times 8 = 128$ cores under proportional scaling). The die area allocation for cores decreases even more at future process generations.

16.4 Analyzing Memory Traffic Reduction Techniques

In Sections 16.2 and 16.3, our model has shown that the bandwidth wall, left unaddressed, will quickly dampen CMP scaling and force die area allocation heavily tilted toward caches. Considering the severity of the problem, we believe that future large-scale CMPs must take the bandwidth wall problem as a primary concern. The goal of this section is to evaluate the effectiveness of various bandwidth conservation techniques at addressing the bandwidth wall problem.

Recall how memory traffic is impacted by the number of cores and the amount of cache per core (Equation 16.5). There are two main ways to maintain a nearly-constant memory traffic requirement as we increase the number of on-chip cores proportional to the growth in transistor density. The first way is to *indirectly* reduce the memory traffic requirements by increasing the effective on-chip cache capacity, which reduces cache misses. The second way is to *directly* reduce the memory traffic requirements by either reducing the number of bytes that are brought on-chip, or by increasing the raw amount of available off-chip bandwidth. Based on this insight, we divide the discussion of various bandwidth conservation

Figure 16.3: Die area allocation for cores and the number of supportable cores assuming constant memory traffic (a), number of supportable cores for various bandwidth envelope growth rates (b).

techniques into three categories: those in the *indirect* category, *direct* category, and *dual* category for techniques that produce both indirect and direct effects.

Note that the scope of our study is to evaluate existing or promising techniques that have been proposed in literature, and compare them in their effectiveness in reducing memory traffic requirements. Fully analyzing whether or when the different techniques become feasible for commercial implementation is beyond the scope of this study.

To compare the benefits of various techniques to CMP scaling, we will use the same baseline CMP as in Section 16.3.1, (8 cores and 8-CEA caches) and evaluate how many cores can be supported in the next generation when the number of total CEAs doubles to 32. Under proportional scaling, we would like to be able to

support 16 cores, but the bandwidth wall limits it to only 11 (Figure 16.1).

16.4.1 Indirect Techniques

Indirect techniques reduce memory traffic requirements by increasing the effective cache capacity per core. In our model, the impact is captured by a multiplicative factor F, which we refer to as the *effectiveness factor*:

$$M_2 = \left(\frac{P_2}{P_1}\right) \cdot \left(\frac{\mathbf{F} \cdot S_2}{S_1}\right)^{-\alpha} \cdot M_1 \tag{16.11}$$

For some indirect techniques, the memory traffic equation may not conform to the simple form in Equation 16.11. In these cases, we develop the appropriate traffic model, as necessary. More specifically, such a technique may be able to increase the amount of effective cache space available to a processor by some factor (we call this the *effectiveness factor* of the technique) compared to the same area of cache available to the processor without that technique. We can incorporate the effect of this type of technique into our model by adding the factor F shown in bold: $M_2 = (\frac{P_2}{P_1}) \cdot (\frac{\boldsymbol{F} \cdot S_2}{S_1})^{-\alpha} \cdot M_1$. this follows that different techniques will have different effectiveness factors. Thus they will allow different increases in the ability of the cache to filter memory traffic, and they will enable different amounts of additional cores to be placed on-chip while maintaining our memory traffic requirements.

Note that the impact of increased effective cache capacity per core (i.e. F) on memory traffic is dampened by the $-\alpha$ power. The smaller α is for a workload, the larger the dampening becomes. For example, if $\alpha = 0.9$, to reduce memory traffic by half, the cache size per core needs to be increased by a factor of $2.16\times$. However, if $\alpha = 0.5$, to reduce memory traffic by half, the cache size per core needs to be increased by a factor of $4\times$.

Cache Compression. Cache compression (e.g. [Alameldeen, 2006]) exploits repeated patterns in data values to store them using fewer bits. A hardware compression engine is used to compress data before it is placed in an on-chip cache and to decompress it when the processor needs it. Prior work on cache compression has shown that a compression scheme with reasonable area and latency overheads can achieve compression ratios (i.e. effectiveness factors) of 1.4x to 2.1x for a set of commercial workloads, 1.7x to 2.4x for the SPEC2k integer benchmarks, and 1.0x to 1.3x for the SPEC2k floating point benchmarks [Alameldeen, 2006, Alameldeen and Wood, 2004a, Alameldeen and Wood, 2004b].

Figure 16.4 shows the implications of this observation on the number of cores that can be supported in a next-generation CMP with 32 total CEAs, if the memory traffic requirement is kept constant relative to a baseline system with 8 cores and 8-CEA caches. Recall from Section 16.3 that without cache compression, our new CMP can support only 11 cores under a constant memory traffic requirement. The figure shows the number of cores possible using a cache compression scheme with various effectiveness factors (i.e. compression ratio) within the ranges observed in [Alameldeen, 2006, Thuresson et al., 2008], as well as an upper-limit case.

Figure 16.4: Increase in number of on-chip cores enabled by cache compression (32 CEAs).

With cache compression support with a compression ratio of 1.3x, 1.7x, 2.0x, 2.5x, and 3.0x, the number of supportable cores grows to 11, 12, 13, 14, and 14 respectively. As CMP systems scale, this savings potentially can represent a larger raw number of additional cores that can be placed on chip. However, unless the compression ratios reach the upper end of achievable range, the ability of cache compression in helping CMP scaling is relatively modest. Note that our analysis does not factor the power, complexity, and die area overheads needed to implement the compression engine.

Also, as discussed above, the benefits of compression in creating the potential for more on-chip cores diminishes as the compression ratios become larger. This suggests that the most effective design point for a cache compression scheme intended to reduce memory traffic requirements is a small, low-overhead hardware design which is able to obtain a reasonable compression ratio with minimal latency overheads.

DRAM Cache. Another way to increase the effective cache capacity is to implement caches using denser memory such as DRAM. The increase in density from replacing SRAM caches with DRAM caches is estimated to be between

8× [Black et al., 2006] and 16× [Yamauchi et al., 1997]. Figure 16.5 shows the impact of changing the on-chip L2 cache from an SRAM to a DRAM technology on the number of supportable cores in the next-generation CMP under a constant memory traffic requirement.

Figure 16.5: Increase in number of on-chip cores enabled by DRAM caches.

Recall from Section 16.3 that moving from our UltraSPARC T2-like baseline CMP with 16 CEAs to a next-generation CMP with 32 CEAs only allows us 3 additional cores for a total of 11 without oversaturating our memory bandwidth requirement. The bar labeled "SRAM L2" represents the baseline case in which the L2 caches are implemented in SRAM. The remaining bars show the number of cores that can be supported without increasing the memory traffic demand if DRAM technology with the above mentioned density assumptions is used to implement the L2 cache. From this figure, we can see that the desired proportional scaling of 16 cores is possible even assuming a conservative density increase of 4× in the DRAM implementation. With 8× and 16× density improvement, the situation improves further and allows super-proportional scaling to 18 and 21 cores respectively. While there are other implementation aspects to consider, such as the refresh capacity needed for DRAM or possible access latency increases, DRAM caches provide a very effective means of filtering the extra memory traffic generated by a larger number of cores, and they alleviate the bandwidth wall problem by slightly more than one technology generation.

3D-Stacked Cache. Another potential solution to achieve a large on-chip cache capacity is 3D chip stacking, where an additional die consisting of cache storage is stacked vertically on top of the traditional CMP die. We restrict our analysis to the case where there are only 2 dies stacked and the processors only occupy one die. The cache occupies the entire extra die, and, potentially, a share of

the processor die. We also assume that the cache that shares the die with the processors is a traditional SRAM cache. For the extra (cache-only) die, we investigate the impact of an SRAM cache as well as DRAM cache with various density improvements over SRAM Our assumptions for this analysis are based on prior work on 3D caches and die-stacking [Black et al., 2006, Puttaswamy and Loh, 2005].

To incorporate 3D stacking into our model, we note that the extra cache-only die increases the area for cache allocation by N CEAs. The new total CEAs for cache are $N + (N - P)$. If the cache-only-layer of the stacked CMP is implemented using D×-denser DRAM cells, then the effective cache CEAs on the cache-only layer is $D \cdot N$, and the total cache CEAs for the CMP are $D \cdot N + (N - P)$. Substituting the S_2 term in the memory traffic model of Equation 16.5 by $\frac{D \cdot N_2 + (N_2 - P_2)}{P_2}$, we get:

$$M_2 = \left(\frac{P_2}{P_1}\right) \cdot \left(\frac{\left(\frac{D \cdot N_2 + (N_2 - P_2)}{P_2}\right)}{S_1}\right)^{-\alpha} \cdot M_1 \tag{16.12}$$

Figure 16.6 shows the impact of adding a 3D-stacked die for cache storage on the number of supportable cores in the next generation CMP under a constant memory traffic requirement.

Figure 16.6: Increase in number of on-chip cores enabled by 3D-stacked caches.

The bar labeled "No 3D Cache" corresponds to the next-generation CMP without an additional die for caches, which can support only 11 cores. The remaining bars show how the caches in the additional die are implemented: in SRAM or DRAM with 8× or 16× density improvement over SRAM. From this figure, we can see that adding a die layer of SRAM caches allows 14 cores to be implemented, and 25 and 32 cores when DRAM caches are used with 8× or 16×,

respectively. The result shows that the density improvement of DRAM caches, coupled with an entire die allocated for them, result in a highly effective way to mitigate the bandwidth wall, allowing super-proportional scaling.

Unused Data Filtering. Previous work [Chen et al., 2004, Qureshi et al., 2007] has shown that a substantial amount of cached data (roughly 40%) is unused by the processor because of the mispredicted spatial locality inherent in typical cache line sizes. Unused data filtering consists of techniques that improve cache efficiency by retaining only useful words in a cache line and discarding unused words. While unused data filtering might not be designed primarily as a memory traffic reduction technique, its impact on memory traffic is significant enough to merit its inclusion in our model. There are also related studies [Chen et al., 2004, Kumar and Wilkerson, 1998, Pujara and Aggarwal, 2006] which reduce memory traffic directly by selectively reading in predicted-useful words of a cache line, and they will be discussed in Section 16.4.2 under the broad category of Sectored Caches.

Figure 16.7: Increase in number of on-chip cores enabled by filtering unused data from the cache.

Using Equation 16.11 and effectiveness factors for unused-data-filtering techniques from previous research [Qureshi et al., 2007] we can evaluate the sensitivity of CMP scaling to this class of techniques. Figure 16.7 shows that under optimistic assumptions in which 80% of cached data goes unused, corresponding to a $5\times$ effective increase in cache capacity, a proportional scaling to 16 cores can be achieved. Under our more realistic assumptions in which 40% of cached data goes unused, the techniques provides a much more modest benefit of one additional core in the next technology generation.

Smaller Cores. One approach for allowing aggressive core scaling is to use simpler, smaller cores. One reason behind this approach is that more complex pro-

cessors waste bandwidth due to speculative techniques geared towards increasing the throughput of each individual core. With less speculation, smaller cores can be more area, power, and bandwidth efficient. This efficiency allows a greater number of cores to be placed on-chip and improves the overall throughput of the chip, even as the throughput of each core is lower compared to a more complex core. However, there is a limitation to this approach. There is a certain design point which is most efficient in terms of performance per unit area and bandwidth per unit area. Any further reduction in the size of the core will yield no overall throughput benefit.

In the discussion of all the other memory traffic reduction techniques, we have assumed an area- and bandwidth-efficient base core design, such that simplifying the core further is not a good design choice. In order to draw insights into the effect of changing the core size, in this section we assume that going to a smaller core leads to a more efficient design. In addition, we assume that the memory traffic requirements (not bandwidth) of the smaller core remain the same as the larger core, that is, the total number of cache accesses generated by the smaller core remains unchanged. A smaller core, typically, also generates cache and memory accesses slower than a more complex core, even when running at the same frequency as a more complex core. This reduces the chip's memory bandwidth demand (but not memory traffic demand) compared to the chip with the same number of larger cores. This reduction in bandwidth demand is similar to artificially reducing the frequency of the cores to reduce the rate at which memory traffic is generated by the cores, causing a corresponding reduction in memory bandwidth, and a corresponding increase in the number of cores supported by the chip. Therefore, to avoid the artificial effects of bandwidth reduction on scaling, we assume that even with smaller cores the overall bandwidth continues to remain saturated. Under this assumption, overall memory traffic continues to be a good indicator of overall throughput.

By going to smaller cores, the die area occupied by the same number of cores declines. This frees up die area that can be reallocated to increase cache capacity and reduce memory traffic. As defined in Table 16.1, the amount of cache per core, S, is equal to $\frac{N-P}{P}$. Let the smaller core be a fraction, f_{sm}, of the larger base core. Then the new effective amount of cache per core, S' can be expressed as:

$$S' = \frac{N - f_{sm} \cdot P}{P} \tag{16.13}$$

Replacing S_2 in Equation 16.5 by the expansion for S_2' as suggested by Equation 16.13 we get:

$$M_2 = \left(\frac{P_2}{P_1}\right) \cdot \left(\frac{\left(\frac{N_2 - f_{sm} \cdot P_2}{P_2}\right)}{S_1}\right)^{-\alpha} \cdot M_1 \tag{16.14}$$

Using Equation 16.14 and solving for P_2 for different values of f_{sm}, we can evaluate the sensitivity of CMP scaling to the size of the core. Figure 16.8 shows the number of smaller on-chip cores the next-generation chip can support. We use chip size estimates from prior work [Kumar et al., 2005a, Kumar et al., 2004] which shows that simpler smaller cores can be up to 80× smaller.

Figure 16.8: Increase in number of on-chip cores enabled by smaller cores.

Note that even with significantly smaller cores, the system does not scale very well even across one generation. The reason is that even when the core is infinitesimally small and the cache occupies the entire chip, the amount of cache per core only increases by 2×, whereas for proportional core scaling the cache needs to grow by 4×.

Therefore, the main reason going to a simpler core design allows a more aggressive core scaling is because simpler cores are also slower in generating memory traffic, hence they naturally fit within a lower bandwidth envelope. For example, if a simpler core is twice as slow and twice as small as a more complex one, we can fit twice as many simple cores while keeping the same bandwidth envelope. However, in such a case, the scaling is obtained directly at the cost of lower per-core performance. In addition, in practice, there is a limit to this approach, since with increasingly smaller cores, the interconnection between cores (routers, links, buses, etc.) becomes increasingly larger and more complex.

16.4.2 Direct Techniques

We have discussed that by increasing the effective cache capacity per core, the memory traffic demand from each core can be reduced. However, the benefit of such techniques on memory traffic reduction is dampened by the power of $-\alpha$ in Equation 16.5. A more promising approach is to increase the actual or effective bandwidth term, $\frac{M_2}{M_1}$, directly. To directly increase the available bandwidth, the industry has relied on (1) increasing the frequency of the off-chip memory interface, or (2) increasing the number of channels to off-chip memory. For example, Sun's Niagara2 went with Fully Buffered memory DIMMs compared to the DDR2 memory in Niagara1, increasing the peak memory bandwidth from about 25GBps to 42GBps [McGhan, 2006]. IBM's Power6 chip doubled the number of memory controllers and memory channels on chip, and went to a faster clocked DDR2 memory compared to the Power5 chip [Le et al., 2007] (800MHz vs. 533MHz). Unfortunately, increasing the actual bandwidth is eventually limited by the growth in pin counts and in the power and cooling requirements from a higher DRAM frequency.

Link Compression. One technique to increase the effective memory bandwidth is link compression, which reduces the amount of data communicated from/to off-chip memory by receiving/sending it in a compressed form. Previous research has shown that with relatively simple compression schemes, such as those which exploit value locality, memory bandwidth demand can be reduced by about 50% for commercial workloads, and up to 70% for integer and media workloads [Thuresson et al., 2008]. In other words, link compression, can result in 2x to 3x increase in effective bandwidth. Though link compression may be applied only once, its potential impact and its orthogonality to other techniques make this promising. Figure 16.9 shows the number of supportable cores in a next-generation CMP assuming various compression ratios. The figure shows that proportional scaling is achievable, while a super-proportional scaling is a possibility.

Sectored Caches. As discussed in the section on Unused Data Filtering, studies have found that many words in a cache line go unused. Here, we consider techniques that exploit this finding by fetching on chip only words that are likely to be referenced. We evaluate their potential benefit assuming that a cache line is broken into sectors, and when a cache line is requested, only sectors that will be referenced by the processor are fetched and fill the cache line. Because unfilled sectors still occupy cache space, a sectored cache only reduces memory traffic but

Figure 16.9: Increase in number of on-chip cores enabled by link compression.

does not increase the effective cache capacity. We restrict our analysis to evaluating techniques which use filtering and prediction capabilities to read in and write back only the necessary sectors(also known as sub-blocks) of a cache line, thus reducing memory traffic demand. The sectors are stored in the cache as part of the full cache line; no improvement in cache space efficiency is expected. Smaller cache lines, which affect both bandwidth inefficiency and cache inefficiency, are discussed in Section 16.4.3.

Figure 16.10 shows the number of supportable cores in a next generation CMP, assuming various fractions of data that are unused by the processor, as in Unused Data Filtering in Section 16.4.1. The figure shows that Sectored Caches have more potential to improve core scaling compared to Unused Data Filtering, especially when the amount of unused data is high.

Figure 16.10: Increase in number of on-chip cores enabled by a sectored cache.

16.4.3 Dual Techniques

The previous sections explored two classes of techniques to overcome the bandwidth wall: those that directly increase the effective bandwidth at the chip interface, and those that indirectly reduce off-chip traffic through increasing the effective cache capacity. This section considers mechanisms that simultaneously achieve both of these benefits.

Small Cache Lines. Historically, larger cache lines have been used to improve application performance by banking on the existence of spatial locality for all words in a cache line. However, if not all words in a line are accessed, unused words waste bandwidth in two ways: they consume off-chip bandwidth during their transfer from the off-chip memory, and they reduce the cache capacity available to useful words while they are cached on chip. Therefore, while using smaller line sizes can increase the number of cache misses, it can reduce memory traffic both directly and indirectly, since only useful words are brought in and stored in the cache.

Prior work [Chen et al., 2004, Qureshi et al., 2007] has studied the amount of useless traffic for various line sizes and concluded that, on average, 40% of the 8-byte words in a 64-byte cache line are never accessed. This represents a 40% decrease in effective cache capacity and, potentially, a corresponding 40% decrease in memory bandwidth effectiveness. Let f_w denote the average fraction of unused cache space, we can represent the new memory traffic demand from using smaller cache lines as:

$$M_2 = \left(\frac{P_2}{P_1}\right) \cdot \left(\frac{S_2 \cdot \left(\frac{1}{1-f_w}\right)}{S_1}\right)^{-\alpha} \cdot \frac{M_1}{\left(\frac{1}{1-f_w}\right)} \tag{16.15}$$

where $\frac{1}{1-f_w}$ represents the factor by which the cache space increases and the memory traffic decreases. Figure 16.11 shows the number of supportable cores in a next-generation CMP for various fractions of unused data, with an assumption that unreferenced words consume neither bus bandwidth nor cache capacity, corresponding to the case in which word-sized cache lines are used. We point out the *realistic* case, which shows that a 40% reduction in memory traffic enables proportional scaling (16 cores in a 32-CEA). Note also that our base cache line size of 64 bytes is on the small end. Some current systems use larger L2 cache line size, in which case we would expect the effectiveness of this technique to be greater.

Figure 16.11: Increase in number of on-chip cores enabled by smaller cache lines.

Cache + Link Compression. If link and cache compression are applied together, i.e. compressed data transmitted over the off-chip link is also stored compressed in the L2 cache, memory traffic can be reduced directly and indirectly. In our model, such a combined compression is expressed by replacing the $\frac{1}{1-f_w}$ terms in Equation 16.15 with the achievable compression ratio. Figure 16.12 shows the number of supportable cores in a next generation CMP with various compression ratios. The dual benefit of direct and indirect reduction of memory traffic improves core scaling substantially. For example, even a moderate compression ratio of 2.0 is sufficient to allow a super-proportional scaling to 18 cores, while higher compression ratios allow even higher scaling.

Figure 16.12: Increase in number of on-chip cores enabled by cache+link compression.

Data Sharing. So far we have assumed that threads that run in different cores do not share data. We recognize that parallel scientific workloads as well as multi-threaded commercial workloads may have substantial data sharing between dif-

ferent threads [Barroso et al., 1998, Cain et al., 2001, Tseng et al., 2007]. In this section, we relax the assumption of no data sharing, and evaluate how data sharing in multi-threaded workloads affect CMP core scaling. However, as with independent threads, we assume *problem scaling*, i.e. each additional thread brings its own working set, except for the data shared by multiple threads.

Data sharing affects our model in two ways. First, a data line that is shared by multiple threads is fetched from off-chip memory into the on-chip cache by only one thread. Secondly, how much cache capacity is occupied by a shared data block depends on the cache configuration. If we assume each core has a private L2 cache, then the block will be replicated at different caches by all cores which access it. However, if we assume the cores share the L2 cache (which may be physically distributed), the block only occupies one cache line. In order to estimate the full potential for CMP scaling due to data sharing, we assume a shared cache configuration. Thus, data sharing brings both a direct and indirect reduction in memory traffic. We also restrict our analysis to the scenario where data is either private to a thread, or is shared by all threads on the chip. This simplification, in addition to the shared cache assumption, overstates the benefit provided by data sharing in most cases, thus giving an upper bound to its effectiveness.

From the point of view of modeling, the effect of data sharing impacts the memory traffic as if there are fewer *independent* (non data sharing) cores which are generating memory requests because the first core that accesses a block suffers a cache miss, but other cores' requests result in cache hits. The effect of data sharing also appears in the cache capacity as if there are fewer independent cores which store their working sets in the shared cache, because a shared line stored in the cache by one core belongs to the working sets of several threads that run on different cores. Thus, to incorporate the effect of data sharing in our model, we define an abstract CMP configuration that has P_2' independent cores, where $P_2' < P_2$, such that our CMP power law model holds:

$$M_2 = \left(\frac{P_2'}{P_1}\right) \cdot \left(\frac{C_2/P_2'}{S_1}\right)^{-\alpha} \cdot M_1 \tag{16.16}$$

where P_2' is an unknown value that must be estimated [1]. Let f_{sh} represent the average fraction of data cached on-chip that is shared by threads executing on different cores. To estimate P_2', we analyze how f_{sh} directly affects memory

[1] If private caches are assumed, a shared block occupies multiple cache lines as it is replicated at multiple private caches. Thus, the cache capacity per core is unchanged. In the equation, $\frac{C2}{P_2'}$ should be replaced with $\frac{C2}{P2}$.

traffic requirement. Note that each core fetches its own private data (hence we have $(1 - f_{sh}) \cdot P_2$ fetchers), but only one core fetches the shared data (hence we have f_{sh} fetchers). Thus, there is a total of $f_{sh} + (1 - f_{sh}) \cdot P_2$ cores that fetch data from off-chip memory. However, we also know that in the abstract CMP model, we have P_2' independent cores that fetch data from off-chip memory. Since they must be equal, we can estimate P_2' as:

$$P_2' = f_{sh} + (1 - f_{sh}) \cdot P_2 \tag{16.17}$$

Figure 16.13 shows the impact of data sharing with various fraction shared (f_{sh}) values on memory traffic requirements, normalized to our baseline CMP of 8 cores and 8-CEA caches. Each line corresponds to a future technology generation where we would like to continue proportional core scaling to 16, 32, 64, and 128 cores. An important observation from this figure is that in order to keep the memory traffic constant (at 100%) while still allowing proportional core scaling in each subsequent future technology generation, the fraction of shared data in an application must continually increase to 40%, 63%, 77%, and 86%, respectively. Even if the bandwidth envelope grows in each subsequent generation, the trend of a required increase in the fraction of data sharing in future generations still holds.

Figure 16.13: Impact of data sharing on traffic.

Interestingly, in practice the fraction of data sharing in an application tends to decline (rather than increase) as the application runs on more cores. To illustrate this case, we run PARSEC [Bienia et al., 2008] benchmarks on a shared L2 cache multicore simulator, and measure the actual fraction of data sharing. Specifically, each time a cache line is evicted from the shared cache, we record whether the block is accessed by more than one core or not during the block's lifetime. The results are shown in Figure 16.14, which displays the average fraction of cache

lines that are shared by two or more cores. The figure shows that the fraction of shared data in the cache tends to decrease with the number of cores, which is the opposite of what is needed to hold down the memory traffic requirement (Figure 16.13). A possible reason for this is that while the shared data set size remains somewhat constant, each new thread requires its own private working set [Bienia et al., 2008]. Therefore, while an increase in data sharing can reduce the memory traffic requirement, it is likely that such an increase will not occur automatically and instead require a significant reworking of the algorithm or parallelization techniques.

Figure 16.14: Data sharing behavior in PARSEC.

16.4.4 Discussion and Recommendations

In this section, we compare the various memory traffic reduction techniques described in Sections 16.4.1, 16.4.2, and 16.4.3 in their effectiveness at enabling CMP scaling in several future generations. We start with a comparison of individual techniques.

Figure 16.15 plots the number of supportable on-chip cores given by our model for four future technology generations (with 2, 4, 8, and 16 times the amount of on-chip transistors available), assuming a baseline of 16 available CEAs (8 for cores, 8 for cache) and memory traffic that is kept constant across generations. The *IDEAL* case shows proportional core scaling, while the *BASE* case corresponds to the number of supportable cores if no memory traffic reduction techniques are used. Each remaining configuration corresponds to one of the traffic reduction techniques. Data sharing improvement techniques are excluded because they require program, rather than architecture, changes, and they have been covered in their respective section. The labels used on the x-axis are specified in

Table 16.2. Each data point in the figure shows a main point representing a realistic assumption, with a candle bar range representing the spread between optimistic and pessimistic assumptions (Table 16.2). The case of a 3D-stacked cache only shows a single point because we only consider an additional layer of SRAM die.

Figure 16.15: Core-scaling with various techniques for four future technology generations.

Table 16.2: Summary of memory traffic reduction techniques.

Technique	Label	Realistic	Pessimistic	Optimistic
Cache Compress	CC	2x compr.	1.25x compr.	3.5x compr.
DRAM Cache	DRAM	8x density	4x density	16x density
3D-stacked Cache	3D	3D SRAM layer	-	-
Unused Data Filter	Fltr	40% unused data	10% unused data	80% unused data
Smaller Cores	SmCo	40x less area	9x less area	80x less area
Link Compress	LC	2x compr.	1.25x compr.	3.5x compr.
Sectored Caches	Sect	40% unused data	10% unused data	80% unused data
Cache+Link Cprs	CC/LC	2x compr.	1.25x compr.	3.5x compr.
Smaller Cache Ln	SmCl	40% unused data	10% unused data	80% unused data

The figure clearly shows the growing gap between ideal versus baseline core scaling. Comparing the effect of various memory traffic reduction techniques, in general we observe that the "indirect" techniques which increase the effective amount of data that can be cached on-chip (CC, DRAM, 3D, Fltr, and SmCo) are much less beneficial than "direct" techniques which affect the actual amount of data that must be brought on-chip to execute a workload (LC and Sect), which are in turn less beneficial than "dual" techniques (SmCl and CC/LC). The reason for this is evident from our model (Equation 16.5): the benefit of indirect techniques is dampened by the $-\alpha$ exponent, while the benefit of direct techniques

directly reduces memory traffic requirements. One exception to this general trend is DRAM storage for on-chip caches, but only because of the huge $8\times$ density improvement over SRAM.

It is also interesting to note that the ranges of potential core scaling benefits offered by the direct techniques are quite large, depending on the effectiveness of the technique. While under realistic assumptions, no technique can outpace the ideal proportional scaling, under optimistic assumptions, small cache lines can outpace proportional core scaling for up to three future technology generations. The variability between the impact of optimistic versus realistic assumptions implies that the actual workload characteristics should be fully understood to accurately evaluate the impact of small cache lines.

Table 16.3: Assessment of memory traffic reduction techniques.

Technique	Effectiveness	Range	Complexity
Cache Compress	Med.	Low	Med.
DRAM Cache	High	Med.	Low
3D-stacked Cache	Med	Low	High
Unused Data Filter	Med.	Med.	Med.
Smaller Cores	Low	Low	Low
Link Compress	High	Med.	Low
Sectored Caches	Med.	High	Med.
Cache+Link Compress	High	High	Low
Smaller Cache Lines	High	High	Med.

Table 16.3 also summarizes qualitative attributes of each technique: the expected benefit to CMP core scaling (*Effectiveness*), the variability of its benefit to core scaling (*Range*), and our estimates of the cost and feasibility to implement it (*Complexity*).

As previously noted, using smaller cores as a memory traffic reduction technique has a low level of effectiveness in aiding CMP core scaling, because the die area freed up for additional cache space has a $-\alpha$ exponent dampening effect on memory traffic reduction. However, since smaller cores are slower, the memory traffic will be stretched over a longer time period, allowing them to fit the bandwidth envelope more easily, but at a direct cost to performance. We classify cache compression, 3D-stacked cache, unused data filtering, and sectored caches as techniques with a medium level of effectiveness. While these techniques also

increase the effective cache capacity, compared to using smaller cores, they have the potential to increase the amount of cache per core much more drastically (up to $5\times$). Therefore, they provide a moderate benefit to CMP core scaling. Sectored caches directly impact memory traffic requirements by reducing the amount of data that must be fetched on-chip. However, since the fraction of memory traffic that can be eliminated is modest (40% in our realistic assumptions), it only has a medium level of effectiveness. The most effective techniques for CMP core scaling are DRAM caches, link compression, smaller cache lines, and cache+link compression. Interestingly, although DRAM caches only affect memory traffic indirectly, their huge $8\times$ density improvement over SRAM allows very effective core scaling.

Cache compression, 3D-stacked cache, and smaller cores have a narrow range of variability, due to their small range of reasonable assumptions. This implies that the benefit of these techniques on core scaling is more predictable and does not vary much based on workload characteristics. DRAM caches, unused data filtering, and link compression have a medium variability. It is not surprising that more effective techniques have a larger variability. Finally, sectored caches, smaller lines, and cache+link compression have a large variability, and their potential benefits can vary drastically with different workload characteristics (e.g. the compressibility of data, and the degree of spatial locality). It is also interesting to note that the majority of variability for these schemes is in the positive direction, i.e. the number supportable cores with realistic assumptions is much closer to the number of cores with pessimistic assumptions. The fundamental reason is that the impact of these techniques becomes magnified as they become more effective. For example, sectored caches can reduce the memory traffic by $5\times$ for a workload which does not use 80% of its fetched data, versus $10\times$ for a workload which does not use 90%.

DRAM caches, smaller cores, and link compression have relatively low complexity compared to the other techniques, and in some cases they could be implemented in current-generation systems. Because of the low complexity of DRAM caches, as well as their high effectiveness and low variability, this is an extremely promising technique to enable CMP core scaling in the face of bandwidth wall. Cache compression, unused data filtering, sectored caches, cache+link compression and smaller cache lines are all likely more complex to implement. Cache compression requires a cache organization that handles variable-sized lines, in addition to the compression engine. Unused data filtering, sectored caches, and

smaller cache lines will likely require a mechanism to accurately predict which sections of a cache line will not be referenced. Link compression seems more promising than cache compression both because of its effectiveness and its lower complexity because the cache design does not change. Lastly, we rank 3D-caches as the most complex option because it relies on a complex new technology. 3D-stacked caches on their own have a limited effectiveness and the complexity hurdle, but they become a lot more effective when combined with other techniques.

Figure 16.16: Core-scaling with combinations of various techniques for four future technology generations.

Figure 16.16 shows similar results as Figure 16.15 for various combinations of memory traffic reduction techniques, with an increasing number of techniques as we go to the right of the x-axis. The figure includes ideal and baseline scaling, along with the various combinations of techniques that we evaluate. Again, the vertical axis shows the number of on-chip cores that can be supported at a constant memory traffic requirement for four technology generations in the future. This figure shows several significant insights. The rightmost case, which combines all of the highly effective techniques we evaluate, shows a very promising result: even with realistic assumptions about the effectiveness of individual techniques, super-proportional scaling is possible for all four future technology generations (183 cores vs. 128 cores with proportional scaling, at the fourth future generation). It follows that the most effective individual techniques from Figure 16.15 also have the most benefit when combined with other techniques. Together, the combination of link compression and small cache lines alone can directly reduce memory traffic by 70%, while the combination of 3D-stacked DRAM cache, cache compression, and small cache lines, can increase the effective cache capacity by 53×, indirectly reducing memory traffic by 84%.

Finally, Figure 16.17 shows the effect of different α values on CMP core scaling. Recall that α is essentially a measure of an application's sensitivity to changes in cache size. Applications which behave according to different values of $alpha$

will be affected differently by the memory traffic reduction techniques that affect effective cache size. In this figure, we evaluate the minimum and maximum α values (that we observed in Section 6.3.1 Figure 6.9) on a select group of techniques starting with DRAM caches and working up through combinations of DRAM caches with several other techniques. An important observation from this figure is that α can have a large impact on core scaling in future generations. In the baseline case, a large α enables almost twice as many cores as a small α. When traffic reduction techniques are applied, the gap in number of supportable cores between a large and small α's grows even more. A small α prevents the techniques from achieving a proportional scaling, while a large α allows a super-proportional scaling.

Figure 16.17: Core scaling with select techniques for a high and low α.

16.5 Concluding Remarks

We have discussed how we approach to answer two key questions: (1) to what extent the bandwidth wall problem restricts future multicore scaling, and (2) to what extent various bandwidth conservation techniques are able to mitigate this problem. We find that the bandwidth wall problem can severely limit core scaling. When starting with a balanced, 8 core CMP, in four technology generations the number of cores can scale to only 24, as opposed to 128 cores which is desired under proportional scaling, without increasing the memory traffic requirement.

We find that the impact of the various bandwidth conservation techniques we evaluate have a variation of benefits to core scaling. For example, using DRAM caches can potentially enable 47 cores in four technology generations, while other techniques such as using smaller cores provide a limited benefit to core scaling.

Also, we find that when combined together, these techniques have the potential to enable super-proportional core scaling (183 cores for the combination of all techniques we evaluate) for up to four technology generations.

However, most of the architecture techniques discussed, while able to reduce off-chip traffic demand, are not free from the performance and energy consumption point of view. Cache and link compressions have the effect of increasing cache access latencies and memory access latencies, respectively. They also require decompression and compression engines to be implemented on the die and consume energy to operate. Unused data filtering requires a predictor to be implemented, and suffers from performance loss from mispredicting useful data as non-useful. Sectored caches suffer from a similar performance cost when the needed words in the cache are not found. Smaller cores slow down the performance of each thread, and there is a limit to how small the cores can be made while still reducing energy efficiency. Small cache lines suffer from the increase in address bus occupancy, a larger cache tag array, and from lower efficiency at the main memory for fetching small data chunks. DRAM caches, although denser, will have a lower performance per capacity because DRAM imposes destructive reads, and needs refresh. 3D stacked cache is an exception. It will likely not reduce performance or impose much die area overheads, except for vertical wire connections between layers. However, at the moment, it is not yet practical to implement at the current technology and may have a lower manufacturing yield compared to non-stacked die.

Therefore, we believe the major implication of this bandwidth wall study is that we should not rely on any particular technology to solve the problem, but rather need to think in terms of *bandwidth-efficient computing*, in which design for computing platforms at all levels must take into account how to conserve off-chip bandwidth, in order to allow multicore scaling to translate into gain in the overall performance of the system. In addition to the architecture-level techniques, we should also think about improvement at the application level, algorithm level, and the operating system and runtime levels. Interestingly, there is a potential that reducing off-chip traffic requirement through algorithm changes may simultaneously increase its performance. Techniques to reduce working set sizes, improve spatial and temporal locality, have such potentials. Other possible changes will be to exploit pipeline parallelism instead of data parallelism. All these are beyond the scope of this book, but readers are encouraged to venture into thinking about these possibilities.

16.6 Exercise Problems

1. **Core vs. cache allocation**. Suppose a baseline processor has 8 cores and 8 CEAs allocated for L2 cache. If the die area for 2 cores is reallocated for caches, how much is the memory traffic impacted, assuming:

   (a) the workload's α is 0.3

   (b) the workload's α is 0.5

   (c) the workload's α is 0.7

2. **DRAM cache**. Suppose a given workload has $\alpha = 0.5$. If we add a layer of cache on top of the current one that already has 8 cores and 8 CEAs of SRAM cache, what will be the new memory traffic requirement in the current generation, and in the next generation assuming we would like to have a proportional scaling to 16 cores? Repeat the exercise if instead of an additional layer of SRAM cache, we add a layer of DRAM cache with 8x density improvement over SRAM.

3. **Small cache line size**. Suppose it is known that for a given workload that has $\alpha = 0.5$, 50% of all words brought into the cache are unused by the processor. If we reduce the cache line size to word size, what will be the new memory traffic requirement in the current generation, and in the next generation? Repeat the exercise for $\alpha = 0.25$ and $\alpha = 0.65$?

4. **Direct vs. indirect techniques**. Why are in general direct techniques are more effective in reducing memory traffic than indirect techniques?

5. **Dual techniques**. What techniques have dual benefit of reducing memory traffic both directly and indirectly?

6. **Core scaling**. Suppose as a processor designer you currently have a CMP product with 8 cores and 8 CEAs of cache. As you project the design of the product out for the next four technology generations, you would actually like to achieve super-proportional scaling where your next four generation CMP systems have 18, 36, 72, and 144 cores respectively. Assuming that the amount of off-chip memory traffic that can be handled by the bandwidth system is projected to grow by 15% at each generation, how much effective cache per core is required in each of the next four generations in order to support the desired number of cores (assuming $\alpha = 0.5$)?

7. **Core scaling**. The bandwidth conservation techniques evaluated in this chapter are not generally scalable across multiple generations. In other words, each technique can provide a boost to the number of supportable CMP cores once. However, its effectiveness does not increase in future technology generations to allow even more cores. Let us assume that the technique of cache+link compression is indeed scalable such that its effectiveness increases by 20% at each generation (e.g. advances in compression techniques allow data to be compressed by an additional 20% each generation). Assume a baseline CMP with 8 cores and 8 CEAs of cache. Also assume that in the next technology generation, cache+link compression is applied with an effectiveness of 2x, and that the compression effectiveness improves by 20% in every following generation after that. How many cores are enabled in such a scenario for the next four technology generations (assuming $\alpha = 0.5$)?

8. **Comparing two techniques**. Assume that a given workload has $\alpha = 0.25$, and that your current baseline CMP system has 8 cores and 8 CEAs of SRAM L2 cache. If the next generation CMP will have 32 total CEAs available, evaluate the tradeoffs of two different approaches to enabling more cores to be allocated from those 32 CEAs. The first technique is changing the cache technology to DRAM with an 8x density advantage over SRAM. The second technique is rewriting the target workload so that its locality is enhanced such that its alpha value increases to 0.8. Compare these two approaches in terms of the number of on-chip cores they can enable.

International Symposium on Architecture Support for Programming Languages and Operating Systems.

[Standard Performance Evaluation Corporation, 2006] Standard Performance Evaluation Corporation (2006). http://www.spec.org.

[Strauss et al., 2007] Strauss, K., Shen, X., and Torrellas, J. (2007). UncoRq: Unconstrained Snoop Request Delivery in Embedded-Ring Multiprocessors. In *Proc of the International Symposium on Microarchitecture*.

[Suh et al., 2002] Suh, G. E., Devadas, S., and Rudolph, L. (2002). A New Memory Monitoring Scheme for Memory-Aware Scheduling and Partitioning. In *Proc. of International Symposium on High Performance Computer Architecture*, pages 117–128.

[Tanenbaum, 1992] Tanenbaum, A. (1992). *Modern Operating Systems*. Prentice Hall.

[Thuresson et al., 2008] Thuresson, M., Spracklen, L., and Stenstrom, P. (2008). Memory-Link Compression Schemes: A Value Locality Perspective. *IEEE Transactions on Computers*, 57:916–927.

[Tseng et al., 2007] Tseng, J., Yu, H., Nagar, S., Dubey, N., Franke, H., and Pattnaik, P. (2007). Performance Studies of Commercial Workloads on a Multicore System. In *Proc of the International Symposium on Workload Characterization*, pages 57–65.

[Windheiser et al., 1993] Windheiser, D., Boyd, E., Hao, E., Abraham, S., and Davidson, E. (1993). KSR1 Multiprocessor: Analysis of Latency Hiding Techniques in a Sparse Solver . In *Proceedings of the 7th International Parallel Processing Symposium*.

[Yamauchi et al., 1997] Yamauchi, T., Hammond, L., and Olukotun, K. (1997). A Single Chip Multiprocessor Integrated with High Density DRAM. Technical report, Stanford University.

[Zhang and Asanovic, 2005] Zhang, M. and Asanovic, K. (2005). Victim Replicatoin: Maximizing Capacity while Hiding Wire Delay in Tiled Chip Multiprocessors. In *Proc. of the International Symposium on Computer Architecture*.

List of Abbreviations

AAT Average Access Time, referring to a metric that represents the mean of time to access the memory hierarchy.

CCR communication-to-computation ratio, referring to the amount of communication of a thread divided by the amount of computation of the thread. The parameters are the number of processors and the input size.

CISC Complex Instruction Set Computer, referring to an instruction set architecture (ISA) which has relatively complex instructions that can perform several simple operations in one instruction. For example, an instruction in CISC may involve multiple addressing modes, accessing both register and memory operands, etc.

CMP Chip Multi-Processors, referring to a multiple processor cores implemented in a single chip. Also referred to as multicore processors.

COMA Cache-Only Memory Architecture, referring to a multiprocessor architecture in which the main memory is organized like a cache, supporting cache-like placement, replication, and eviction.

DMA Direct Memory Access, referring to a device that can transfer data in and out of the memory (main memory, or on-chip scratch memory) without involving the processor. The processor is only interrupted at the completion of the transfer.

DRAM Dynamic Random Access Memory, referring to a type of memory that can be accessed randomly but loses its content over time. DRAM cells need to be refreshed periodically to keep its content.

DSM Distributed Shared Memory, referring to a multiprocessor architecture in which nodes have their local memories that abstracted as a single shared memory. Also referred to as NUMA.

FSM Finite State Machine, referring to a machine that has states and transitions from one state to another state. The transition in an FSM depends only on the type of event and the current state.

ILP Instruction Level Parallelism, referring to a parallelism that exists at the level of instructions, in which different instructions from a sequential flow are executed in parallel because they are speculated or discovered to be independent from one another.

ITG Iteration-space Traversal Graph, referring to a graphical representation of the order of traversal in the iteration space. ITG shows the order in which an iteration is visited before another.

LDG Loop-carried Dependence Graph, referring to a graph that shows true, anti, and output dependences graphically, in which a node is a point in the iteration space and the directed edge shows the direction of the dependence.

LDS Linked data structures, referring to a data structure that consists of nodes that are linked by pointers. Examples include linked lists, hash tables, trees, and graphs.

LINPACK refers to a parallel benchmark containing a library of numerical linear algebra. Linpack is a benchmark used for ranking the performance of top supercomputers in www.top500.org.

LL Load Linked or Load Locked, referring to a load instruction that is linked to a specific address. LL is paired up with a matching store conditional (SC) instruction. LL is used for implementing other basic primitives such as atomic instructions and locks.

LRU Least Recently Used, referring to a replacement policy in caches in which the cache block that was accessed the farthest back in the past is selected to be evicted to make room for a new block.

MESI refers to a cache coherence protocol that keeps four states for a cache block: Modified, Exclusive, Shared, and Invalid.

MIMD Multiple Instruction Stream Multiple Data Stream, referring to an multiple processing elements execute from different instruction streams and the instructions execute different data. MIMD is the most flexible machine in the category of machines under the Flynn's taxonomy.

MISD Multiple Instruction Stream Single Data Stream, referring to an architecture in which multiple processing elements execute from a single instruction stream, and data is passed from one processing element to the next. MISD is one category of machines under the Flynn's taxonomy.

MLP Memory Level Parallelism, referring to the overlap between multiple accesses to the main memory.

MOESI refers to a cache coherence protocol that keeps five states for a cache block: Modified, Owned, Exclusive, Shared, and Invalid.

MPI Message Passing Interface, referring to an application programming interface (API) standard for expressing parallelism where processes interact with each other through explicit messages.

MSHR Miss Status Handling Register, referring to a register that tracks the status of a cache miss, in order to facilitate waking up the load instruction that causes the cache miss, or to merge a written value to the block before it is stored in the cache. MSHRs allow multiple cache misses to be serviced simultaneously.

MSI refers to a cache coherence protocol that keeps three states for a cache block: Modified, Shared, and Invalid.

NUMA Non-Uniform Memory Architecture, referring to a multiprocessor architecture in which nodes have their local memories that abstracted as a single shared memory. Also referred to as DSM.

OOO Out of Order, referring to a technique to exploit parallelism among instructions by issuing instructions out of the program order, based on other criteria such as whenever their operands are ready.

OpenMP refers to an application programming interface (API) standard for expressing parallelism at the program source code. OpenMP includes a platform-independent language extension for Fortran/C/C++, a library, a and runtime system.

OS Operating System, a software layer between the hardware and applications. It manages hardware, gives an abstraction of the machine to applications, and protects one application from another.

PARSEC Princeton Application Repository for Shared-memory Computers, a collection of shared memory parallel benchmarks consisting of primarily emerging non-scientific computing applications.

PC Processor Consistency, referring to a memory consistency model that is almost as strict as sequential consistency, but allows a younger load to bypass an older store.

RC Release Consistency, referring to a memory consistency model that allows reordering of memory accesses except at synchronization boundaries, and the synchronization accesses are distinguished between acquire and release accesses.

RISC Reduced Instruction Set Computer, referring to an instruction set architecture (ISA) which has relatively simple instructions that are amenable to pipelining.

SC refers to either Store Conditional instruction, or Sequential Consistency model. Store conditional instruction is matched with a load link (LL) instruction to provide an effect of atomicity in a sequence of instructions. SC fails when the address linked by an earlier load link mismatches the address in SC when the SC is executed.

SC refers to either Store Conditional instruction, or Sequential Consistency model. Sequential consistency model refers to a model of the ordering of memory accesses that corresponds the closest to programmers' expectation.

SIMD Single Instruction Stream Multiple Data Stream, referring to a parallel architecture in which a single instruction operates on multiple data. SIMD is one category of machines under the Flynn's taxonomy.

SISD Single Instruction Stream Single Data Stream, referring to an architecture in which a processing element executes from a single stream of instructions, and each instruction operates on a single data stream. SISD is one category of machines under the Flynn's taxonomy.

SMP Symmetric Multi-Processing, referring to a multiprocessor system in which the latency to access the main memory from any processor is uniform. SMP often refers to a bus-based multiprocessor.

SMT Simultaneous Multi-Threading, referring to a processor architecture in which a single processor core can execute multiple threads simultaneously. Most processor resources can be shared by the thread contexts, with the exception of thread-specific structures such as program counters and stack pointers.

SPEC Standard Performance Evaluation Corporation, a non-profit corporation formed to establish, maintain and endorse benchmark suites for evaluating the performance of computers. One such suite is the SPEC CPU 2006, a collection of benchmarks to evaluate the performance of the processor and cache memories.

SRAM Static Random Access Memory, referring to a type of memory that can be accessed randomly and does not lose its content over time.

TLB Translation Lookaside Buffer, referring to an on-chip storage that keeps most recent virtual-to-physical address translation. TLB is used in conjunction with the OS page table to support OS memory management functions.

WO Weak Ordering, referring to a consistency model that allows memory accesses to be reordered except at synchronization points.

Index

AMD, 405
    K7, 227
    Opteron, 4, 428
Amdahl's law, 87
    sequential execution time, 89
Application, 388
    applied intelligence, 388
    data mining, 388
    multimedia, 388
    PARSEC, 452
    SPEC, 162
    SPEC 2006, 445
Arithmetic mean of speedups, 389
Atomic instruction
    cache coherence, 250
    illusion of atomicity, 258
    in x86 architecture, 251
    need for, 249

Bandwidth partitioning, 462
    cache partitioning interaction, 465
    natural, 463
    token bucket, 462
    weighted speedup optimum, 463
Bandwidth wall, 394, 399, 472
    3D-stacked cache, 484
    cache compression, 482, 491, 495
    data filtering, 486, 495
    data sharing, 492
    die are allocation, 479
    DRAM cache, 483, 495
    link compression, 489, 491, 495
    power law in, 475
    sectored, 489
    sectored caches, 495
    small cache lines, 491, 495
    smaller cores, 486, 495
Barrier, 269
    centralized barrier, 270
        scalability, 273
    combining tree barrier, 274
        scalability, 274
    deadlock situation, 272
    hardware network, 274
    incorrect implementation, 271
    performance criteria, 270
BlueGene, 15
    barrier network, 275
Bus, 199
    arbitration, 200
    atomic, 200
    physical limitations, 320
    snooper, 201
    split transaction, 200

Cache, 141
    3D stacked, 484
    addressing

physical addressing, 156
virtual addressing, 156
anti-LRU behavior, 424
associativity, 144
average access time, 165
bank, 147
cache line size, 491, 495
capacity sharing, 417, 422
compression, 482, 491, 495
congruence classes, 144
data filtering, 486
direct-mapped, 144
distance locality, 418
DRAM, 483, 495
Error Correcting Code, 3
error detection, 152
example of hierarchy, 143
fully-associative, 144
hit rate, 144
inclusion property, 153
indexing function, 146
lockup-free, 158
multi-level issues, 153
non-blocking, 158, 292
organization in multicore, 395
partitioning, 458
placement policy, 144
prefetching, 168
replacement policy, 144, 149
FIFO, 149
LRU, 149
pseudo LRU, 151
pseudo random, 150
sectored, 486, 489, 495
set-associative, 144
sets, 144
size, 144
split organization, 155
tag bits, 147
tile associativity, 414
tile mapping function, 419
types of misses, 159
ways, 144
write policy, 144
write allocate, 152
write back, 151
write no-allocate, 152
write through, 151
Cache coherence
atomic instruction support, 250
basic approaches, 195
cache-to-cache transfer, 213
coherence controller, 200
correctness requirements
write propagation, 183, 427
write serialization, 183, 427
directory approach, 321
directory protocol, 434
downgrade request, 207
Dragon protocol, 229–235
exclusive ownership, 204
for multi-level caches, 237
inclusion property, 238
for write-through caches, 203–206
global sequencer, 428
in a shared cache, 407
in multicore, 426, 434
intervention, 207
invalidation, 207
MESI protocol, 215–222
MOESI protocol, 222–229
MSI protocol, 206–215

need for protocol, 184
ordering point, 428
problem illustration, 181–183
protocol races, 338
snoop filtering, 238
upgrade request, 207
write invalidate, 202
write propagation, 202
write serialization, 202
write update, 202
Cache misses
capacity misses, 159
Coherence misses, 235
coherence misses, 159, 212
compulsory misses, 159
conflict misses, 159
in multicore, 475
natural misses, 160
power law, 162, 475
stack distance, 164
system-related misses, 159
Cache only memory, 325
Cache partitioning, 458
bandwidth partitioning interaction, 465
goals of, 460
modified LRU replacement, 458
performance volatility, 461
Cache-to-cache transfer, 213, 215, 218
Capacity sharing, 422
cooperating caching, 423
dynamic spill receive, 423
Channel
virtual, 376
CISC, 6
Clean sharing, 209, 221
Coherence controller, 200
Coherence misses, 235
determining factors, 235
false sharing, 235
impact of prefetching, 236
true sharing, 235
Communication
artifactual, 92
inherent, 92
Communication-to-computation ratio (CCR), 92
Compiler
automatic parallelization, 81
limitations, 87
blocking transformation, 100
loop interchange transformation, 102
MIPSPro, 81
Core-hour, 392, 393
Core-hour speedup, 393
Cut-through, 371

Deadlock
in barrier implementation, 272
in interconnection network, 374
virtual channels, 376
Dependence
anti, 32
control flow, 185
inter-thread, 54
loop-carried, 33
loop-independent, 33
output, 32
relation with memory consistency, 185
true, 32
Digital
AlphaServer GS320

protocol race handling, 352
Directory, 321
basic coherence operations, 327
cache-based, 326
centralized, 326
directory cache, 330
distributed, 326
format
coarse bit vector, 329
full bit vector, 328
in SGI Origin 2000, 331
limited pointer, 329
sparse directory, 330
home node, 325
in IBM Power4, 332
in multicore, 434
memory-based, 326
protocol correctness, 352
protocol races, 338
home-centric approach, 345
out-of-sync state, 338
overlapped requests, 342
requestor-assisted approach, 345
storage
DRAM, 331
SRAM, 331
transient states, 347
Dirty sharing, 209, 221
in Dragon protocol, 230
Distance locality
data, 413
data replication, 421
directory, 435
in shared cache, 421
selective data replication, 422
virtual hierarchy, 436
DMA, 173
DOACROSS parallelism, 40
DOALL parallelism, 38
DOPIPE
benefits over DOALL, 47
DOPIPE parallelism, 45
Dragon protocol
diagram, 231
DSM, 13, 198, 319
home node, 325
memory consistency, 356
memory interleaving, 322
page allocation, 323
synchronization support, 354

Earth Simulator, 14
Error correcting code, 152

False sharing misses, 235
Fence instruction
load fence, 297
store fence, 297
Finite state machine, 198
for Dragon protocol, 231
for MESI protocol, 216
for MOESI protocol, 224
for MSI protocol, 207, 213
for write-through coherence cache, 204
in uniprocessor cache, 198
Floating point
operation reordering, 76
Flynn's taxonomy, 11

Geometric mean of speedups, 389

Harmonic mean of speedups, 389

IBM
BlueGene, 239
barrier network, 275
Power4, 395, 405
directory, 332
prefetcher, 172
PowerPC
consistency model, 310
ILP
relation with sequential consistency, 292
Image smoothing algorithm, 51
Inclusion property, 153
multi-level cache coherence, 238
value inclusion, 154
Intel
IA-64
consistency model, 311
Larrabee, 427
MMX, 2, 12
Nehalem, 405
Pentium 4
prefetcher, 172
Pentium Pro, 2
SSE, 12
Xeon, 4, 445
Interconnection fabric, 361
performance metrics, 361
protocol layers, 362
Interconnection network
average distance, 365
bisection bandwidth, 365
channel
physical, 376
virtual, 376
components, 362
deadlock, 374
diameter, 365
network hops, 365
network topology, 365
packet format, 364
router, 378
routing, 371
switch degrees, 365
topology, 366
IPC throughput, 389
Iteration-space traversal graph, 34

Java memory model, 312

Lazy release consistency, 307
software shared memory, 308
Linked data structure
fine-grain parallelization
potential deadlock, 133
Linked data structures, 113
approach to parallelization, 117
fine-grain lock parallelization, 131
global lock parallelization, 127
loop-carried dependence, 115
readers parallelization, 126
recursive traversal, 116
Linpack, 15
Load balancing, 90
dynamic, 96
static, 96
using dynamic task mapping, 90
Load linked instruction, 257
Load speculation
support for sequential consistency, 294
Lock
array-based queuing lock, 265

comparison, 267
fairness, 262
illusion of atomicity, 258
load linked/store conditional, 257
linked register, 259
test and set, 252
test and test and set, 255
ticket lock, 262
Loop-carried dependence graph, 34

Memory consistency
atomicity expectation, 285
correctness of synchronization, 184
DSM, 356
implementation, 309
in uniprocessor system, 185
instruction level parallelism, 292
Java memory model, 312
lazy release consistency, 307
non-deterministic execution, 287
on bus-based multiprocessors, 290
problem illustration, 184–186
processor consistency, 297
program order, 185, 285
programmers' intuition, 284
release consistency, 302
sequential consistency, 287
weak ordering, 299
Memory-level parallelism, 158, 165
Memory-processor speed gap, 142
MESI protocol
diagram, 216
drawbacks, 221
Message passing
programming model, 20–26
Metrics, 389
Arithmetic mean of speedups, 389
Core-hour, 392, 393
Core-hour speedup, 393
Geometric mean of speedups, 389
Harmonic mean of speedups, 389
IPC throughput, 389
Job throughput, 393
MIMD, 12
MIPS
R10000, 296
MISD, 12
MOESI protocol
AMD K7, 227
diagram, 224
owned state, 223
owner selection, 224
Moore's law, 2
slowing of, 3
MPI, 14
MSHR, 158
MSI protocol, 207
Multicore, 387
bandwidth wall problem, 394, 471
cache
anti-LRU behavior, 424
capacity sharing, 417, 423
distance locality, 413, 418
logical organization, 413
physical organization, 405
tile mapping function, 419
cache coherence, 426
broadcast/snoopy, 427
directory protocol, 434
distance locality, 436
global sequencer, 428
cache organization, 395
core size, 486

data sharing, 492
distance locality, 421
directory, 435
interconnection, 409
performance volatility, 397, 443, 445,462
quality of service, 461
scalability, 471
scalability challenges, 399
bandwidth limitation, 399
main memory size, 399
power density, 400
reliability, 400
Mutual exclusion, 186
atomic instruction, 188
disabling interrupts, 186
in multiprocessors, 187
lock, 187
Peterson's algorithm, 189

NEC, 14
NUMA, 13, 103, 319

OpenMP, 14
brief introduction, 60
declaring reduction, 62
directive format, 60
expressing function parallelism, 61
implicit barrier, 62
limitations, 63
Schedule types, 64
scheduling for load balance, 99
types of variables, 62
Operating system
page allocation policy, 104, 323
first touch, 104
round robin, 105
page table, 155
relation with memory consistency, 309
time quanta, 160
Origin 2000
Synchronization, 355
Out-of-order execution, 6
Outstanding transaction table, 200

Page allocation, 323
Page table, 155
Parallel programming
synchronization, 58
algorithm-level, 31, 48
anti-diagonal traversal, 39
code-level, 31
correctness pitfalls, 76
data parallelism, 43
DOACROSS parallelism, 40
DOALL parallelism, 38
DOPIPE parallelism, 45
finding tasks, 35
for linked data structures, 113
function parallelism, 43
image smoothing, 51
incorrect synchronization, 78
incorrect variable scope, 80
loop distribution, 43
mapping tasks to processors, 59
matrix multiplication example, 64
maximum speedup, 87
memory hierarchy considerations, 99
parallelizing compiler, 81
program-level, 31
red-black partitioning, 49
reduction, 55

spatial locality consideration, 101
speedup limitation, 88
steps, 30
techniques for linked lists, 126
thread granularity, 90
variable partitioning, 52
privatization, 53
Parity bit, 152
PARSEC benchmark suite, 452
Performance volatility, 397
bandwidth contention, 462
cache partitioning, 461
capacity contention, 456
reasons for, 456
empirical observation, 445
impact of, 443
lock contention, 398
main memory contention, 397
mitigating, 458
network I/O bandwidth, 397
processor core, 397
quality of service, 461
system performance, 468
temporal reuse patterns, 457
Peterson's algorithm, 189
lack of scalability, 190
Power law of cache misses, 162
PowerPC, 15
Prefetching
coherence misses, 236
hardware prefetching, 168
in multiprocessor systems, 172
metrics
accuracy, 168
coverage, 168
timeliness, 168
prefetch buffers, 169, 170
sequential prefetching, 170
software prefetching, 168
stride prefetching, 170
support for sequential consistency, 293
Processor consistency, 297
comparison with SC, 298
comparison with weak ordering, 302
programmer's intuition, 298
using write buffer, 298
Program order, 185
Protocol races
due to out-of-sync state, 338
due to overlapped requests, 342
handling in GS320, 352
handling in SGI Origin2000, 351
Token coherence, 354

Reduction
operation criteria, 55
Relaxed consistency
fence instruction, 297
memory barrier, 297
Release consistency, 302
acquire synchronization, 302
comparison with weak ordering, 307
downward migration, 303
lazy release consistency, 307
release synchronization, 302
upward migration, 303
Reliability, 400
process variation, 400
soft errors, 400
RISC, 6
Router
architecture, 378

Routing
adaptive, 373
cut-through, 371
deadlock, 374
deadlock freedom, 373
deterministic, 373
dimension-ordered, 374
minimal, 373
store and forward, 371
turn restriction, 375
negative first, 376
north last, 376
west first, 376

Scratchpad memory, 141, 173
Sequential consistency
architecture mechanism, 290–296
definition, 287
in MIPS R10000, 296
load speculation, 294
possible outcome, 287
prefetching, 293
problem with non-blocking cache, 292
relaxation of, 296
write buffer incompatibility, 291
Serializability, 117
conflict criteria, 125
example in linked list operations, 118
SGI
Origin 2000
directory format, 331
protocol race handling, 351
Shared memory
parallel programming, 29
programming model, 20–26
Shared memory multiprocessor
benefits of, 180
SIMD, 11
SISD, 11
SMP, 13, 197
bus snooper, 201
cache-to-cache transfer, 213
snoop filtering, 238
support for, 200
supporting SC, 290
write-back buffer, 201
SMT, 143
Snoop filtering, 239
Soft error, 152
Software shared memory, 308
Spatial locality, 101
effects from page allocation, 104
improvement through loop interchange, 103
improvement through task mapping, 103
in NUMA, 103
SPEC
power law fitting, 162
SPEC 2006, 445
Stack-distance profiling, 164
Store and forward, 371
Store conditional instruction, 257
Sun
Niagara, 4
Niagara2, 395
SPARC
consistency model, 311
partial store ordering, 311
total store ordering, 311
Switch

crossbar, 378
Synchronization
acquire access, 302
back-off
exponential back-off, 255
barrier, 269
barrier network, 274
centralized barrier, 270
combining tree barrier, 274
granularity, 91
impact of missing, 78
in DSM, 354
in-memory, 355
lock
ABQL, 265
comparison, 267
compatibility table, 126
ensuring mutual exclusion, 187
intention read lock, 127
intention write lock, 127
read lock, 126, 131
ticket lock, 262
upgrade lock, 127
write lock, 126, 131
needs for hardware support, 186
performance metrics, 248
point-to-point
barrier, 58
lock, 58
post and wait, 58
usage in DOACROSS, 41
usage in DOPIPE, 46
problem illustration, 186–191
relation with memory consistency, 184
relation with weak ordering, 300
release access, 302

Task mapping
block partitioning, 60
block-wise, 92
column-wise, 92
cyclic partitioning, 60
dynamic, 59
for improving load balance, 90
row-wise, 92
static, 59
Temporal locality
blocking transformation, 100
optimization, 100
performance volatility, 457
Topology
2-D mesh, 409
butterfly, 369
crossbar, 408
high vs. low dimensional, 369
k-ary d-dimensional mesh, 366
k-ary d-dimensional torus, 367
k-ary fat tree, 369
k-ary tree, 367
ring, 409
Translation Lookaside buffer, 155
Translation lookaside buffer
address space identifiers, 157
True sharing misses, 235

Victim replication, 421
Volatile
variable declaration, 185

Weak ordering, 299
comparison with processor consistency, 302

comparison with release consistency, 307
correctness assumptions, 300
effects of data race, 299
synchronization points, 300
Write buffer, 291
in processor consistency, 298
Write propagation
write invalidate, 195
write update, 195

FUNDAMENTALS of PARALLEL COMPUTER ARCHITECTURE

Multichip and Multicore Systems

The world of parallel computers is undergoing a significant change. Parallel computers started as high end computing systems mainly used for scientific computation. Recently, the trend toward a multicore design has enabled an implementation of a parallel computer on a single chip. Parallel computers are now ubiquitous, present in all mainstream architectures for server, desktop, and embedded systems. Single-chip parallel computers now co-exist with larger parallel computer architectures for servers and supercomputers. The purpose of this book is to bring together concepts, technologies, and implementations of parallel computer architectures in a package that is suitable for use as an introductory graduate text.

Topics Highlights

- Shared memory parallel programming techniques: parallelization analysis for loop-intensive code and code that employs linked data structures.
- Mechanisms to ensure that a parallel computer architecture is correct, fast, and scalable: cache coherence protocols, hardware synchronization primitives, and memory consistency models.
- Components of a parallel computer architecture such as caches, directory, and interconnection network.
- Unique problems and challenges in multicore systems: cache organization, performance volatility, bandwidth wall, and others.

Key Features

- Intuition-based discussion: intuition is built before difficult concepts are introduced.
- Emphasis on practical skills: parallel programming concepts and techniques that can be applied to real-world problems are covered.
- "Did you know" boxes to stimulate students' creativity and interests.
- Short and modular chapters encourage students to read each chapter in its entirety.

About the Author

Yan Solihin is an Associate Professor of Electrical and Computer Engineering at North Carolina State University.

ISBN: 978-0-9841630-0-7