

Modern Processor Design

Fundamentals of Superscalar Processors

John Paul Shen
Intel Corporation

Mikko H. Lipasti
University of Wisconsin

Mc Graw Hill **Higher Education**

Boston Burr Ridge, IL Dubuque, IA Madison, WI New York San Francisco St. Louis
Bangkok Bogotá Caracas Kuala Lumpur Lisbon London Madrid Mexico City
Milan Montreal New Delhi Santiago Seoul Singapore Sydney Taipei Toronto

Ex.1019 / Page 1 of 66
TESLA, INC.

The McGraw-Hill Companies

Mc Graw Hill **Higher Education**

MODERN PROCESSOR DESIGN: FUNDAMENTALS OF SUPERSCALAR PROCESSORS

Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2005 by The McGraw-Hill Companies, Inc. All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of The McGraw-Hill Companies, Inc., including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 0 DOC/DOC 0 9 8 7 6 5 4

ISBN 0–07–057064–7

Publisher: *Elizabeth A. Jones*
Senior Sponsoring Editor: *Carlise Paulson*
Developmental Editor: *Michelle L. Flomenhoft*
Marketing Manager: *Dawn R. Bercier*
Project Manager: *Jodi Rhomberg*
Senior Production Supervisor: *Laura Fuller*
Lead Media Project Manager: *Audrey A. Reiter*
Media Technology Producer: *Eric A. Weber*
Senior Coordinator of Freelance Design: *Michelle D. Whitaker*
Cover Designer: *Elise Lansdon*
Compositor: *Interactive Composition Corporation*
Typeface: *10.5/12 Times Roman*
Printer: *R. R. Donnelley Crawfordsville, IN*

Library of Congress Cataloging-in-Publication Data

Shen, John Paul.
Modern processor design : fundamentals of superscalar processors / John Paul Shen, Mikko H. Lipasti.—1 st ed.
p. cm.
Includes index.
ISBN 0–07–057064–7
1. Microprocessors—Design and construction. I. Lipasti, Mikko H. II. Title.

TK7895.M5S52 2005
621.39'16—dc22 2004050406
CIP

www.mhhe.com

To

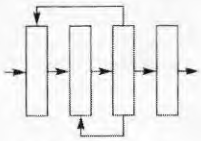
Our parents:
Paul and Sue Shen
Tarja and Simo Lipasti

Our spouses:
Amy C. Shen
Erica Ann Lipasti

Our children:
Priscilla S. Shen, Rachael S. Shen, and Valentia C. Shen
Emma Kristiina Lipasti and Elias Joel Lipasti

General Library System
University of Wisconsin - Madison
728 State Street
Madison, WI 53706-1494
U.S.A.

This material may be protected by Copyright law (Title 17 U.S. Code)



CHAPTER
11

Executing Multiple Threads

CHAPTER OUTLINE

11.1 Introduction
11.2 Synchronizing Shared-Memory Threads
11.3 Introduction to Multiprocessor Systems
11.4 Explicitly Multithreaded Processors
11.5 Implicitly Multithreaded Processors
11.6 Executing the Same Thread
11.7 Summary

References
Homework Problems

11.1 Introduction

Thus far in our exploration of high-performance processors, we have focused exclusively on techniques that accelerate the processing of a single thread of execution. That is to say, we have concentrated on compressing the latency of execution, from beginning to end, of a single serial program. As first discussed in Chapter 1, there are three fundamental interrelated terms that affect this latency: processor cycle time, available instruction-level parallelism, and the number of instructions per program. Reduced cycle time can be brought about by a combination of circuit design techniques, improvements in circuit technology, and architectural tradeoffs. Available instruction-level parallelism can be affected by advances in compilation technology, reductions in structural hazards, and aggressive microarchitectural techniques such as branch or value prediction that mitigate the negative effects of control and data dependences. Finally, the number of instructions per program is determined by algorithmic advances, improvements in compilation technology, and the fundamental characteristics of the instruction set being executed. All these

Ex.1019 / Page 3 of 66
TESLA, INC.

factors assume a single thread of execution, where the processor traverses the static control flow graph of the program in a serial fashion from beginning to end, aggressively resolving control and data dependences but always maintaining the illusion of sequential execution.

In this chapter, we broaden our scope to consider an alternative source of performance that is widely exploited in real systems. This source, called *thread-level parallelism,* is primarily used to improve the throughput or instruction processing bandwidth of a processor or collection of processors. Exploitation of thread-level parallelism has its roots in the early time-sharing mainframe computer systems. These early systems coupled relatively fast CPUs with relatively slow input/output (I/O) devices (the slowest I/O device of all being the human programmer or operator sitting at a terminal). Since CPUs were very expensive, while slow I/O devices such as terminals were relatively inexpensive, operating system developers invented the concept of *time-sharing,* which allowed multiple I/O devices to connect to and share, in a time-sliced fashion, a single CPU resource. This allowed the expensive CPU to switch contexts to an alternative user thread whenever the current thread encountered a long-latency I/O event (e.g., reading from a disk or waiting for a terminal user to enter keystrokes). Hence, the most expensive resource in the system—the CPU—was kept busy as long as there were other users or threads waiting to execute instructions. The time-slicing policies—which also included time quanta that enforced fair access to the CPU—were implemented in the operating system using software, and hence introduced additional execution-time overhead for switching contexts. Hence, the latency of a single thread of execution (or the latency perceived by a single user) would actually increase, since it would now include context-switch and operating system policy management overhead. However, the overall instruction throughput of the processor would increase due to the fact that instructions were executed from alternative threads when an otherwise idle CPU would be waiting for a long-latency I/O event to complete.

From a microarchitectural standpoint, these types of time-sharing workloads provide an interesting challenge to a processor designer. Since they interleave the execution of multiple independent threads, they can wreak havoc on caches and other structures that rely on the spatial and temporal locality exhibited by the reference stream of a single thread. Furthermore, interthread conflicts in branch and value predictors can significantly increase the pressure on such structures and reduce their efficacy, particularly when these structures are not adequately sized. Finally, the large aggregate working set of large numbers of threads (there can be tens of thousands to hundreds of thousands of active threads in a modern, high-end time-shared system) can easily overwhelm the capacity and bandwidth provided by conventional memory subsystems, leading to designs with very large secondary and tertiary caches and extremely high memory bandwidth. These effects are illustrated in Figure 3.31.

Time-shared workloads that share data between concurrently active processes must serialize access to those shared data in a well-defined and repeatable manner. Otherwise, the workloads will generate nondeterministic or even erroneous results. We will consider some simple and widely used schemes for serialization or

synchronization in Section 11.2; all these schemes rely on hardware support for *atomic* operations. An operation is considered *atomic* if all its suboperations are performed as an indivisible unit; that is to say, they are either all performed without interference by other operations or processes, or none of them are performed. Modern processors support primitives that can be used to implement various atomic operations that enable multiple processes or threads to synchronize correctly.

From the standpoint of system architecture, time-shared workloads create an additional opportunity for building systems that provide scalable throughput. Namely, the availability of large numbers of active and independent threads of execution motivates the construction of systems with multiple processors in them, since the operating system can distribute these ready threads to multiple processors quite easily. Building a *multiprocessor system* requires the designer to resolve a number of tradeoffs related primarily to the memory subsystem and how it provides each processor with a coherent and consistent view of memory. We will discuss some of these issues in Section 11.2 and briefly describe key attributes of the *coherence interface* that a modern processor must supply in order to support such a view of memory.

In addition to systems that simultaneously execute multiple threads of control on physically separate processors, processors that provide efficient, fine-grained support for interleaving multiple threads on a single physical processor have also been proposed and built. Such *multithreaded* processors come in various flavors, ranging from fine-grained multithreading, which switches between multiple thread contexts every cycle or every few cycles; to coarse-grained multithreading, which switches contexts only on long-latency events such as cache misses; to simultaneous multithreading, which does away with context switching by allowing individual instructions from multiple threads to be intermingled and processed simultaneously within an out-of-order processor's execution window. We discuss some of the tradeoffs and implementation challenges for proposed and real multithreaded processors in Section 11.4.

The availability of systems with multiple processors has also spawned a large body of research into parallel algorithms that use multiple collaborating threads to arrive at an answer more quickly than with a single serial thread. Many important problems, particularly ones that apply regular computations to massive data sets, are quite amenable to parallel implementations. However, the holy grail of such research—*automated parallelization of serial programs*—has yet to materialize. While automated parallelization of certain classes of algorithms has been demonstrated, such success has largely been limited to scientific and numeric applications with predictable control flow (e.g., nested loop structures with statically determined iteration counts) and statically analyzable memory access patterns (e.g., sequential walks over large multidimensional arrays of floating-point data). For such applications, a parallelizing compiler can decompose the total amount of computation into multiple independent threads by distributing partitions of the data set or the total set of loop iterations across multiple threads. Naturally, the partitioning algorithm must take care to avoid violating data dependences across parallel threads and may need to incorporate synchronization primitives across the

threads to guarantee correctness in such cases. Successful automatic parallelization of scientific and numeric applications has been demonstrated over the years and is in fact in commercial use for many applications in this domain.

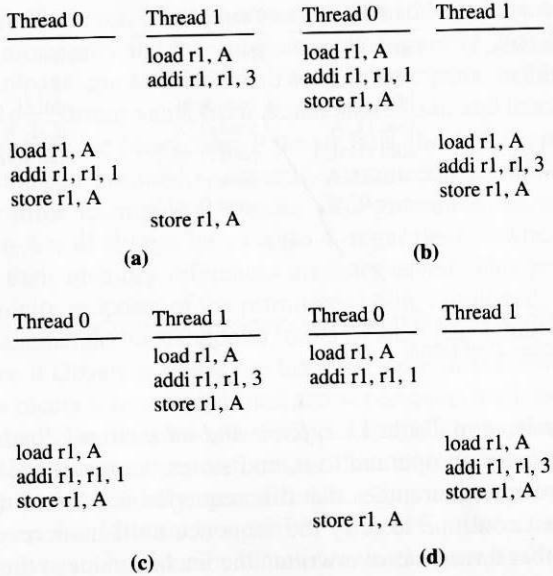
However, there are many difficulties in extracting thread-level parallelism from typical non-numeric serial applications by automatically parallelizing them at compile time. Namely, applications with irregular control flow, ones that tend to access data in unpredictable patterns, or ones that are replete with accesses to pointer-based data structures make it very difficult to statically determine memory data dependences between various portions of the original sequential program. Automatic parallelization of such codes is difficult because partitioning the serial algorithm into multiple parallel and independent threads becomes virtually impossible without exact compile-time knowledge of control flow and data dependence relationships.

Recently, several researchers have proposed shifting the process of automatic parallelization of serial algorithms from compile time to run time, or at least providing efficient hardware support for solving some of the thorny problems associated with the efficient extraction of multiple threads of execution. These *implicit multithreading* proposals range from approaches such as *dynamic multithreading* [Akkary and Driscoll, 1998], which advocates a pure hardware approach that automatically identifies and spawns speculative implicit threads of execution, to the *multiscalar* [Sohi et al., 1995] paradigm which uses a combination of hardware support and aggressive compilation to achieve the same purpose, to *thread-level speculation* [Steffan et al., 1997; 2000; Steffan and Mowry, 1998; Hammond et al., 1998; Krishnan and Torrellas, 2001], which relies on the compiler to create parallel threads but provides simple hardware support for detecting data dependence violations between threads. We will discuss some of these proposals for implicit multithreading in Section 11.5.

In another variation on this theme, researchers have proposed *preexecution,* which uses a second runahead thread to execute only critical portions of the main execution thread in order to prefetch data and instructions and to resolve difficult-to-predict conditional branches before the main thread encounters them. A similar approach has also been suggested for fault detection and fault-tolerant execution. We will discuss some of these proposals and their associated implementation challenges in Section 11.6.

11.2 Synchronizing Shared-Memory Threads

Time-shared workloads that share data between concurrently active processes must serialize access to those shared data in a well-defined and repeatable manner. Otherwise, the workloads will have nondeterministic or even erroneous results. Figure 11.1 illustrates four possible interleavings for the loads and stores performed against a shared variable A by two threads. Any of these four interleavings is possible on a time-shared system that is alternating execution of the two threads. Assuming an initial value of A = 0, depending on the interleaving, the final value of A can be either 3 [Figure 11.1(a)], 4 [Figure 11.1(b) and (c)], or 1 [Figure 11.1(d)]. Of course, a well-written program should have a predictable and repeatable outcome, instead of one determined only by the operating system's task dispatching policies.



This figure shows four possible interleavings of the references made by two threads to a shared variable A, resulting in 3 different final values for A.

Figure 11.1
The Need for Synchronization.

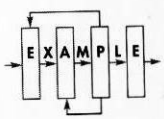


Table 11.1
Some common synchronization primitives

Primitive	Semantic	Comments
Fetch-and-add	Atomic load → add → store operation	Permits atomic increment; can be used to synthesize locks for mutual exclusion
Compare-and-swap	Atomic load → compare → conditional store	Stores only if load returns an expected value
Load-linked/store-conditional	Atomic load → conditional store	Stores only if load/store pair is atomic; that is, if there is no intervening store

This simple example motivates the need for well-defined synchronization between shared-memory threads.

Modern processors supply primitives that can be used to implement various atomic operations that enable multiple processes or threads to synchronize correctly. These primitives guarantee hardware support for *atomic* operations. An operation is considered *atomic* if all its suboperations are performed as an indivisible unit; that is to say, they are either all performed without interference by other operations or processes, or none of them are performed. Table 11.1 summarizes three commonly implemented primitives that can be used to synchronize shared-memory threads.

Thread 0	Thread 1
fetchadd A, 1	fetchadd A, 3

(a)

Thread 0	Thread 1
spin:	spin:
cmpswp AL, 1	cmpswp AL, 1
bfail spin	bfail spin
load r1, A	load r1, A
addi r1, r1, 1	addi r1, r1, 3
store r1, A	store r1, A
store 0, AL	store 0, AL

(b)

Thread 0	Thread 1
spin:	spin:
ll r1, A	ll r1, A
addi r1, r1, 1	addi r1, r1, 3
stc r1, A	stc r1, A
bfail spin	bfail spin

(c)

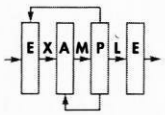


Figure 11.2
Synchronization with (a) Fetch-and-Add, (b) Compare-and-Swap, and (c) Load-Linked/Store-Conditional.

The first primitive in Table 11.1, *fetch-and-add,* simply loads a value from a memory location, adds an operand to it, and stores the result back to the memory location. The hardware guarantees that this sequence occurs atomically; in effect, the processor must continue to retry the sequence until it succeeds in storing the sum before any other thread has overwritten the fetched value at the shared location. As shown in Figure 11.2(a), the code snippets in Figure 11.1 could be rewritten as "`fetchadd A,1`" and "`fetchadd A,3`" for the threads 0 and 1, respectively, resulting in a deterministic, repeatable shared-memory program. In this case, the only allowable outcome would be A = 4.

The second primitive, *compare-and-swap,* simply loads a value, compares it to a supplied operand, and stores the operand to the memory location if the loaded value matches the operand. This primitive allows the programmer to atomically swap a register value with the value at a memory location whenever the memory location contains the expected value. If the compare fails, a condition flag is set to reflect this failure. This primitive can be used to implement mutual exclusion for critical sections protected by locks. Critical sections are simply arbitrary sequences of instructions that are executed atomically by guaranteeing that no other thread can enter such a section until the thread currently executing a critical section has completed the entire section. For example, the updates in the snippets in Figure 11.1 could be made atomic by performing them within a critical section and protecting that critical section with an additional lock variable. This is illustrated in Figure 11.2(b), where the *cmpswp* instruction checks the `AL` lock variable. If it is set to 1, the `cmpswp` fails, and the thread repeats the `cmpswp` instruction until it succeeds, by branching back to it repeatedly (this is known as *spinning on a lock*). Once the `cmpswp` succeeds, the thread enters its critical section and performs its load, add, and store atomically (since mutual exclusion guarantees that no other processor is concurrently executing a critical section protected by the same lock). Finally, the thread stores a 0 to the lock variable `AL` to indicate that it is done with its critical section.

The third primitive, *load-linked/store-conditional* (`ll/stc`), simply loads a value, performs other arbitrary operations, and then attempts to store back to the same address it loaded from. Any intervening store by another thread will cause the store

conditional to fail. However, if no other store to that address occurred, the load/store pair can execute atomically and the store succeeds. Figure 11.2(c) illustrates how the shared memory snippets can be rewritten to use `ll/stc` pairs. In this example, the `ll` instruction loads the current value from A, then adds to it, and then attempts to store the sum back with the `stc` instruction. If the stc fails, the thread spins back to the `ll` instruction until the pair eventually succeeds, guaranteeing an atomic update.

Any of the three examples in Figure 11.2 guarantee the same final result: memory location A will always be equal to 4, regardless of when the two threads execute or how their memory references are interleaved. This property is guaranteed by the atomicity property of the primitives being employed.

From an implementation standpoint, the `ll/stc` pair is the most attractive of these three. Since it closely matches the load and store instructions that are already supported, it fits nicely into the pipelined and superscalar implementations detailed in earlier chapters. The other two, fetch-and-add and compare-and-swap, do not, since they require two memory references that must be performed indivisibly. Hence, they require substantially specialized handling in the processor pipeline.

Modern instruction sets such as MIPS, PowerPC, Alpha, and IA-64 provide ll/stc primitives for synchronization. These are fairly easy to implement; the only additional semantic that has to be supported is that each `ll` instruction must, as a side effect, remember the address it loaded from. All subsequent stores (including stores performed by remote processors in a multiprocessor system) must check their addresses against this linked address and must clear it if there is a match. Finally, when the `stc` executes, it must check its address against the linked address. If it matches, the `stc` is allowed to proceed; if not, the `stc` must fail and set a condition code that reflects that failure. These changes are fairly incremental above and beyond the support that is already in place for standard loads and stores. Hence, `ll/stc` is easy to implement and is still powerful enough to synthesize both fetch-and-add and compare-and-swap as well as many other atomic primitives.

In summary, proper synchronization is necessary for correct, repeatable execution of shared-memory programs with multiple threads of execution. This is true not only for such programs running on a time-shared uniprocessor, but also for programs running on multiprocessor systems or multithreaded processors.

11.3 Introduction to Multiprocessor Systems

Building multiprocessor systems is an attractive proposition for system vendors for a number of reasons. First of all, they provide a natural, incremental upgrade path for customers with growing computational demands. As long as the key user applications provide thread-level parallelism, adding processors to a system or replacing a smaller system with a larger one that contains more processors provides the customer with a straightforward and efficient way to add computing capacity. Second, multiprocessor systems allow the system vendor to amortize the cost of a single microprocessor design across a wide variety of system design points that provide varying levels of performance and scalability. Finally, multiprocessors that provide coherent shared memory provide a programming model

that is compatible with time-shared uniprocessors, making it easy for customers to deploy existing applications and develop new ones. In these systems, the hardware and operating system software collaborate to provide the user and programmer with the appearance of four *multiprocessor idealisms:*

- *Fully shared memory* means that all processors in the system have equivalent access to all the physical memory in the system.
- *Unit latency* means that all requests to memory are satisfied in a single cycle.
- *Lack of contention* means that the forward progress of one processor's memory references is never slowed down or affected by memory references from another processor.
- *Instantaneous propagation of writes* means that any changes to the memory image made by one processor's write are immediately visible to all other processors in the system.

Naturally, the system and processor designers must strive to approximate these idealisms as closely as possible so as to satisfy the performance and correctness expectations of the user. Obviously, factors such as cost and scalability can play a large role in how easy it is to reach these goals, but a well-designed system can in fact maintain the illusion of these idealisms quite successfully.

11.3.1 Fully Shared Memory, Unit Latency, and Lack of Contention

As shown in Figure 11.3, most conventional shared-memory multiprocessors that provide uniform memory access (UMA) are usually built using a *dancehall* organization, where a set of memory modules or banks is connected to the set of processors

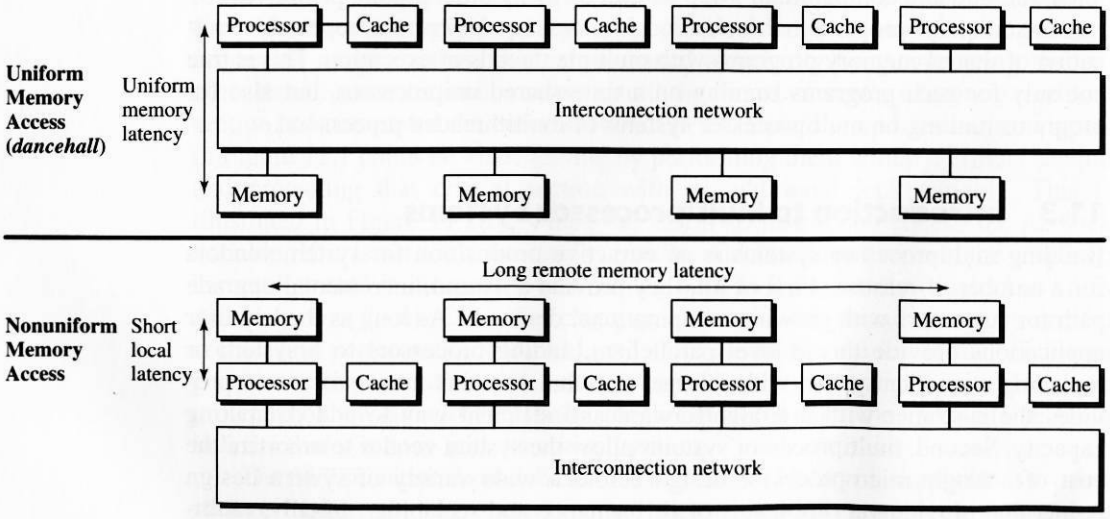


Figure 11.3
UMA versus NUMA Multiprocessor Architecture.

via a crossbar interconnect, and each processor incurs the same uniform latency in accessing a memory bank through this crossbar. The downsides of this approach are the cost of the crossbar, which increases as the square of the number of processors and memory banks, and the fact that every memory reference must traverse this crossbar. As an alternative, many system vendors now build systems with nonuniform memory access (NUMA), where the processors are still connected to each other via a crossbar interconnect, but each processor has a local bank of memory with much lower access latency. In a NUMA configuration, only references to remote memory must pay the latency penalty of traversing the crossbar.

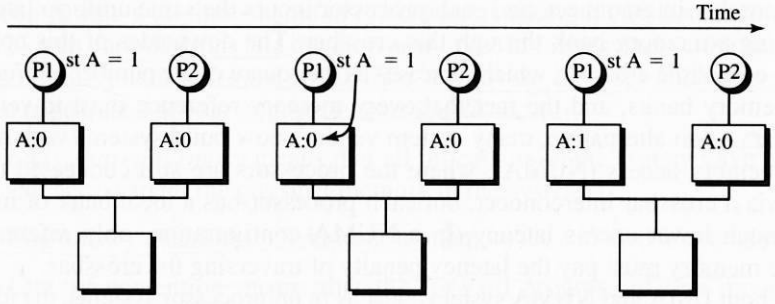
In both UMA and NUMA systems, just as in uniprocessor systems, the idealism of unit latency is approximated with the use of caches that are able to satisfy references to both local and remote (NUMA) memories. Similarly, the traffic filtering effect of caches is used to mitigate contention in the memory banks, as is the use of intelligent memory controllers that combine and reorder requests to minimize latency. Hence, caches, which we have already learned are indispensable in uniprocessor systems, are similarly very effective in multiprocessor systems as well. However, the presence of caches in a multiprocessor system creates additional difficulties when dealing with memory writes, since these must now be somehow made visible to or propagated to other processors in the system.

11.3.2 Instantaneous Propagation of Writes

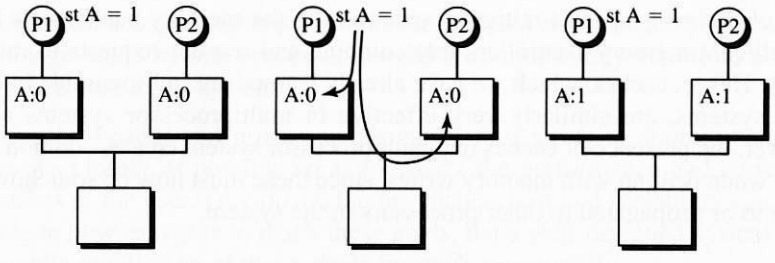
In a time-shared uniprocessor system, if one thread updates a memory location by writing a new value to it, that thread as well as any other thread that eventually executes will instantaneously see the new value, since it will be stored in the cache hierarchy of the uniprocessor. Unfortunately, in a multiprocessor system, this property does not hold, since subsequent references to the same address may now originate from different processors. Since these processors have their own caches that may contain private copies of the same cache line, they may not see the effects of the other processor's write. For example, in Figure 11.4(a), processor P1 writes a "1" to memory location A. With no coherence support, the copy of memory location A in P2's cache is not updated to reflect the new value, and a load at P2 would still observe the stale value of "0." This is known as the classic *cache coherence* problem, and to solve it, the system must provide a cache coherence protocol that ensures that all processors in the system gain visibility to all the other processors' writes, so that each processor has a coherent view of the contents of memory [Censier and Feautrier, 1978]. There are two fundamental approaches to cache coherence—update protocols and invalidate protocols—and these are discussed briefly in Section 11.3.3. These are illustrated in Figure 11.4(b) and (c).

11.3.3 Coherent Shared Memory

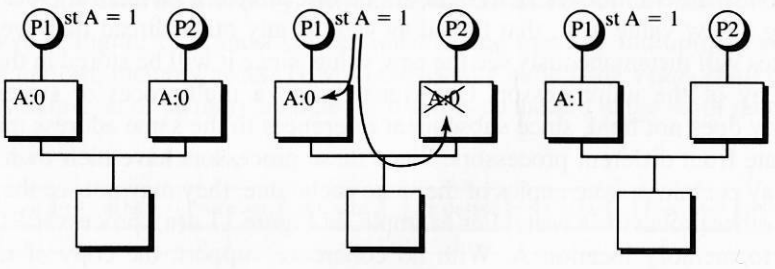
A coherent view of memory is a hard requirement for shared-memory multiprocessors. Without it, programs that share memory would behave in unpredictable ways, since the value returned by a read would vary depending on which processor performed the read. As already stated, the coherence problem is caused by the fact that writes are not automatically and instantaneously propagated to other processors'



(a) No coherence protocol: stale copy of A at P2



(b) Update protocol writes through to both copies of A



(c) Invalidate protocol eliminates stale remote copy

An update protocol updates all remote copies, while an invalidate protocol removes remote copies.

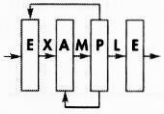


Figure 11.4
Update and Invalidate Protocols.

caches. To ensure that writes are made visible to other processors, two classes of coherence protocols exist.

11.3.3.1 Update Protocols. The earliest proposed multiprocessors employed a straightforward approach to maintaining cache coherence. In these systems, the processors' caches used a write-through policy, in which all writes were performed not just against the cache of the processor performing the write, but also against main memory. Such a protocol is illustrated in Figure 11.4(b). Since all

processors were connected to the same electrically shared bus that also connected them to main memory, all other processors were able to observe the write-throughs as they occurred and were able to directly *update* their own copies of the data (if they had any such copies) by snooping the new values from the shared bus. In effect, these *update protocols* were based on a *broadcast write-through policy;* that is, every write by every processor was written through, not just to main memory, but also to any copy that existed in any other processor's cache. Obviously, such a protocol is not scalable beyond a small number of processors, since the write-through traffic from multiple processors will quickly overwhelm the bandwidth available on the memory bus.

A straightforward optimization allowed the use of writeback caching for private data, where writes are performed locally against the processor's cache and the changes are written back to main memory only when the cache line is evicted from the processor's cache. In such a protocol, however, any writes to shared cache lines (i.e., lines that were present in any other processor's cache) still had to be broadcast on the bus so the sharing processors could update their copies. Furthermore, a remote read to a line that was now dirty in the local cache required the dirty line to be flushed back to memory before the remote read could be satisfied.

Unfortunately, the excessive bandwidth demands of update protocols have led to their virtual extinction, as there are no modern multiprocessor systems that use an update protocol to maintain cache coherence.

11.3.3.2 Invalidate Protocols. Today's modern shared-memory multiprocessors all use invalidate protocols to maintain coherence. The fundamental premise of an invalidate protocol is simple: only a single processor is allowed to write a cache line at any point in time (such protocols are also often called *single-writer* protocols). This policy is enforced by ensuring that a processor that wishes to write to a cache line must first establish that its copy of the cache line is the only valid copy in the system. Any other copies must be invalidated from other processors' caches (hence the term *invalidate protocol*). This protocol is illustrated in Figure 11.4(c). In short, before a processor performs its write, it checks to see if there are any other copies of the line elsewhere in the system. If there are, it sends out messages to invalidate them; finally, it performs the write against its private and exclusive copy. Subsequent writes to the same line are streamlined, since no check for outstanding remote copies is required. Once again, as in uniprocessor writeback caches, the updated line is not written back to memory until it is evicted from the processor's cache. However, the coherence protocol must keep track of the fact that a modified copy of the line exists and must prevent other processors from attempting to read the stale version from memory. Furthermore, it must support flushing the modified data from the processor's cache so that a remote reference can be satisfied by the only up-to-date copy of the line.

Minimally, an invalidate protocol requires the cache directory to maintain at least two states for each cached line: modified (M) and invalid (I). In the invalid state, the requested address is not present and must be fetched from memory. In

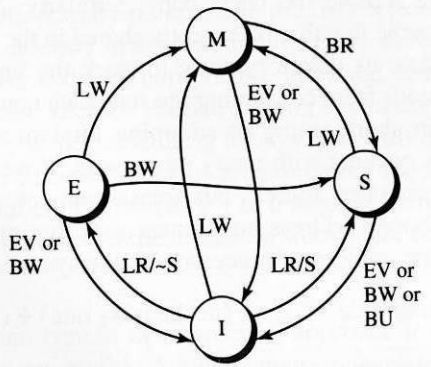
the modified state, the processor knows that there are no other copies in the system (i.e., the local copy is the exclusive one), and hence the processor is able to perform reads and writes against the line. Note that any line that is evicted in the modified state must be written back to main memory, since the processor may have performed a write against it. A simple optimization incorporates a dirty bit in the cache line's state, which allows the processor to differentiate between lines that are exclusive to that processor (usually called the E state) and ones that are exclusive and have been dirtied by a write (usually called the M state). The IBM/Motorola PowerPC G3 processors used in Apple's Macintosh desktop systems implement an MEI coherence protocol.

Note that with these three states (MEI), no cache line is allowed to exist in more than one processor's cache at the same time. To solve this problem, and to allow readable copies of the same line in multiple processors' caches, most invalidate protocols also include a shared state (S). This state indicates that one or more remote readable copies of a line may exist. If a processor wishes to perform a write against a line in the S state, it must first *upgrade* that line to the M state by invalidating the remote copies.

Figure 11.5 shows the state table and transition diagram for a straightforward MESI coherence protocol. Each row corresponds to one of the four states (M, E, S, or I), and each column summarizes the actions the coherence controller must perform in response to each type of bus event. Each transition in the state of a cache line is caused either by a local reference (read or write), a remote reference (bus read, bus write, or bus upgrade), or a local capacity-induced eviction. The cache directory or tag array maintains the MESI state of each line that is in that cache. Note that this allows each cache line to be in a different state at any point in time, enabling lines that contain strictly private data to stay in the E or M state, while lines that contain shared data can simultaneously exist in multiple caches in the S state. The MESI coherence protocol supports the single-writer principle to guarantee coherence but also allows efficient sharing of read-only data as well as silent upgrades from the exclusive (E) state to the modified (M) state on local writes (i.e., no bus upgrade message is required).

A common enhancement to the MESI protocol is achieved by adding an O, or owned state to the protocol, resulting in an MOESI protocol. The O state is entered following a remote read to a dirty block in the M state. The O state signifies that multiple valid copies of the block exist, since the remote requestor has received a valid copy to satisfy the read, while the local processor has also kept a copy. However, it differs from the conventional S state by avoiding the writeback to memory, hence leaving a stale copy in memory. This state is also known as *shared-dirty,* since the block is shared, but is still dirty with respect to memory. An owned block that is evicted from a cache must be written back, just like a dirty block in the M state, since the copy in main memory must be made up-to-date. A system that implements the O state can place either the requesting processor or the processor that supplies the dirty data in the O state, while placing the other copy in the S state, since only a single copy needs to be marked dirty.

	Event and Local Coherence Controller Responses and Actions (s′ refers to next state)					
Current State s	**Local Read (LR)**	**Local Write (LW)**	**Local Eviction (EV)**	**Bus Read (BR)**	**Bus Write (BW)**	**Bus Upgrade (BU)**
Invalid (I)	Issue bus read if no sharers then s′ = E else s′ = S	Issue bus write s′ = M	s′ = I	Do nothing	Do nothing	Do nothing
Shared (S)	Do nothing	Issue bus upgrade s′ = M	s′ = I	Respond shared	s′ = I	s′ = I
Exclusive (E)	Do nothing	s′ = M	s′ = I	Respond shared s′ = S	s′ = I	Error
Modified (M)	Do nothing	Do nothing	Write data back; s′ = I	Respond dirty; Write data back; s′ = S	Respond dirty; Write data back; s′ = I	Error



In response to local and bus events the coherence controller may need to change the local coherence state of a line, and may also need to fetch or supply the cache line data.

Figure 11.5
Sample MESI Cache Coherence Protocol.

11.3.4 Implementing Cache Coherence

Maintaining cache coherence requires a mechanism that tracks the state (e.g., MESI) of each active cache line in the system, so that references to those lines can be handled appropriately. The most convenient place to store the coherence state is in the cache tag array, since state information must be maintained for each line in the cache anyway. However, the local coherence state of a cache line needs to be available to other processors in the system so that their references to the line

coherence state stored in cache tag array.

can be correctly satisfied as well. Hence, a cache coherence implementation must provide a means for distributed access to the coherence state of the lines in its cache. There are two overall approaches for doing so: *snooping* implementations and *directory* implementations.

11.3.4.1 Snooping Implementations. The most straightforward approach for implementing coherence and consistency is via *snooping*. In a snooping implementation, all off-chip address events evoked by the coherence protocol (e.g., cache misses and invalidates in an invalidate protocol) are made visible to all other processors in the system via a shared address bus. In small-scale systems, the address bus is electrically shared and each processor sees all the other processors' commands as they are placed on the bus. More advanced point-to-point interconnect schemes that avoid slow multidrop busses can also support snooping by reflecting all commands to all processors via a hierarchical snoop interconnect. For notational convenience, we will simply refer to any such scheme as an address bus.

In a snooping implementation, the coherence protocol specifies if and how a processor must react to the commands that it observes on the address bus. For example, a remote processor's read to a cache line that is currently modified in the local cache must cause the cache controller to flush the line out of the local cache and transmit it either directly to the requester and/or back to main memory, so the requester will receive the latest copy. Similarly, a remote processor's invalidate request to a cache line that is currently shared in the local cache must cause the controller to update its directory entry to mark the line invalid. This will prevent all future local reads from consuming the stale data now in the cache.

The main shortcoming of snooping implementations of cache coherence is scalability to systems with many processors. If we assume that each processor in the system generates address bus transactions at some rate, we see that the frequency of inbound address bus transactions that must be snooped is directly proportional to the number of processors in the system.

$$\text{Outbound snoop rate} = s_o = (\text{cache miss rate}) + (\text{bus upgrade rate}) \tag{11.1}$$

$$\text{Inbound snoop rate} = s_i = n \times s_o \tag{11.2}$$

That is to say, if each processor generates s_o address transactions per second (consisting of read requests from cache misses and upgrade requests for stores to shared lines), and there are n processors in the system, then each processor must also snoop ns_o transactions per second. Since each snoop minimally requires a local cache directory lookup to check to see if the processor needs to react to the snoop (refer to Figure 11.5 for typical reactions), the aggregate lookup bandwidth required for large n can quickly become prohibitive. Similarly, the available link bandwidth connecting the processor to the rest of the system can be easily overwhelmed by this traffic; in fact, many snoop-based multiprocessors are performance-limited by address-bus bandwidth. Snoop-based implementations have been shown to scale to several dozen processors (up to 64 in the case of the Sun Enterprise 10000 [Charlesworth, 1997]), but scaling up to and beyond that number requires an expensive investment in increased address bus bandwidth.

Large-scale snoop-based systems can also suffer dramatic increases in memory latency when compared to systems designed for fewer processors, since the memory latency will be determined by the latency of the coherence response, rather than the DRAM and data interconnect latency. In other words, for large n, it often takes longer to snoop and collect snoop responses from all the processors in the system than it does to fetch the data from DRAM, even in a NUMA configuration that has long remote memory latencies. Even if the data from memory are transmitted speculatively to the requester, they are not known to be valid until all processors in the system have responded that they do not have a more up-to-date dirty copy of the line. Hence, the snoop response latency often determines how quickly a cache miss can be resolved, rather than the latency to retrieve the cache line itself from any local or remote storage location.

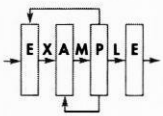
11.3.4.2 Directory Implementation. The most common solution to the scalability and memory latency problems of snooping implementations is to use *directories*. In a directory implementation, coherence is maintained by keeping a copy of a cache line's coherence state collocated with main memory. The coherence state, which is stored in a directory that resides next to main memory, indicates if the line is currently cached anywhere in the system, and also includes pointers to all cached copies in a *sharing list* or *sharing vector*. Sharing lists can be either precise (meaning each sharer is individually indicated in the list) or coarse (meaning that multiple processors share an entry in the list, and the entry indicates that one or more of those processors has a shared copy of the line) and can be stored as linked lists or fixed-size presence vectors. Precise sharing vectors have the drawback of significant storage overhead, particularly for systems with large numbers of processors, since each cache line-size block of main memory requires directory storage proportional to the number of processors. For a large system with 64-byte cache lines and 512 processors, this overhead can be 100% just for the sharing vector.

Bandwidth Scaling. The main benefit of a directory approach is that directory bandwidth scales with memory bandwidth. Adding a memory bank to supply more memory data bandwidth also adds directory bandwidth. Another benefit is that demand for address bandwidth is reduced by filtering commands at the directory. In a directory implementation, address commands are sent to the directory first and are forwarded to remote processors only when necessary (e.g., when the line is dirty in a remote cache or when writing to a line that is shared in a remote cache). Hence, the frequency of inbound address commands to each processor is no longer proportional to the number of processors in the system, but rather it is proportional to the degree of data sharing, since a processor receives an address command only if it owns or has a shared copy of the line in question. Hence, systems with dozens to hundreds of processors can and have been built.

Memory Latency. Finally, latency for misses that are satisfied from memory can be significantly reduced, since the memory bank can respond with nonspeculative data as soon as it has checked the directory. This is particularly advantageous in

a NUMA configuration where the operating and run-time systems have been optimized to place private or nonshared data in a processor's local memory. Since the latency to local memory is usually very low in such a configuration, misses can be resolved in dozens of nanoseconds instead of hundreds of nanoseconds.

Communication Miss Latency. The main drawback of directory-based systems is the additional latency incurred for cache misses that are found dirty in a remote processor's cache (called *communication misses* or *dirty misses*). In a snoop-based system, a dirty miss is satisfied directly, since the read request is transmitted directly to the responder that has the dirty data. In a directory implementation, the request is first sent to the directory and then forwarded to the current owner of the line; this results in an additional traversal of the processor/memory interconnect and increases latency. Applications such as database transaction processing that share data intensively are very sensitive to dirty miss latency and can perform poorly on directory-based systems.



Hybrid snoopy/directory systems have also been proposed and built. For example, the Sequent NUMA-Q system uses conventional bus-based snooping to maintain coherence within four-processor quads, but extends cache coherence across multiple quads with a directory protocol built on the scalable coherent interface (SCI) standard [Lovett and Clapp, 1996]. Hybrid schemes can obtain many of the scalability benefits of directory schemes while still maintaining a low average latency for communication misses that can be satisfied within a local snoop domain.

11.3.5 Multilevel Caches, Inclusion, and Virtual Memory

Most modern processors implement multiple levels of cache to trade off capacity and miss rate against access latency and bandwidth: the level-1 or primary cache is relatively small but allows one- or two-cycle access, frequently through multiple banks or ports, while the level-2 or secondary cache provides much greater capacity but with multicycle access and usually just a single port. The design of multilevel cache hierarchies is an exercise in balancing implementation cost and complexity to achieve the lowest average memory latency for references that both hit and miss the caches. As shown in Equation (11.3), the average memory reference latency lat_{avg} can be computed as the weighted sum of the latencies to each of n levels of the cache hierarchy, where each latency lat_i is weighted by the fraction of references ref_i satisfied by that level:

$$\text{lat}_{\text{avg}} = \sum_{i=1}^{n} \text{ref}_i \times \text{lat}_i \tag{11.3}$$

Of course, such an average latency measure is less meaningful in the context of out-of-order processors, where miss latencies to the secondary cache can often by overlapped with other useful work, reducing the importance of high hit rates in the primary cache. Besides reducing average latency, the other primary objective of primary caches is to reduce the bandwidth required to the secondary cache. Since

the majority of references will be satisfied by a reasonably sized primary cache, only a small subset need to be serviced by the secondary cache, enabling a much narrower and usually single-ported access path to such a cache.

Guaranteeing cache coherence in a design with multiple levels of cache is only incrementally more complex than in the base case of only a single level of cache; some benefit can be obtained by maintaining *inclusion* between levels of the cache by forcing each line that resides in a higher level of cache to also reside in a lower level.

Noninclusive Caches. A straightforward approach to multilevel cache coherence which does not require inclusion treats each cache in the hierarchy as a peer in the coherence scheme, implying that coherence is maintained independently for each level. In a snooping implementation, this implies that all levels of the cache hierarchy must snoop all the address commands traversing the system's address bus. This can lead to excessive bandwidth demands on the level-1 tag array, since both the processor core and the inbound address bus can generate a high rate of references to the tag array. The IBM Northstar/Pulsar design [Storino et al., 1998], which is noninclusive and employs snoop-based coherence, maintains two copies of the level-1 tag array to provide what is effectively dual-ported access to this structure. In a noninclusive directory implementation, the sharing vector must maintain separate entries for each level of each processor (if the sharing vector is precise), or it can revert to a coarse sharing scheme which implies that messages must be forwarded to all levels of cache of the processor that has a copy of the line.

Inclusive Caches. A common alternative to maintaining coherence independently for each level of cache is to guarantee that the coherence state of each line in an upper level of cache is consistent with the lower private levels by maintaining *inclusion*. For example, in a system with two levels of cache, the cache hierarchy must ensure that each line that resides in the level-1 cache also resides in (or is *included* in) the level-2 cache in a consistent state. Maintaining inclusion is fairly straightforward: Whenever a line enters the level-1 cache, it must also be placed in the level-2 cache. Similarly, whenever a line leaves the level-2 cache (is evicted due to a replacement or is invalidated), it must also leave the level-1 cache. If inclusion is maintained, only the lower level of the cache hierarchy needs to participate directly in the cache coherence scheme. By definition, any coherence operation that pertains to lines in the level-1 cache also pertains to the corresponding line in the level-2 cache, and the cache hierarchy, upon finding such a line in the level-2 cache, must now apply that operation to the level-1 cache as well. In effect, snoop lookups in the tag array of the level-2 cache serve as a filter to prevent coherence operations that are not relevant from requiring a lookup in the level-1 tag array. In snoop-based implementations with lots of address traffic, this can be a significant advantage, since the tag array references are now mostly partitioned into two disjoint groups: 90% or more of processor core references are satisfied by the level-1 tag array as cache hits, while 90% or more of the address bus commands are satisfied by the level-2 tag array as misses. Only the level-1 misses require a level-2 tag lookup, and only coherence hits to shared lines require accesses to the level-1 tag

array. This approach avoids having to maintain multiple copies of or dual-porting the level-1 tag array.

Cache Coherence and Virtual Memory. Additional complexity is introduced by the fact that nearly all modern processors implement virtual memory to provide access protection and demand paging. With virtual memory, the effective or virtual address generated by a user program is translated to a physical address using a mapping that is maintained by the operating system. Usually, this address translation is performed prior to accessing the cache hierarchy, but, for cycle time and capacity reasons, some processors implement primary caches that are virtually indexed or tagged. The access time for a virtually addressed cache can be lower since the cache can be indexed in parallel with address translation. However, since cache coherence is typically handled using physical addresses and not virtual addresses, performing coherence-induced tag lookups in such a cache poses a challenge. Some mechanism for performing reverse address translation must exist; this can be accomplished with a separate reverse address translation table that keeps track of all referenced real addresses and their corresponding virtual addresses, or—in a multilevel hierarchy—with pointers in the level-2 tag array that point to corresponding level-1 entries. Alternatively, the coherence controller can search all the level-1 entries in the congruence class corresponding to a particular real address. In the case of a large set-associative virtually addressed cache, this alternative can be prohibitively expensive, since the congruence class can be quite large. The interested reader is referred to a classic paper by Wang et al. [1989] on this topic.

11.3.6 Memory Consistency

In addition to providing a coherent view of memory, a multiprocessor system must also provide support for a predefined *memory consistency model*. A consistency model specifies an agreed-upon convention for ordering the memory references of one processor with respect to the references of another processor and is an integral part of the instruction set architecture specification of any multiprocessor-capable system [Lamport, 1979]. Consistent ordering of memory references across processors is important for the correct operation of any multithreaded applications that share memory, since without an architected set of rules for ordering such references, such programs could not correctly and reliably synchronize between threads and behave in a repeatable, predictable manner. For example, Figure 11.6 shows a simple

```
Reorder load before store
        ↑
Proc0                          Proc1

st A=1                         st B=1
if (load B==0) {               if (load A==0) {
   ...critical section            ...critical section
}                              }
```

If either processor reorders the load and executes it before the store, both processors can enter the mutually exclusive critical section simultaneously.

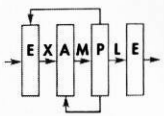
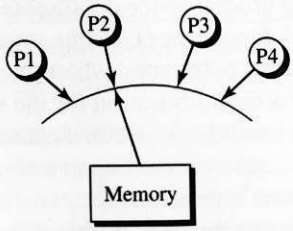


Figure 11.6
Dekker's Algorithm for Mutual Exclusion.

serialization scheme that guarantees mutually exclusive access to a critical section, which may be updating a shared datum. Dekker's mutual exclusion scheme for two processors consists of processor 0 setting a variable A, testing another variable B, and then performing the mutually exclusive access (the variable names are reversed for processor 1). As long as each processor sets its variable before it tests the other processor's variable, mutual exclusion is guaranteed. However, without a consistent ordering between the memory references performed here, two processors could easily get confused about whether the other has entered the critical section. Imagine a scenario in which both tested each other's variables at the same time, but neither had yet observed the other's write, so both entered the critical section, continuing with conflicting updates to some shared object. Such a scenario is possible if the processors are allowed to reorder memory references so that loads execute before independent stores (termed *load bypassing* in Chapter 5).

11.3.6.1 Sequential Consistency. The simplest consistency model is called sequential consistency, and it requires imposing a total order among all references being performed by all processors [Lamport, 1979]. Conceptually, a sequentially consistent (SC) system behaves as if all processors take turns accessing the shared memory, creating an interleaved, totally ordered stream of references that also obeys program order for each individual processor. This approach is illustrated in Figure 11.7 and is in principle similar to the interleaving of references from multiple threads executing on a single time-shared processor. Because of this similarity, it is easier for programmers to reason about the behavior of shared-memory programs on SC systems, since multithreaded programs that operate correctly on time-shared uniprocessors will also usually operate correctly on a sequentially consistent multiprocessor.

However, sequential consistency is challenging to implement efficiently. Consider that imposing a total order requires not only that each load and store must issue in program order, effectively crippling a modern out-of-order processor, but that each reference must also be ordered with respect to all other processors in the system, naively requiring a very-high-bandwidth interconnect for establishing



Each processor accesses memory in program order, and accesses from all processors are interleaved as if memory serviced requests from only one processor at a time.

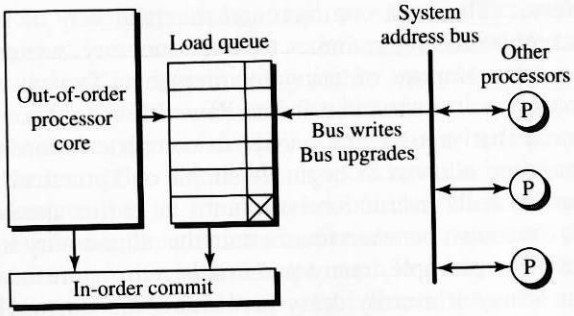
Figure 11.7
Sequentially Consistent Memory Reference Ordering.
Source: Lamport, 1979.

this global order. Fortunately, the same principle that allows us to relax instruction ordering within an out-of-order processor also allows us to relax the requirement for creating a total memory reference order. Namely, just as sequential execution of the instruction stream is an overspecification and is not strictly required for correctness, SC total order is also overly rigorous and not strictly necessary. In an out-of-order processor, register renaming and the reorder buffer enable relaxed execution order, gated only by true data dependences, while maintaining the illusion of sequential execution. Similarly, SC total order can be relaxed so that only those references that must be ordered to enforce data dependences are in fact ordered, while others can proceed out of order. This allows programs that expect SC total order to still run correctly, since the failures can only occur when the reference order of one processor is exposed to another via data dependences expressed as accesses to shared locations.

11.3.6.2 High-Performance Implementation of Sequential Consistency. There are a number of factors that enable efficient implementation of this relaxation of SC total order. The first is the presence of caches and a cache coherence mechanism. Since cache coherence guarantees each processor visibility to other processors' writes, it also reveals to us any interprocessor data dependences; namely, if a read on one processor is data-dependent on a write from another processor [i.e., there is a read-after-write (RAW) dependence], the coherence mechanism must intervene to satisfy that dependence by first invalidating the address in question from the reader's cache (to guarantee single-writer coherence) and then, upon the subsequent read that misses the invalidated line, supplying the updated line by flushing it from the writer's cache and transmitting it to the reader's cache.

Conveniently, in the absence of such coherence activity (invalidates and/or cache misses), we know that no dependence exists. Since the vast majority of memory references are satisfied with cache hits which require no such intervention, we can safely relax the reference order between cache hits. This decomposes the problem of reference ordering into a local problem (ordering local references with respect to boundaries formed by cache misses and remote invalidate requests) and a global problem (ordering cache misses and invalidate requests). The former is accomplished by augmenting a processor's load/store queue to monitor global address events in addition to processor-local addresses, which it monitors anyway to enforce local store-to-load dependences. Cache misses and upgrades are ordered by providing a global ordering point somewhere in the system. For small-scale systems with a shared address bus, arbitration for the single shared bus establishes a global order for misses and invalidates, which must traverse this bus. In a directory implementation, commands can be ordered upon arrival at the directory or at some other shared point in the system's interconnect.

However, we must still solve the local problem by ordering all references with respect to coherence events. Naively, this requires that we must ensure that all prior memory references result in a cache hit before we can perform the current reference. Clearly, this degenerates into in-order execution of all memory references and precludes high-performance out-of-order execution. Here, we can apply speculation to solve this problem and enable relaxation of this ordering requirement.



Loads issue out of order, but loaded addresses are tracked in the load queue. Any remote stores that occur before the loads retire are snooped against the load queue. Address matches indicate a potential ordering violation and trigger refetch-based recovery when the load attempts to commit.

Figure 11.8
Read Set Tracking to Detect Consistency Violations.

Namely, we can speculate that a particular reference in fact need not be ordered, execute it speculatively, and recover from that speculation only in those cases where we determine that it needed to be ordered. Since a canonical out-of-order processor already supports speculation and recovery, we need only to add a mechanism that detects ordering violations and initiates recovery in those cases.

The most straightforward approach for detecting ordering violations is to monitor global address events and check to see if they conflict with local speculatively executed memory references. Since speculatively executed memory references are already tracked in the processor's load/store queue, a simple mechanism that checks global address events (invalidate messages that correspond to remote writes) against all unretired loads is sufficient. As shown in Figure 11.8, a matching address causes the load to be marked for a potential ordering violation. As instructions are retired in program order at completion time, they are checked for ordering violations. If the processor attempts to retire such a load, the processor treats the load as if it were a branch misprediction and refetches the load and all subsequent instructions. Upon re-execution, the load is now ordered after the conflicting remote write. A mechanism similar to this one for guaranteeing adherence to the memory consistency model is implemented in the MIPS R10000 [Yeager, 1996], HP PA-8000, and Intel Pentium Pro processors and their later derivatives.

11.3.6.3 Relaxed Consistency Models. An architectural alternative to sequential consistency is to specify a more relaxed consistency model to the programmer. A broad variety of *relaxed consistency* (RC) models have been proposed and implemented, with various subtle differences. The interested reader is referred to Adve and Gharachorloo's [1996] consistency model tutorial for a detailed discussion of several relaxed models. The underlying motivation for RC models is to simplify implementation of the hardware by requiring the programmer to identify and label those references that need to be ordered, while allowing the hardware to proceed with unordered execution of all unlabeled references.

Memory Barriers. The most common and practical way of labeling ordered references is to require the programmer to insert memory barrier instructions or fences in the code to impose ordering requirements. Typical memory barrier semantics (e.g., the `sync` instruction in the PowerPC instruction set) require all memory references that precede the barrier to complete before any subsequent memory references are allowed to begin. A simple and practical implementation of a memory barrier stalls instruction issue until all earlier memory instructions have completed. Care must be taken to ascertain that all memory instructions have in fact completed; for example, many processors retire store instructions into a store queue, which may arbitrarily delay performing the stores. Hence, checking that the reorder buffer does not contain stores is not sufficient; checking must be extended to the queue of retired stores. Furthermore, invalidate messages corresponding to a store may still be in flight in the coherence interconnect, or may even be delayed in an invalidate queue on a remote processor chip, even though the store has already been performed against the local cache and removed from the store queue. For correctness, the system has to guarantee that all invalidates originating from stores preceding a memory barrier have actually been applied, hence preventing any remote accesses to stale copies of the line, before references following the memory barrier are allowed to issue. Needless to say, this can take a very long time, even into hundreds of processor cycles for systems with large numbers of processors.

The main drawback of relaxed models is the additional burden placed on the programmer to identify and label references that need to be ordered. Reasoning about the correctness of multithreaded programs is a difficult challenge to begin with; imposing subtle and sometimes counterintuitive correctness rules on the programmer can only hurt programmer productivity and increase the likelihood of subtle errors and problematic race conditions.

Benefits of Relaxed Consistency. The main advantage of relaxed models is better performance with simpler hardware. This advantage can disappear if memory barriers are frequent enough to require implementations that are more efficient than simply stalling issue and waiting for all pending memory references to complete. A more efficient implementation of memory barriers can look very much like the invalidation tracking scheme illustrated in Figure 11.8; all load addresses are snooped against invalidate messages, but a violation is triggered only if a memory barrier is retired before the violating load is retired. This can be accomplished by marking a load in the load/store queue twice: first, when a conflicting invalidate occurs, and second, when a local memory barrier is retired and the first mark is already present. When the load attempts to retire, a refetch is triggered only if both marks are present, indicating that the load may have retrieved a stale value from the data cache.

The fundamental advantage of relaxed models is that in the absence of memory barriers, the hardware has greater freedom to overlap the latency of store misses with the execution of subsequent instructions. In the SC execution scheme outlined in Section 11.3.6.2, such overlap is limited by the size of the out-of-order

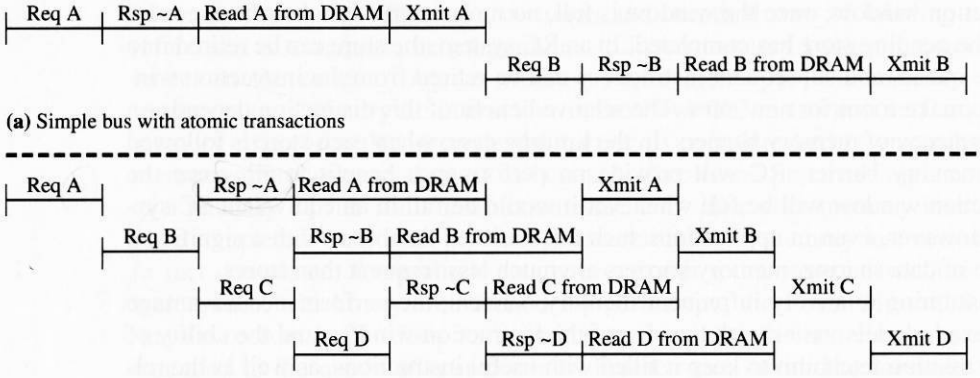
instruction window; once the window is full, no more instructions can be executed until the pending store has completed. In an RC system, the store can be retired into a store queue, and subsequent instructions can be retired from the instruction window to make room for new ones. The relative benefit of this distinction depends on the frequency of memory barriers. In the limiting case, when each store is followed by a memory barrier, RC will provide no performance benefit at all, since the instruction window will be full whenever it would be full in an equivalent SC system. However, even in applications such as relational databases with a significant degree of data sharing, memory barriers are much less frequent than stores.

Assuming relatively infrequent memory barriers, the performance advantage of relaxed models varies with the size of the instruction window and the ability of the instruction fetch unit to keep it filled with useful instructions, as well as the relative latency of retiring a store instruction. Recent trends indicate that the former is growing with better branch predictors and larger reorder buffers, but the latter is also increasing due to increased clock frequency and systems with many processors interconnected with multistage networks. Given what we know, it is not clear if the fundamental advantage of RC systems will translate into a significant performance advantage in the future. In fact, researchers have recently argued against relaxed models, due to the difficulty of reasoning about their correctness [Hill, 1998]. Nevertheless, all recently introduced instruction sets specify relaxed consistency (Alpha, PowerPC, IA-64) and serve as existence proofs that the relative difficulty of reasoning about program correctness with relaxed consistency is by no means an insurmountable problem for the programming community.

11.3.7 The Coherent Memory Interface

A simple uniprocessor interfaces to memory via a bus that allows the processor to issue read and write commands as single atomic bus transactions. With a simple bus, once a processor has successfully arbitrated for the bus, it places the appropriate command on the bus, and then holds the bus until it receives all data and address responses, signaling completion of the transaction. More advanced uniprocessors add support for *split transactions*, where requests and responses are separated to expose greater concurrency and allow better utilization of the bus.

On a split-transaction bus, the processor issues a request and then releases the bus before it receives a data response from the memory controller, so that subsequent requests can be issued and overlapped with the response latency. Furthermore, requests can be split from coherence responses as well, by releasing the address bus before the coherence responses have returned. Figure 11.9 illustrates the benefits of a split-transaction bus. In Figure 11.9(a), a simple bus serializes the request, snoop response, DRAM fetch, and data transmission latencies for two requests, one to address A and one to address B. Figure 11.9(b) shows how a split-transaction bus that releases the bus after every request, and receives snoop responses and data responses on separate busses, can satisfy four requests in a pipelined fashion in less time than the simple bus can satisfy two requests. Of course, the design is significantly more complex, since multiple concurrent split transactions are in flight and have to be tracked by the coherence controller.

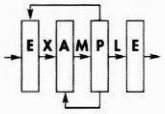


(a) Simple bus with atomic transactions

(b) Split-transaction bus with separate requests and responses

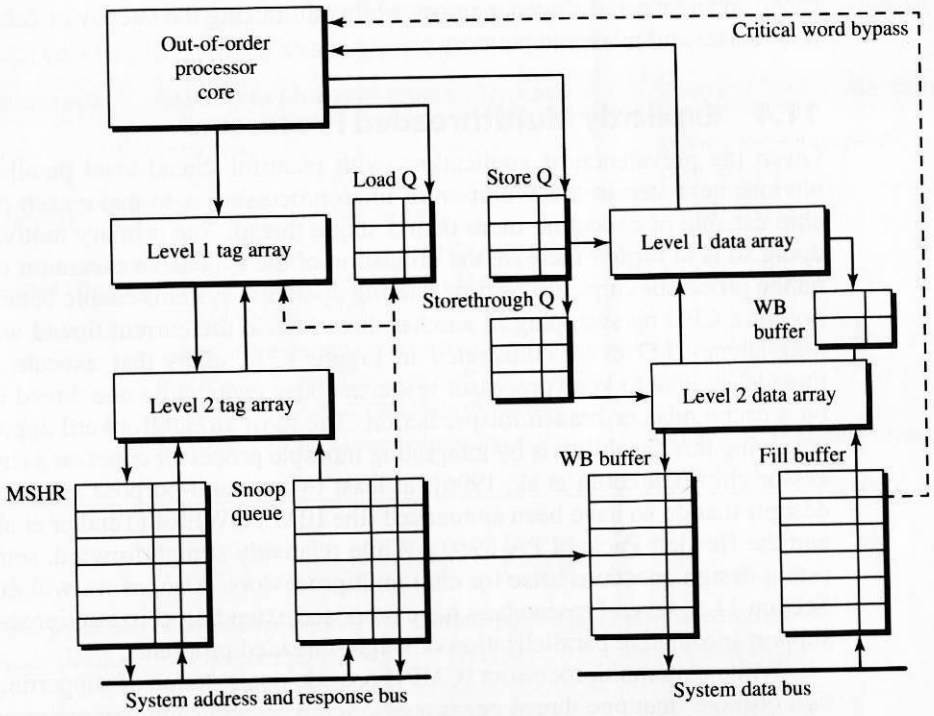
A split-transaction bus enables higher throughput by pipelining requests, responses, and data transmission.

Figure 11.9
Simple Versus Split-Transaction Busses.



Usually, a tag that is unique systemwide is associated with each outstanding transaction; this tag, which is significantly shorter than the physical address, is used to identify subsequent coherence and data messages by providing additional signal lines or message headers on the data and response busses. Each outstanding transaction is tracked with a miss-status handling register (MSHR), which keeps track of the miss address, critical word information, and rename register information that are needed to restart execution once the memory controller returns the data needed by the missing reference. MSHRs are also used to merge multiple requests to the same line to prevent transmitting the same request multiple times. In addition, writeback buffers are used to delay writing back evicted dirty lines from the cache until after the corresponding demand miss has been satisfied; and fill buffers are used to collect a packetized data response into a whole cache line, which is then written into the cache. An example of an advanced split-transaction bus interface is shown in Figure 11.10.

This relatively simple uniprocessor interface must be augmented in several ways to handle coherence in a multiprocessor system. First of all, the bus arbitration mechanism will have to be enhanced to support multiple requesters or bus masters. Second, there must be support for handling inbound address commands that originate at other processors in the system. In a snooping implementation, these are all the commands placed on the bus by other processors, while in a directory implementation these are commands forwarded from the directory. Minimally, this command set must provide functionality for probing the processor's tag array to check the current state of a line, for flushing modified data from the cache, and for invalidating a line. While earlier microprocessor designs required external board-level coherence controllers that issued such low-level commands to the processor's cache, virtually all modern processors support *glueless multiprocessing* by integrating the coherence controller directly on the processor chip. This on-chip



A processor may communicate with memory through two levels of cache, a load queue, store queue, store-through queue (needed if L1 is write-through), MSHR (miss-status handling registers), snoop queue, fill buffers, and write-back buffers. Not shown is the complex control logic that coordinates all this activity.

Figure 11.10
Processor-Memory Interface.

coherence controller reacts to higher-level commands observed on the address bus (e.g., remote read, read exclusive, or invalidate), and then issues the appropriate low-level commands to the local cache. To expose as much concurrency as possible, modern processors implement snoop queues (see Figure 11.10) that accept snoop commands from the bus and then process their semantics in a pipelined fashion.

11.3.8 Concluding Remarks

Systems that integrate multiple processors and provide a coherent and consistent view of memory have enjoyed tremendous success in the marketplace. They provide obvious advantages to system vendors and customers by enabling scalable, high-performance systems that are straightforward to use and write programs for and provide a growth path from entry-level to enterprise-class systems. The abundance of thread-level parallelism in many important applications is a key enabler for such systems. As the demand for performance and scalability continues to grow, designers of such systems are faced with a myriad of tradeoffs for implementing

cache coherence and shared memory while minimizing the latency of communication misses and misses to memory.

11.4 Explicitly Multithreaded Processors

Given the prevalence of applications with plentiful thread-level parallelism, an obvious next step in the evolution of microprocessors is to make each processor chip capable of executing more than a single thread. The primary motivation for doing so is to further increase the utilization of the expensive execution resources on the processor chip. Just as time-sharing operating systems enable better utilization of a CPU by swapping in another thread while the current thread waits on a long-latency I/O event (illustrated in Figure 3.31), chips that execute multiple threads are able to keep processor resources busy even while one thread is stalled on a cache miss or branch misprediction. The most straightforward approach for achieving this capability is by integrating multiple processor cores on a single processor chip [Olukotun et al., 1996]; at least two general-purpose microprocessor designs that do so have been announced (the IBM POWER4 [Tendler et al., 2001] and the Hewlett-Packard PA-8900). While relatively straightforward, some interesting design questions arise for chip multiprocessors. Also, as we will discuss in Section 11.5, several researchers have proposed extending chip multiprocessors to support speculative parallelization of single-threaded programs.

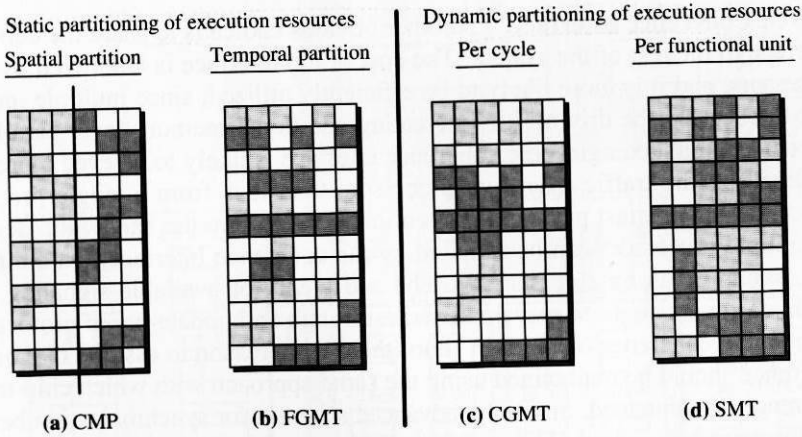
While chip multiprocessors (CMPs) provide one extreme of supporting execution of more than one thread per processor chip by replicating an entire processor core for each thread, other less costly alternatives exist as well. Various approaches to multithreading a single processor core have been proposed and even realized in commercial products. These range from fine-grained multithreading (FGMT), which interleaves the execution of multiple threads on a single execution core on a cycle-by-cycle basis; coarse-grained multithreading (CGMT), which also interleaves multiple threads, but on coarser boundaries delimited by long-latency events like cache misses; and simultaneous multithreading (SMT), which eliminates context switching between multiple threads by allowing instructions from multiple simultaneously active threads to occupy a processor's execution window. Table 11.2 summarizes the context switch mechanism and degree of resource sharing for several approaches to on-chip multithreading. The assignment of execution resources for each of these schemes is illustrated in Figure 11.11.

11.4.1 Chip Multiprocessors

Historically, improvements in transistor density have made it possible to incorporate increasingly complex and area-intensive architectural features such as out-of-order execution, highly accurate branch predictors, and even sizable secondary caches directly onto a processor chip. Recent designs have also integrated coherence controllers to enable glueless multiprocessing, tag arrays for large off-chip cache memories, as well as memory controllers for direct connection of DRAM. System-on-a-chip designs further integrate graphics controllers, other I/O devices, and I/O bus interfaces directly on chip. An obvious next step, as transistor dimensions

Table 11.2
Various approaches to resource sharing and context switching

MT Approach	Resources Shared between Threads	Context Switch Mechanism
None	Everything	Explicit operating system context switch
Fine-grained	Everything but register file and control logic/state	Switch every cycle
Coarse-grained	Everything but I-fetch buffers, register file, and control logic/state	Switch on pipeline stall
SMT	Everything but instruction fetch buffers, return address stack, architected register file, control logic/state, reorder buffer, store queue, etc.	All contexts concurrently active; no switching
CMP	Secondary cache, system interconnect	All contexts concurrently active; no switching



Four possible alternatives are: chip multiprocessing (a), which statically partitions execution bandwidth; fine-grained multiprocessing (b), which executes a different thread in alternate cycles; coarse-grained multithreading (c), which switches threads to tolerate long-latency events; and simultaneous multithreading (d), which intermingles instructions from multiple threads. The CMP and FGMT approaches partition execution resources statically, either with a spatial partition by assigning a fixed number of resources to each processor, or with a temporal partition that time-multiplexes multiple threads onto the same set of resources. The CGMT and SMT approaches allow dynamic partitioning, with either a per-cycle temporal partition in the CGMT approach, or a per-functional unit partition in the SMT approach. The greatest flexibility and highest resource utilization and instruction throughput are achieved by the SMT approach.

Figure 11.11
Running Multiple Threads on One Chip.
Source: Tullsen et al., 1996.

continue to shrink, is to incorporate multiple processor cores onto the same piece of silicon. Chip multiprocessors provide several obvious advantages to system designers: Integrating multiple processor cores on a single chip eases the physical challenges of packaging and interconnecting multiple processors; tight integration reduces off-chip signaling and results in reduced latencies for processor-to-processor communication and synchronization; and finally, chip-scale integration provides interesting opportunities for rethinking and perhaps sharing elements of the cache hierarchy and coherence interface [Olukotun et al., 1996].

Shared Caches. One obvious design choice for CMPs is to share the on- or off-chip cache memory between multiple cores (both the IBM POWER4 and HP PA-8800 do so). This approach reduces the latency of communication misses between the on-chip processors, since no off-chip signaling is needed to resolve such misses. Of course, sharing misses to remote processors are still a problem, although their frequency should be reduced. Unfortunately, it is also true that if the processors are executing unrelated threads that do not share data, a shared cache can be overwhelmed by conflict misses. The operating system's task scheduler can mitigate conflicts and reduce off-chip sharing misses by scheduling for *processor affinity;* that is, scheduling the same and related tasks on processors sharing a cache.

Shared Coherence Interface. Another obvious choice is to share the coherence interface to the rest of the system. The cost of the interface is amortized over two processors, and it is more likely to be efficiently utilized, since multiple independent threads will be driving it and creating additional memory-level parallelism. Of course, an underengineered coherence interface is likely to be even more overwhelmed by the traffic from two processors than it is from a single processor. Hence, designers must pay careful attention to make sure the bandwidth demands of multiple processors can be satisfied by the coherence interface. On a different tack, assuming an on-chip shared cache and plenty of available signaling bandwidth, designers ought to reevaluate write-through and update-based protocols for maintaining coherence on chip. In short, there is no reason to assume that on-chip coherence should be maintained using the same approach with which chip-to-chip coherence is maintained. Similarly, advanced schemes for synchronization between on-chip processors should be investigated.

CMP Drawbacks. However, CMP designs have some drawbacks as well. First of all, one can always argue that given equivalent silicon technology, one can always build a uniprocessor that executes a single thread faster than a CMP of the same cost, since the available die area can be dedicated to better branch prediction, larger caches, or more execution resources. Furthermore, the area cost of multiple cores can easily lead to a very large die that may cause yield or manufacturability issues, particularly when it comes to speed-binning parts for high frequency; empirical evidence suggests that the CMP part, even though designed for the same nominal target frequency, may suffer from a yield-induced frequency disadvantage. Finally, many argue that operating system and software scalability constraints place a ceiling on the total number of processors in a system that is well

below the one imposed by packaging and other physical constraints. Hence, one might conclude that CMP is left as a niche approach that may make sense from a cost/performance perspective for a subset of a system vendor's product range, but offers no fundamental advantage at the high end or low end. Nevertheless, several system vendors have announced CMP designs, and they do offer some compelling advantages, particularly in the commercial server market where applications contain plenty of thread-level parallelism.

IBM POWER4. Figure 11.12 illustrates the IBM POWER4 chip multiprocessor. Each processor chip contains two deeply pipelined out-of-order processor cores, each with a private 64K-byte level-1 instruction cache and a private 32K-byte data cache. The level-1 data caches are write-through; writes from both processors are collected and combined in store queues within each bank of the shared level-2 cache (shown as P0 STQ and P1 STQ). The store queues have four 64-byte entries that allow arbitrary write combining. Each of the three level-2 banks is approximately 512K bytes in size and contains multiple MSHRs for tracking outstanding transactions, multiple

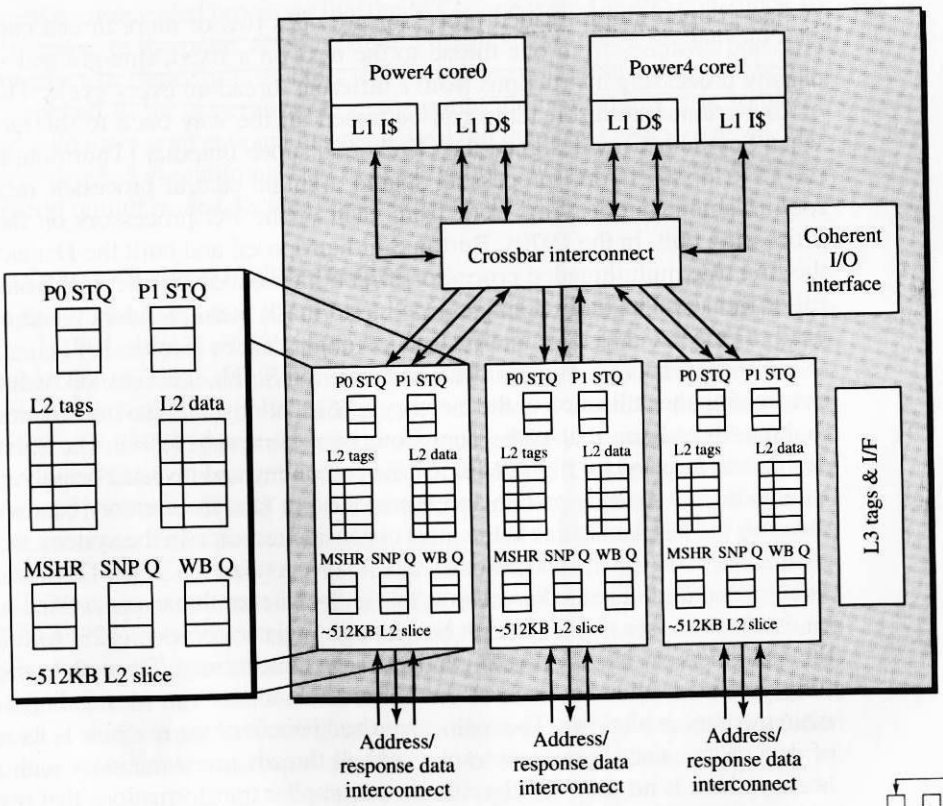


Figure 11.12
IBM POWER4: Example Chip Multiprocessor.
Source: Tendler et al., 2001.

writeback buffers, and multiple snoop queue entries for handling incoming coherence requests. The processors also share the coherence interface to the other processors in the system, a separate interface to the coherent I/O subsystem, as well as the interface to the off-chip level-3 cache and its on-chip tag array. Because of the store-through policy for the level-1 data caches, all coherence requests from remote processors as well as reads from the other on-chip core can be satisfied from the level-2 cache. The level-2 tag array maintains a sharing vector for the two on-chip processors that records which of the two cores contains a shared copy of any cache line in the inclusive level-2 cache. This sharing vector is referenced whenever one of the local cores or a remote processor issues a write to a shared line; an invalidate message is forwarded to one or both of the local cores to guarantee single-writer cache coherence. The POWER4 design supplies tremendous bandwidth (in excess of 100 Gbytes/s) from the level-2 to the processor cores, and also provides multiple high-bandwidth interfaces (each in excess of 10 Gbytes/s) to the level-3 cache and to surrounding processor chips in a multiprocessor configuration.

11.4.2 Fine-Grained Multithreading

A fine-grained multithreaded processor provides two or more thread contexts on chip and switches from one thread to the next on a fixed, fine-grained schedule, usually processing instructions from a different thread on every cycle. The origins of fine-grained multithreading can be traced all the way back to the mid-1960s, when Seymour Cray designed the CDC-6600 supercomputer [Thornton, 1964]. In the CDC-6600, 10 I/O processors shared a single central processor in a round-robin fashion, interleaving work from each of the I/O processors on the central processing unit. In the 1970s, Burton Smith proposed and built the Denelcor HEP, the first true multithreaded processor, which interleaved instructions from a handful of thread contexts in a single pipeline to mask memory latency and avoid the need to detect and resolve interinstruction dependences [Smith, 1991].

A more recent yet similar machine by Burton Smith, the Tera MTA, focused on maximizing the utilization of the memory access path by interleaving references from multiple threads on that path [Tera Computer Company, 1998]. The recent MTA design was targeted for high-end scientific computing and invested heavily in a high-bandwidth, low-latency path to access memory. In fact, the memory bandwidth provided by the MTA machine is the most expensive resource in the system; hence, it is reasonable to design the processor to maximize its utilization. The MTA machine is a fine-grained multithreaded processor; that is, it switches threads on a fixed schedule, on every processor clock cycle. It has enough register contexts (128) to fully mask the main memory latency, making a data cache unnecessary. The path to memory is fully pipelined, allowing each of the 128 threads to have an outstanding access to main memory at all times. The main advertised benefit of the machine is its very lack of data cache; since there is no cache, and all threads access memory with uniform latency, there is no need for algorithmic or compiler transformations that restructure access patterns to maximize utilization of a data cache hierarchy. Instead, the compiler concentrates on identifying independent threads of computation (e.g., do-across loops in scientific programs) to schedule into each of the 128 contexts. While some early performance success has been reported for the Tera MTA machine, its future is

currently uncertain due to delays in its second-generation CMOS implementation (the first generation used an exotic gallium arsenide technology).

Single-Thread Performance. The main drawback of fine-grained multithreaded processors like the Tera MTA is that they sacrifice single-thread performance for overall throughput. Since each memory reference takes 128 cycles to complete, the latency to complete the execution of a single thread on the MTA can be longer by a factor of more than 100 when compared to a conventional cache-based design, where the majority of references are satisfied from cache in a few cycles. Of course, for programs with poor cache locality, the MTA will perform no worse than a cache-based system with similar memory latency but will achieve much higher throughput for the entire set of threads. Unfortunately, there are many applications where single-thread performance is very important. For example, most commercial workloads restrict access to shared data by limiting shared references to critical sections protected by locks. To maintain high throughput for software systems with frequent sharing (e.g., relational database systems), it is very important to execute those critical sections as quickly as possible to reduce the occurrence of lock contention. In a fine-grained multithreaded processor like the MTA, one would expect contention for locks to increase to the point where system throughput would be dramatically and adversely affected. Hence, it is unlikely that fine-grained multithreading will be successfully applied in the general-purpose computing domain unless it is somehow combined with more conventional means of masking memory latency (e.g., caches). However, fine-grained multithreading of specific pipe stages can play an important role in hybrid multithreaded designs, as we will see in Section 11.4.4.

11.4.3 Coarse-Grained Multithreading

Coarse-grained multithreading (CGMT) is an intermediate approach to multithreading that enjoys many of the benefits of the fine-grained approach without imposing severe limits on single-thread performance. CGMT, first proposed at the Massachusetts Institute of Technology and incorporated in several research machines there [Agarwal et al., 1990; Fillo et al., 1995], was successfully commercialized in the Northstar and Pulsar PowerPC processors from IBM [Eickemeyer et al., 1996; Storino et al., 1998]. A CGMT processor also provides multiple thread contexts within the processor core, but differs from fine-grained multithreading by switching contexts only when the currently active thread stalls on a long-latency event, such as a cache miss. This approach makes the most sense on an in-order processor that would normally stall the pipeline on a cache miss. Rather than stall, the pipeline is filled with ready instructions from an alternate thread, until, in turn, one of those threads also misses the cache. In this manner, the execution of two or more thread contexts is interleaved in the processor, resulting in better utilization of the processor's execution resources and effectively masking a large fraction of cache miss latency.

Thread-Switch Penalty. One key design issue in a CGMT processor is the cost of performing a context switch between threads. Since context switches occur in response to dynamic events such as cache misses, which may not be detected until late in the pipeline, a naive context-switch implementation will incur several penalty

cycles. Since instructions following the missing instruction may already be in the pipeline, they need to be drained from the pipeline. Similarly, instructions from the new thread will not reach the execution stage until they have traversed the earlier pipeline stages. Depending on the length of the pipeline, this results in one or more pipeline bubbles. A straightforward approach for avoiding a thread-switch penalty is to replicate the processor's pipeline registers for each thread and to save the current state of the pipeline at each context switch. Hence, an alternate thread context can be switched back in the very next cycle, avoiding any pipeline bubbles (a similar approach was employed in the Motorola 88000 processor to reduce interrupt latency). Of course, the area and complexity cost of shadowing all the pipeline state is considerable. With a fairly short pipeline and a context-switch penalty of only three cycles, the IBM Northstar/Pulsar designers found that such complexity was not merited; eliminating the three-cycle switch penalty provided only marginal performance benefit. This is reasonable, since the switches are triggered to cover the latency of cache misses that can take a hundred or more processor cycles to resolve; saving a few cycles out of hundreds does not translate into a worthwhile performance gain. Of course, a design with a longer pipeline and a larger switch penalty could face a very different tradeoff and may need to shadow pipeline registers or mitigate switch penalty in some other fashion.

Guaranteeing Fairness. One of the challenges of building a CGMT processor is to provide some guarantee of fairness in the allocation of execution resources to prevent starvation from occurring. As long as each thread has comparable cache miss rates, the processor pipeline will be shared fairly among the thread contexts, since each thread will surrender the CPU to an alternate thread at a comparable rate. However, the cache miss rate of a thread is not a property that is easily controlled by the programmer or operating system; hence, additional features are needed to provide fairness and avoid starvation. Standard techniques from operating system scheduling policies can be adopted: Threads with low miss rates can be preempted after a time slice expires, forcing a thread switch; and the hardware can enforce a minimum quantum to avoid starvation caused by premature preemption.

Beyond guaranteeing fairness, a CGMT processor should provide a scheme for minimizing useless execution bandwidth and also for maximizing execution bandwidth for situations where single-thread throughput is critical for performance. The former can occur whenever a thread is in a busy-wait state (e.g., spinning on a lock held by some other thread or processor) or when a thread enters the operating system idle loop. Clearly, in both these cases, all available execution resources should be dedicated to an alternate thread that has useful work, instead of expending them on a busy-wait or idle loop. The latter can occur whenever a thread is holding a critical resource (e.g., a highly contested lock) and there are other threads in the system waiting for that resource to be released. In such a scenario, the execution of the high-priority thread should not be preempted, even if it is stalled on a cache miss, since the alternate threads may slow down the primary thread either directly (due to thread-switch penalty overhead) or indirectly (by causing additional conflict misses or contention in the memory hierarchy).

Thread Priorities. A CGMT processor can avoid these pitfalls of performance by architecting a priority scheme that assigns at least three levels of priority—high, medium, and low—to the active threads. Note that these are not priorities in the operating system sense, where a thread or process has a fixed priority set by the operating system or system administrator. Rather, these thread priorities vary dynamically and reflect the relative importance of execution of the current execution phase of the thread. Hence, programmer intervention is required to notify the hardware whenever a thread undergoes a priority transition. For example, when a thread enters a critical section after acquiring a lock, it should transition to high priority; conversely, when it exits, it should reduce its priority level. Similarly, when a thread enters the idle loop or begins to spin on a lock that is currently held by another thread, it should lower its priority. Of course, such communication requires that an interface be specified, usually through special instructions in the ISA that identify these phase transitions, and also requires programmers to place these instructions in the appropriate locations in their programs. Alternatively, implicit pattern-matching mechanisms that recognize execution sequences that usually accompany these transitions can also be devised. The former approach was employed by the IBM Northstar/Pulsar processors, where specially encoded NOP instructions are used to indicate thread priority level. Fortunately, the required software changes are concentrated in a relatively few locations in the operating system and middleware (e.g., database) and have been realized with minimal effort.

Thread-Switch State Machine. Figure 11.13 illustrates a simple thread-switch state machine for a CGMT processor. As shown, there are four possible states for each processor thread: running, ready, stalled, and swapped. Threads transition between states whenever a cache miss is initiated or completed, and when the thread switch logic decides to switch to an alternate thread. In a well-designed CGMT processor, the following conditions can cause a thread switch to occur:

- A cache miss has occurred in the primary thread, and there is an alternate thread in the ready state.
- The primary thread has entered the idle loop, and there is an alternate nonidle thread in the ready state.

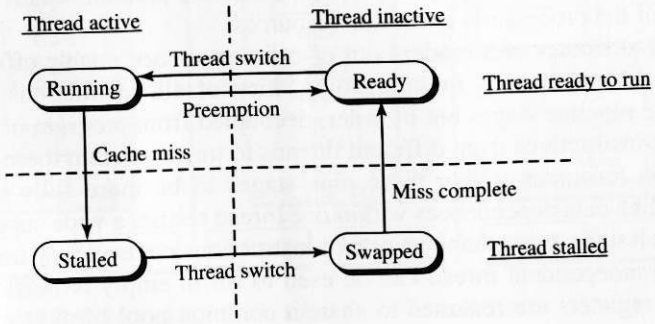


Figure 11.13
CGMT Thread Switch State Machine.

- The primary thread has entered a synchronization spin loop (busy wait), and there is an alternate nonidle thread in the ready state.
- A swapped thread has transitioned to the ready state, and the swapped thread has a higher priority than the primary thread.
- An alternate ready, nonidle thread has not retired an instruction in the last n cycles (avoiding starvation).

Finally, forward progress can be guaranteed by preventing a preemptive thread switch from occurring if the running thread has been active for less than some fixed number of cycles.

Performance and Cost. CGMT has been shown to be a very cost-effective technique for improving instruction throughput. IBM reports that the Northstar/Pulsar line of processors gains about 30% additional instruction throughput at the expense of less than 10% die area and negligible effect on cycle time. The only complexity introduced by CGMT in this incarnation is control complexity for managing thread switches and thread priorities, as well as a doubling of the architected register file to hold two thread contexts instead of one. Finally, the minor software changes required to implement thread priorities must also be figured into the cost equation.

11.4.4 Simultaneous Multithreading

The final and most sophisticated approach for on-chip multithreading is to allow fine-grained and dynamically varying interleaving of instructions from multiple threads across shared execution resources. This technology has recently been commercialized in the Intel Pentium 4 processor but was first proposed in 1995 by researchers at the University of Washington [Tullsen, 1996; Tullsen et al., 1996]. They argued that prior approaches to multithreading shared hardware resources across threads inefficiently, since the thread-switch paradigm restricted either the entire pipeline or minimally each pipeline stage to contain instructions from only a single thread. Since instruction-level parallelism is unevenly distributed, this led to unused instruction slots in each stage of the pipeline and reduced the efficiency of multithreading. Instead, they proposed simultaneous multithreading (SMT), which allows instructions to be interleaved within and across pipeline stages to maximize utilization of the processor's execution resources.

Several attributes of a modern out-of-order processor enable efficient implementation of simultaneous multithreading. First of all, instructions traverse the intermediate pipeline stages out of order, decoupled from program or fetch order; this enables instructions from different threads to mingle within these pipe stages, allowing the resources within these pipe stages to be more fully utilized. For example, when data dependences within one thread restrict a wide superscalar processor from issuing more than one or two instructions per cycle, instructions from an alternate independent thread can be used to fill in empty issue slots. Second, architected registers are renamed to share a common pool of physical registers; this renaming removes the need for tracking threads when resolving data

dependences dynamically. The rename table simply maps the same architected register from each thread to a different physical register, and the standard out-of-order execution hardware takes care of the rest, since dependences are resolved using renamed physical register names. Finally, the extensive buffers (i.e., reorder buffer, issue queues, load/store queue, retired store queue) present in an out-of-order processor to extract and smooth out uneven and irregular instruction-level parallelism can be utilized more effectively by multiple threads, since serializing data and control dependences that can starve the processor now only affect the portion of instructions that belong to the thread that is encountering such a dependence; instructions from other threads are still available to fill the processor pipeline.

11.4.4.1 SMT Resource Sharing. The primary goal of an SMT design is to improve processor resource utilization by sharing those resources across multiple active threads; in fact, the increased parallelism created by multiple simultaneously active threads can be used to justify deeper and wider pipelines, since the additional resources are more likely to be useful in an SMT configuration. However, it is less clear which resources should be shared and which should not or perhaps cannot be shared. Figure 11.14 illustrates a few alternatives, ranging from the design on the left that shares everything but the fetch and retire stages, to the design on the right that shares only the execute and memory stages. Regardless of which design point is chosen, instructions from multiple threads have to be joined before the pipeline stage where resources are shared and must be separated out at the end of the pipeline to preserve precise exceptions for each thread.

Interstage Buffer Implementation. One of the key issues in SMT design, just as in superscalar processor design, is the implementation of the interstage buffers that track instructions as they traverse the pipeline. If the fetch or decode stages

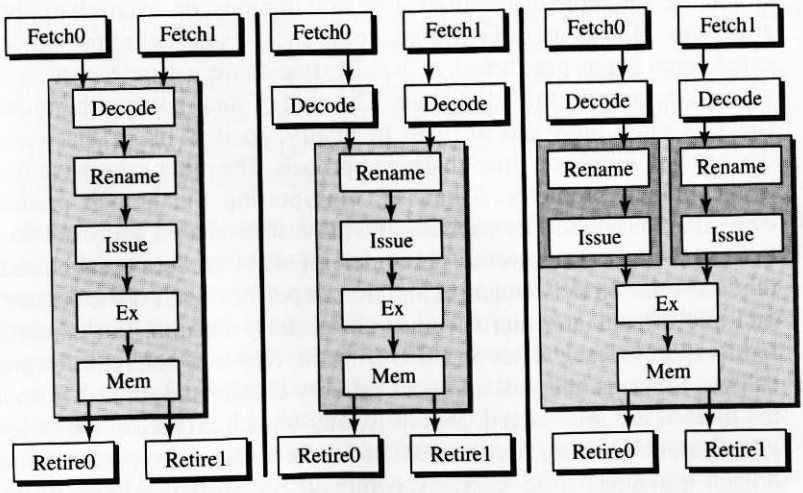


Figure 11.14
SMT Resource Sharing Alternatives.

are replicated, as shown in the left and middle options of Figure 11.14, the stage where the replicated pipelines meet must support multiple simultaneous writers into its buffer. This will complicate the design over a baseline non-SMT processor, since there is only a single writer in that case. Furthermore, the load/store queue and reorder buffer (ROB), which are used to track instructions in program order, must also be redesigned or partitioned to accommodate multiple threads. If they are partitioned per thread, their design will be very similar to the analogous conventional structures. Of course, a partitioned design will preclude best-case single-thread performance, since a single thread will no longer be able to occupy all available slots. Sharing a reorder buffer among multiple threads introduces additional complexity, since program order must be tracked separately for each thread, and the ROB must support selective flushing of nonconsecutive entries to support per-thread branch misprediction recovery. This in turn requires complex free-list management, since the ROB can no longer be managed as a circular queue. Similar issues apply to the load/store queue, but these are further complicated by memory consistency model implications on how the load/store queue resolves memory data dependences; these are discussed briefly here.

SMT Sharing of Pipeline Stages. There are a number of issues that affect how sensible or feasible it is to attempt to share the resources in each pipeline stage; we will discuss some of these issues for each stage, based on the pipeline structure outline in Figure 11.14.

- *Fetch.* The most expensive resource in the instruction fetch stage is the instruction cache port. Since a cache port is limited to accessing a contiguous range of addresses, it would be very difficult to share a single port between multiple threads, as it is very unlikely that more than one thread would be fetching instructions from contiguous or even spatially local addresses. Hence, an SMT design would most likely either provide a dedicated fetch stage per thread or would time-share a single port in a fine-grained or coarse-grained manner. The cost of dual-porting the instruction cache is quite high and difficult to justify, so it is likely that real SMT designs will employ a time-sharing approach. The other expensive resource is the branch predictor. Likewise, multiporting the branch predictor is equivalent to halving its effective size, so a time-shared approach probably makes most sense. However, certain elements of modern branch predictors rely on serial thread semantics and do not perform well if the semantics of multiple threads are interleaved in an arbitrary fashion. For example, the return address stack relies on FIFO (first-in, first-out) behavior for program calls and returns and will not work reliably if calls and returns from multiple threads are interleaved. Similarly, any branch predictor that relies on a global branch history register (BHR) has been shown to perform poorly if branch outcomes from interleaved threads are shifted arbitrarily into the BHR. Hence, it is likely that in a time-shared branch predictor design, at least these elements will need to be replicated for each thread.

- *Decode.* For simple RISC instruction sets, the primary task of the decode stage is to identify source and destination operands and resolve dependences between instructions in a decode group. This involves logic with $O(n^2)$ complexity with respect to decode group width to implement operand specifier comparators and priority decoders. Since there are, by definition, no such inter-instruction dependences between instructions from different threads, it may make sense to partition this resource across threads in order to reduce its complexity. For example, two four-wide decoders could operate in parallel on two threads with much less logic complexity than a single, shared eight-wide decoder. Of course, this design tradeoff could compromise single-thread performance in those cases where a single thread is actually able to supply eight instructions for decoding in a single cycle. For a CISC instruction set, the decode stage is much more complex since it requires determining the semantics of the complex instructions and (usually) decomposing it into a sequence of simpler, RISC-like primitives. Since this can be a very complex task, it may make sense to share the decode stage between threads. However, as with the fetch stage, it may be sensible to time-share it in a fine-grained or coarse-grained manner, rather than attempting to decode instructions from multiple threads simultaneously.

- *Rename.* The rename stage is responsible for allocating physical registers and for mapping architected register names to physical register names. Since physical registers are most likely allocated from a common pool, it makes perfect sense to share the logic that manages the free list between SMT threads. However, mapping architected register names to physical register names is done by indexing into a rename or mapping table with the architected register number and either updating the mapping (for destination operands) or reading it (for source operands). Since architected register numbers are disjoint across threads, the rename table could be partitioned across threads, thus providing high bandwidth into the table at a much lower cost than true multiporting. However, this would imply partitioning the rename stage across threads and, just as with the decode stage, potentially limiting single-thread throughput for programs with abundant instruction-level parallelism.

- *Issue.* The issue stage implements Tomasulo's algorithm for dynamic scheduling of instructions via a two-phase *wakeup-and-select* process: waking up instructions that are data-ready, and then selecting issue candidates from the data-ready pool to satisfy structural dependences. Clearly, if multiple threads are to simultaneously share functional units, the selection process must involve instructions from more than one thread. However, instruction wakeup is clearly limited to intrathread interaction; that is, an instruction wakes up only in response to the execution of an earlier instruction from that same thread. Hence, it may make sense to partition the issue window across threads, since wakeup events will never cross such partitions anyway. Of course, as with the earlier pipe stages, partitioning can

have a negative impact on single-thread performance. However, some researchers have argued that issue window logic will be one of the critical cycle-time-limiting paths in future process technologies. Partitioning this logic to exploit the presence of multiple data-flow-disjoint threads may enable a much larger overall issue window for a fixed cycle-time budget, resulting in better SMT throughput.

- *Execute.* The execute stage realizes the semantics of the instructions by executing each instruction on a functional unit. Sharing the functional units themselves is fairly straightforward, although even here there is an opportunity for multithread optimization: The bypass network that connects functional units to allow back-to-back execution of dependent instructions can be simplified, given that instructions from different threads need never bypass results. For example, in a clustered microarchitecture along the lines of the Alpha 21264, issue logic could be modified to direct instructions from the same thread to the same cluster, hence reducing the likelihood of cross-cluster result bypassing. Alternatively, issue logic could prevent back-to-back issue of dependent instructions, filling the gaps with independent instructions from alternate threads, and hence avoiding the need for the cycle-time critical ALU-output-to-ALU-input bypass path. Again, such optimizations may compromise single-thread performance, except to the extent that they enable higher operating frequency.

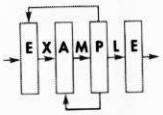
- *Memory.* The memory stage performs cache accesses to satisfy load instructions but is also responsible for resolving memory dependences between loads and stores and for performing other memory-related bookkeeping tasks. Sharing cache access ports between threads to maximize their utilization is one of the prime objectives of an SMT design and can be accomplished in a fairly straightforward manner. However, sharing the hardware that detects and resolves memory dependences is more complex. This hardware consists of the processor's load/store queue, which keeps track of loads and stores in program order and detects if later loads alias to earlier stores. Extending the load/store queue to handle multiple threads requires an understanding of the architected memory consistency model, since certain models (e.g., sequential consistency, see Section 11.3.6) prohibit forwarding a store value from one thread to a load from another. To handle such cases, the load/store queue must be enhanced to be thread-aware, so that it will forward values when it can and will stall the dependent load when it cannot. It may be simpler to provide separate load/store queues for each thread; of course, this will reduce the degree to which the SMT processor is sharing resources across threads and will restrict the effective window size for a single thread to the capacity of its partition of the load/store queue.

- *Retire.* In the retire pipeline stage, instruction results are committed in program order. This involves checking for exceptions or other anomalous conditions and then committing instruction results by updating rename

mappings (in a physical register file-based design) or copying rename register values to architected registers (in a rename register-based design). In either case, superscalar retirement requires checking and prioritizing write-after-write (WAW) dependences (since the last committed write of a register must win) and multiple ports into the rename table or the architected register file. Once again, partitioning this hardware across threads can ease implementation, since WAW dependences can only occur within a thread, and commit updates do not conflict across threads. A viable alternative, provided that retirement latency and bandwidth are not critical, is to time-share the retirement stage in a fine-grained or coarse-grained manner.

In summary, the research to date does not make a clear case for any of the resource-sharing alternatives discussed here. Based on the limited disclosure to date, the Pentium 4 SMT design appears to simultaneously share most of the issue, execute, and memory stages, but performs coarse-grained sharing of the processor front end and fine-grained sharing of the retire pipe-stages. Hence, it is clearly a compromise between the SMT ideal of sharing as many resources as possible and the reality of cycle-time and complexity challenges presented by attempting to maximize sharing.

SMT Support for Serializing Instructions. All instruction sets contain instructions with serializing semantics; typically, such instructions affect the global state (e.g., by changing the processor privilege level or invalidating an address translation) or impose ordering constraints on memory operations (e.g., the memory barriers discussed in Section 11.3.6.3). These instructions are often implemented in a brute-force manner, by draining the processor pipeline of active instructions, applying the semantics of the instruction, and then resuming issue following the instruction. Such a brute-force approach is used because these instructions are relatively rare, and hence even an inefficient implementation does not affect performance very much. Furthermore, the semantics required by the instructions can be quite subtle and difficult to implement correctly in a more aggressive manner, making it difficult to justify a more aggressive implementation.



However, in an SMT design, the frequency of serializing instructions can increase dramatically, since it is proportional to the number of threads. For example, in a single-threaded processor, let's assume that a serializing instruction occurs once every 600 cycles, while in a four-threaded SMT processor that achieves three times the instruction throughput of the single-threaded processor, they will now occur once every 200 cycles. Obviously, a more efficient and aggressive implementation for such instructions may now be required to sustain high performance, since draining the pipeline every 200 cycles will severely degrade performance. The execution of serializing instructions that update the global state can be streamlined by renaming the global state, just as register renaming streamlines execution by removing false dependences between instructions. Once the global state is renamed, only those subsequent instructions that read that state will be delayed, while earlier instructions can continue to read the earlier instance. Hence, instructions from before and after the serializing instruction can be intermingled in the processor's instruction window. However, renaming the global state may not be as easy as it sounds. For example, serializing updates to the translation-lookaside

buffer (TLB) or other address-translation and protection structures may require wholesale or targeted renaming of large array structures. Unfortunately, this will increase the latency of accessing these structures, and such access paths may already be cycle-time-critical. Finally, streamlining the execution of memory barrier instructions, which are used to serialize memory references, requires resolving numerous subtle issues related to the system's memory consistency model; some of these issues are discussed in Section 11.3.6.3. One possible approach for memory barriers is to drain the pipeline selectively for each thread, while still allowing concurrent execution of other threads. This has obvious implications for the reorder buffer design, as well as the issue logic, which must now selectively block issue of instructions from a particular thread while allowing issue to continue from alternate threads. In any case, the complexity implications are nontrivial and largely unexplored in the research literature.

Managing Multiple Threads. Many of the same issues discussed in Section 11.4.3 on coarse-grained multithreading also apply, at least to some extent, to SMT designs. Namely, the processor's issuing policies must provide some guarantee of fairness and forward progress for all active threads. Similarly, priority policies that prevent useless instructions (spin loops, idle loop) from consuming execution resources should be present; similarly, an elevated priority level that provides maximum throughput to thread phases that are performance-critical may also be needed. However, since a pure SMT design has no notion of thread-switching, the mechanism for implementing such policies will be different: rather than switching out a low-priority thread or switching in a high-priority thread, an SMT design can govern execution resource allocation at a much finer granularity, by prioritizing a particular thread in the issue logic's instruction selection phase. Alternatively, threads at various priority levels can be prevented from occupying more than some fixed number of entries in the processor's execution window by gating instruction fetch from those threads. Similar restrictions can be placed on any dynamically allocated resource within the processor. Examples of such resource limits are load/store queue occupancy, to restrict a thread's ability to stress the memory subsystem; or MSHR occupancy, to restrict the number of outstanding cache misses per thread; or entries in a branch or value prediction structure, in order to dedicate more of those resources to high-priority threads.

SMT Performance and Cost. Clearly, there are many subtle issues that can affect the performance of an SMT design. One example is interference between threads in caches, predictors, and other structures. Some published evidence indicates such interference is not excessive, particularly for larger structures such as secondary caches, but the effect on primary caches and other smaller structures is less clear. To date, the only definitive evidence on the performance potential of SMT designs is the preliminary announcement from Intel that claims 16% to 28% throughput improvement for the Pentium 4 design when running server workloads with abundant thread-level parallelism. The following paragraph summarizes some of the details of the Pentium 4 SMT design that have been released. Since the Pentium 4 design has limited machine parallelism, supports only two threads,

and only implements true SMT for parts of the issue, execute, and memory stages, it is perhaps not surprising that this gain is much less than the factor of 2 or 3 improvement reported in the research literature. However, it is not clear that the proposals described in the literature are feasible, or that SMT designs that deal with all the real implementation issues discussed before are scalable beyond two or perhaps three simultaneously active threads. Certainly the cost of implementing SMT, both in terms of implementation complexity as well as resource duplication, has been understated in the research literature to date.

The Pentium 4 Hybrid Multithreading Implementation. The Intel Pentium 4 processor incorporates a hybrid form of multithreading that enables two logical processors to share some of the execution resources of the processor. Intel's implementation—named *hyperthreading*—is conceptually similar to the SMT proposals that have appeared in academic literature, but differs in substantial ways. The limited disclosure to date indicates that the in-order portions of the Pentium 4 pipeline (i.e., the front-end fetch and decode engine and the commit stages) are multithreaded in a fine-grained fashion. That is, the two logical threads fetch, decode, and retire instructions in alternating cycles, unless one of the threads is stalled for some reason. In the latter case a single thread is able to consume all the fetch, decode, or commit resources of the processor until the other thread resolves its stall. Such a scheme could also be described as coarse-grained with a single-cycle time quantum. The Pentium 4 also implements two-stage scheduling logic, where instructions are placed into five issue queues in the first stage and are issued to functional units from these five issue queues in the second stage. Here again, the first stage of scheduling is fine-grained multithreaded: Only one thread can place instructions into the issue queues in any given cycle. Once again, if one thread is stalled, the other can continue to place instructions into the issue queues until the stall is resolved. Similarly, stores are retired from each thread in alternating cycles, unless one thread is stalled. In essence, the Pentium 4 implements a combination of fine-grained and coarse-grained multithreading of all these pipe stages. However, the Pentium 4 does implement true simultaneous multithreading for the second issue stage as well as the execute and memory stages of the pipeline, allowing instructions from both threads to be interleaved in an arbitrary fashion.

Resource sharing in the Pentium 4 is also somewhat complicated. Most of the buffers in the out-of-order portion of the pipeline (i.e., reorder buffer, load queue, store queue) are partitioned in half rather than arbitrarily shared. The scheduler queues are partitioned in a less rigid manner, with high-water marks that prevent either thread from consuming all available entries. As discussed earlier, such partitioning of resources sacrifices maximum achievable single-thread performance in order to achieve high throughput when two threads are available. At a high level, such partitioning can work well if the two threads are largely symmetric in behavior, but can result in poor performance if they are asymmetric and have differing resource utilization needs. However, this effect is mitigated by the fact that the Pentium 4 supports a single-threaded mode in which all resource partitioning is disabled, enabling the single active thread to consume all available resources.

11.5 Implicitly Multithreaded Processors

So far we have restricted our discussion of multithreaded processors and multiprocessor systems to designs that exploit explicit, programmer-created threads to improve instruction throughput. However, there are many important applications where single-thread performance is still of paramount importance. One approach for improving the performance of a single-threaded application is to break that thread down into multiple threads of execution that can be executed concurrently. Rather than relying on the programmer to explicitly create multiple threads by manually parallelizing the application, proposals for *implicit multithreading* (IMT) describe techniques for automatically spawning such threads by exploiting attributes in the program's control flow.

In contrast to automatic compiler-based or manual parallelization of scientific and numeric workloads, which typically attempt to extract thread-level parallelism to occupy dozens to hundreds of CPUs and achieve orders of magnitude speedup, implicit multithreading attempts to sustain up to only a half-dozen or dozen threads simultaneously. This difference in scale is driven primarily by the tightly coupled nature of implicit multithreading, which is caused by threads of execution that tend to be relatively short (tens of instructions) and that often need to communicate large amounts of state with other active threads to resolve data and control dependences. Furthermore, heavy use of speculation in these proposed systems requires efficient recovery from misspeculation, which also requires a tight coupling between the processing elements. All these factors conspire to make it very difficult to scale implicit multithreading beyond a handful of concurrently active threads. Nevertheless, implicit multithreading proposals have claimed nontrivial speedups for applications that are not amenable to conventional approaches for extracting instruction-level parallelism.

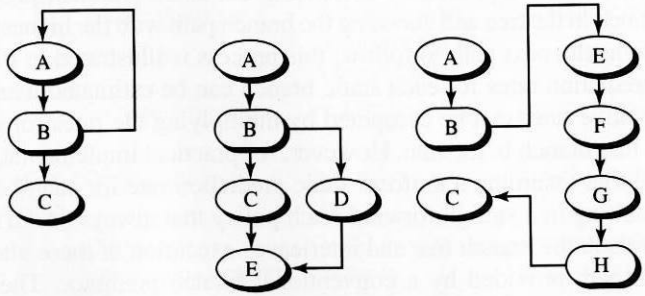
Some IMT proposals are motivated by a desire to extract as much instruction-level parallelism as possible, and achieve this goal by filling a large shared execution window with instructions sequenced from multiple disjoint locations in the program's control flow graph. Other IMT proposals advocate IMT as a means for building more scalable instruction windows: Implicit threads that are independently sequenced can be assigned to and executed in separate processing elements, eliminating the need for a centralized, shared execution window that poses many implementation challenges. Of course, such decentralized designs must still provide a means for satisfying data dependences between the processing elements; much of the research has focused on efficient solutions to this problem.

Fundamentally, there are three main challenges that must be faced when designing an IMT processor. Not surprisingly, these are the same challenges faced by a superscalar design: resolving control dependences, resolving register data dependences, and resolving memory data dependences. However, due to some unique characteristics of IMT designs, resolving them can be substantially more difficult. Some of the proposals rely purely on hardware mechanisms for resolving these problems, while others rely heavily on compilation technology supported by critical hardware assists. We will discuss each of these challenges and describe some of the solutions that have been proposed in the literature.

11.5.1 Resolving Control Dependences

One of the main arguments for IMT designs is the difficulty of effectively constructing and traversing a single thread of execution that is large enough to expose significant amounts of instruction-level parallelism. The conventional approach for constructing a single thread—using a branch predictor to speculatively traverse a program's control flow graph—is severely limited in effectiveness by cumulative branch prediction accuracy. For example, even a 95% accurate branch predictor deteriorates to a cumulative prediction accuracy of only 60% after 10 consecutive branch predictions. Since many important programs have only five or six instructions between conditional branches, this allows the branch predictor to construct a window of only 50 to 60 instructions before the likelihood of a branch misprediction becomes unacceptably high. The obvious solution of improving branch prediction accuracy continues to be an active field of research; however, the effort and hardware required to incrementally improve the accuracy of predictors that are already 95% accurate can be prohibitive. Furthermore, it is not clear if significant improvements in branch prediction accuracy are possible.

Control Independence. All proposed IMT designs exploit the program attribute of control independence to increase the size of the instruction window beyond joins in the control flow graph. A node in a program's control flow graph is said to be control-independent if it post-dominates the current node, that is, if execution will eventually reach that node regardless of how intervening conditional branches are resolved. Figure 11.15 illustrates several sources of control independence in programs. In the proposed IMT designs, implicit threads can be spawned at joins in the control flow, at subroutine return addresses, across loop iterations, or at the loop fall-through point. These threads can often be spawned nonspeculatively, since control independence guarantees that the program will eventually reach these



(a) Loop-closing (b) Control-flow convergence (c) Call/return

There are multiple sources of control independence: in (a), block C eventually follows block B since the loop has a finite number of iterations; in (b) block E always follows B independent of which way the branch resolves; and in (c), block C eventually follows block B after the subroutine call to E completes.

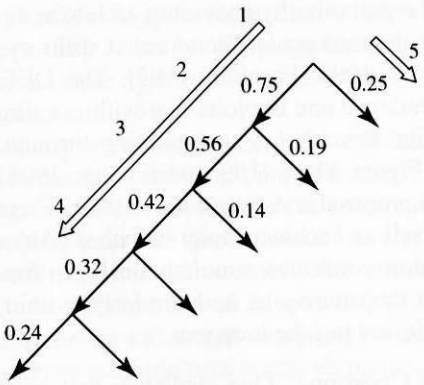
Figure 11.15
Sources of Control Independence.

initiation points. However, they can also be spawned speculatively, to encompass cases where the intervening control flow cannot be fully determined at the time the thread is spawned. For example, a loop that traverses a linked list may have a data-dependent number of iterations: Spawning speculative threads for multiple iterations into the future will often result in better performance, even when some of those speculative threads need to eventually be squashed as incorrect.

Spawning an implicit *future thread* at a subsequent control-independent point in the program's control flow has several advantages. First of all, any intermediate branch instructions that may be mispredicted will not directly affect the control independent thread, since it will be executed no matter what control flow path is used to reach it. Hence, exploiting control independence allows the processor to skip ahead past hard-to-predict branches to find useful instructions. Second, skipping ahead can have a positive prefetching effect. That is to say, the act of fetching instructions from a future point in the control flow can effectively overlap useful work from the current thread with instruction cache misses caused by the future thread. Conversely, the current thread may also encounter instruction cache misses which can now be overlapped with the execution of the future thread. Note that such prefetching effects are impossible with conventional single-threaded execution, since the current and future thread's instruction fetches are by definition serialized. This prefetching effect can be substantial; Akkary reports that a DMT processor fetches up to 40% of its committed instructions from beyond an intervening instruction cache miss [Akkary and Driscoll, 1998].

Disjoint Eager Execution. An interesting alternative for creating implicit threads is proposed in the disjoint eager execution (DEE) architecture [Uht and Sindagi, 1995]. Conventional eager execution attempts to overcome conditional branches by executing both paths following a branch. Of course, this results in a combinatorial explosion of paths as multiple branches are traversed. In the DEE proposal, the eager execution decision tree is pruned by comparing cumulative branch prediction rates along each branch in the tree and choosing the branch path with the highest cumulative prediction rate as the next path to follow; this process is illustrated in Figure 11.16. The branch prediction rates for each static branch can be estimated using profiling, and the cumulative rates can be computed by multiplying the rates for each branch used to reach that branch in the tree. However, for practical implementation reasons, Uht has found that assuming a uniform static prediction rate for each branch works quite well, resulting in a straightforward fetch policy that always backtracks a fixed number of levels in the branch tree and interleaves execution of these alternate paths with the main path provided by a conventional branch predictor. These alternate paths are introduced into the DEE core as implicit threads.

Table 11.3 summarizes four IMT proposals in terms of the control flow attributes they exploit; what the sources of implicit threads are, how they are created, sequenced, and executed; and how dependences are resolved. In cases where threads are created by the compiler, program control flow is statically analyzed to determine opportune thread creation points. Most simply, the thread-level speculation (TLS) proposals create a thread for each iteration of a loop at compile time to harness parallelism [Steffan et al., 1997]. The multiscalar proposal allows much



Assuming each branch is predicted with 75% accuracy, the cumulative branch prediction rate is shown; after fetching branch paths 1, 2, 3, and 4, the next-highest cumulative rate is along branch path 5, so it is fetched next.

Figure 11.16
Disjoint Eager Execution.
Source: Uht and Sindagi, 1995.

Table 11.3
Attributes of several implicit multithreading proposals

	Multiscalar	Disjoint Eager Execution (DEE)	Dynamic Multi-threading (DMT)	Thread-Level Speculation (TLS)
Control flow attribute exploited	Control independence	Control independence, cumulative branch misprediction	Control independence	Control independence
Source of implicit threads	Loop bodies, control flow joins	Loop bodies, control flow joins, cumulative branch mispredictions	Loop exits, subroutine returns	Loop bodies
Thread creation mechanism	Software/compiler	Implicit hardware	Implicit hardware	Software/compiler
Thread creation and sequencing	Program order	Out of program order	Out of program order	Program order
Thread execution	Distributed processing elements	Shared processing elements	Shared multithreaded processing elements	Separate CPUs
Register data dependences	Software with hardware speculation support	Hardware; no speculation	Hardware; data dependence prediction and speculation	Disallowed; compiler must avoid
Memory data dependences	Hardware-supported speculation	Hardware	Hardware; prediction and speculation	Dependence speculation; checked with simple extension to MESI coherence

greater flexibility to the compiler by providing architected primitives for spawning threads (called *tasks* in the multiscalar literature) at arbitrary points in the program's control flow [Sohi et al., 1995; Franklin, 1993]. The DEE proposal dynamically detects control independence and exploits that within a single instruction window, but also creates implicit threads by backtracking through the branch prediction tree, as illustrated in Figure 11.16 [Uht and Sindagi, 1995]. Finally, the dynamic multithreading (DMT) proposal uses hardware detection heuristics to spawn threads at procedure calls as well as backward loop branches [Akkary and Driscoll, 1998]. In these cases execution continues simultaneously within the procedure call as well as following it, at the return site, and similarly, within the next loop iteration as well as at the code following the loop exit.

Out-of-Order Thread Creation. One challenge that is unique to the DMT approach is that threads are spawned out of program order. For example, in the case of nested procedure calls, the first call will spawn a thread for executing the call, as well as executing the code at the subroutine return site, resulting in two active threads. The code in the called procedure now encounters the nested procedure call and spawns an additional thread to execute that call, resulting in three active threads. However, this thread, though created third, actually occurs before the second thread in program order. As a result, the logical reorder buffer used in this design now has to support out-of-order insertion of an arbitrary number of instructions into the middle of a set of already active instructions. As we will see, the process of resolving register and memory data dependences is also substantially complicated by out-of-order thread creation. Whether such an approach is feasible remains to be seen.

Physical Organization. Of course, constructing a large window of instructions is only half the battle; any design that attempts to detect and exploit parallelism from such a window must demonstrate that it is feasible to build hardware that accomplishes such a feat. Many IMT proposals partition the execution resources of a processor so that each thread executes independently on a partition, enabling distributed and scalable extraction of instruction-level parallelism. Since each partition need only contain the instruction window of a single thread, it need not be more aggressive than a current-generation design. In fact, it may even be less aggressive. Additional parallelism is extracted by overlapping the execution of multiple such windows. For TLS proposals, each partition is actually an independent microprocessor core in a system that is very similar to a multiprocessor, or chip multiprocessor (CMP, as discussed in Section 11.4.1). In contrast, the DMT proposal relies on an SMT-like multithreaded execution core that tracks and interleaves implicit threads instead of explicit threads. DMT also proposes a hierarchical two-level reorder buffer that enables a very large instruction window; threads that have finished execution but cannot be committed migrate to the second level of the reorder buffer and are only fetched out of the second level in case they need to re-execute due to data mispredictions. Finally, the DEE processor has a centralized execution window that tracks multiple implicit threads simultaneously by organizing the window basic on the static program structure rather than

a dynamic single path. That is to say, the instruction window of the DEE prototype design, Levo, captures a static view of the program and includes hardware for simultaneously tracking multiple dynamic instances of the same static control flow constructs (e.g., loop bodies).

Finally, the multiscalar proposal is structured as a circular queue of processing elements. The tail of the queue is considered nonspeculative and executes the current thread or task; other nodes are executing future tasks that can be speculative with respect to both control and data dependences. As the tail thread completes execution, its results are retired, and the next node becomes the nonspeculative tail node. Simultaneously, a new future thread is spawned to occupy the processing element that was freed up as the tail thread completed execution. In this way, by overlapping execution across multiple processing elements, additional parallelism is exposed beyond what can be extracted by a single processing element.

Thread Sequencing and Retirement. One of the most challenging aspects of IMT designs is the control and/or prediction hardware that must sequence threads and retire them in program order. Relying on compiler assistance for creating threads can ease this task. Similarly, a queue-based machine organization such as multiscalar can at least conceptually simplify the task of sequencing and retiring tasks. However, all proposals share the need for control logic that determines that no correctness violations have occurred before a task is allowed to retire and update the architected state. Control dependence violations are fairly straightforward; as long as nonspeculative control flow eventually reaches the thread in question, and as long as control flow leaves that thread and proceeds to the next speculative thread, the thread can safely be retired. However, resolving data dependences can be quite a bit more complex and is discussed in the following.

11.5.2 Resolving Register Data Dependences

Register data dependences consist of name or false (WAR and WAW) dependences and true data dependences (RAW). In IMT designs, just as in conventional superscalar processors, the former are solved via register renaming and in-order commit. The only complication is that in-order commit has to be coordinated across multiple threads, but this is easily resolved by committing threads in program order.

True register data dependences can be broken down into two types: dependences within a thread or *intrathread dependences,* and dependences across threads or *interthread dependences*. Intrathread dependences can be resolved with standard techniques studied in earlier chapters, since instructions within a thread are sequenced in program order, and can be renamed, bypassed, and eventually committed using conventional means. Interthread dependences, however, are complicated by the fact that instructions are now sequenced out of program order. For this reason, it can be difficult to identify the correct producer-consumer relationships, since the producer or register-writing instruction may not have been decoded yet at the time the consumer or register-reading instruction becomes a candidate for execution. For example, this can happen when a register value is

read near the beginning of a new thread, while the last write to that register value does not occur until near the end of the prior thread. Since the prior thread is still busy executing older instructions, the instruction that performs the last write has not even been fetched yet. In such a scenario, conventional renaming hardware fails to correctly capture the true dependence, since the producing instruction has not updated the renaming information to reflect its pending write. Hence, either simplifications to the programming model or more sophisticated renaming solutions are necessary to maintain correct execution.

The easiest solution for resolving interthread register data dependences is to simplify the programming model by disallowing them at compile-time. Thread-level speculation proposals take this approach. As the compiler creates implicit threads for parallel execution, it is simply required to communicate all shared operands through memory with loads and stores. Register dependences are tracked within threads only, using well-understood techniques like register renaming and Tomasulo's algorithm, just as in a single-threaded uniprocessor.

In contrast, the multiscalar proposal allows register communication between implicit threads, but also enlists the compiler's help by requiring it to identify interthread register dependences explicitly. This is done by communicating to the future thread, as it is created, which registers in the register file have pending writes to them, and also marking the last instruction to write to any such register so that the prior thread's processing element knows to forward it to future tasks once the write occurs. Transitively, pending writes from older threads must also be forwarded to future threads as they arrive at a processing element. The compiler embeds this information in a write mask that is provided to the future thread when it is spawned. Thus, with helpful assistance from the compiler, it is possible to effectively implement a distributed, scalable dependence resolution scheme with relatively straightforward hardware implementation.

The DEE and DMT proposals assume no compiler assistance, however, and are responsible for dynamically resolving data dependences. The DEE proposal constructs a single, most likely thread of execution, and fetches and decodes all the instructions along that path in program order. Hence, identifying data dependences along that path is relatively straightforward. The alternate *eager execution* paths, which we treat as implicit threads in our discussion, have similar sequential semantics, so forward dependence resolution is possible. However, the DEE proposal also detects control independence by implementing *minimal control dependences* (MCD). The hardware for MCD is capable of identifying and resolving data dependences across divergent control flow paths that eventually join, as these paths are introduced into the execution window by the DEE fetch policy. The interested reader is referred to Uht and Sindagi [1995] for a description of this novel hardware scheme.

The DMT proposal, on the other hand, does not have a sequential instruction stream to work with. Hence, the most challenging task is identifying the last write to a register that is read by a future thread, since the instruction performing that write may not have been fetched or decoded yet. The simplistic solution is to assume that all registers will be written by the current thread and to delay register

reads in future threads until all instructions in the current thread have been fetched and decoded. Of course, this will result in miserable performance. Hence, the DMT proposal relies on data dependence speculation, where future threads assume that their register operands are already stored in the register file and proceed to execute speculatively with those operands. Of course, the future threads must recover by re-executing such instructions if an older thread performs a write to any such register. The DMT proposal describes complex dependence resolution mechanisms that enable such re-execution whenever a dependence violation is detected. In addition, researchers have explored adaptive prediction mechanisms that attempt to identify pending register writes based on historical information. Whenever such a predictor identifies a pending write, dependent instructions in future threads are stalled, and hence prevented from misspeculating with stale data. Furthermore, the register dependence problem can also be eased by employing value prediction; in cases of pending or unknown but likely pending writes, the operand's value can be predicted, forwarded to dependent operands, and later verified. Many of the issues discussed in Chapter 10 regarding value prediction, verification, and recovery will apply to any such design.

11.5.3 Resolving Memory Data Dependences

Finally, an implicit multithreading design must also correctly resolve memory data dependences. Here again, it is useful to decompose the problem into intrathread and interthread memory dependences. Intrathread memory dependences, just as intrathread register dependences, can be resolved with conventional and well-understood techniques from prior chapters: WAW and WAR dependences are resolved by buffering stores until retirement, and RAW dependences are resolved by stalling dependent loads or forwarding from the load/store queue.

Interthread false dependences (WAR and WAW) are also solved in a straightforward manner, by buffering writes from future threads and committing them when those threads retire. There are some subtle differences among the proposed alternatives. The DEE and DMT proposals use structures similar to conventional load/store queues to buffer writes until commit. The multiscalar design uses a complex mechanism called the *address resolution buffer* (ARB) to buffer in-flight writes. Finally, the TLS proposal extends conventional MESI cache coherence to allow multiple instances of cache lines that are being written by future threads. These future instances are tagged with an epoch number that is incremented for each new thread. The epoch number is appended to the cache line address, allowing conventional MESI coherence to support multiple modified instances of the same line. The retirement logic is then responsible for committing these modified lines by writing them back to memory whenever a thread becomes nonspeculative.

True (RAW) interthread memory dependences are significantly more complex than true register dependences, although conceptually similar. The fundamental difficulty is the same: since instructions are fetched and decoded out of program order, later loads are unable to obtain dependence information with respect to earlier stores, since those stores may not have computed their target addresses yet or may not have even been fetched yet.

TLS Memory RAW Resolution. Again, the simplest solution is employed by the TLS design: Future threads simply assume that no dependence violations will occur and speculatively consume the latest available value for a particular memory address. This is accomplished by a simple extension to conventional snoop-based cache coherence: When a speculative thread executes a load that causes a cache miss, the caches of the other processors are searched in reverse program order for a matching address. By searching in reverse program order (i.e., reverse thread creation order), the latest write, if any, is identified and used to satisfy the load. If no match is found, the load is simply satisfied from memory, which holds the committed state for that cache line. In effect the TLS scheme is predicting that any actual store to load dependences occur far enough apart that the older thread will already have performed the relevant store, resulting in a snoop hit when the newer thread issues its load miss. Only those cases where the store and load are actually executed out of order across the speculative threads will result in erroneous speculation.

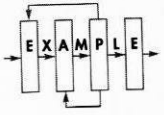
Of course, since TLS is employing a simple form of data dependence speculation, a mechanism is needed to detect and recover from violations that may occur. Again, a simple extension to the existing cache coherence protocol is employed. There are two cases that need to be handled: first, if the future load is satisfied from memory, and second, if the future load is satisfied by a modified cache line written to by an earlier thread. In the former case, the cache line is placed in the future thread's cache in the exclusive state, since it is the only copy in the system. Subsequently, an older thread performs a store to the same cache line, hence causing a potential dependence violation. In order to perform the store, the older thread must snoop the other caches in the system to obtain exclusive access to the line. At this point, the future thread's copy of the line is discovered, and that thread is squashed due to the violation. The latter case, where the future thread's load was satisfied from a modified line written by an older thread, is very similar. The line is placed in the future thread's cache in the shared state and is also downgraded to the shared state in the older thread's cache. This is exactly what would happen when satisfying a remote read to a modified line, as shown earlier in Figure 11.5. When the older thread writes to the line again, it has to upgrade the line by snooping the other processor's caches to invalidate their copies. At this point, again, the future thread's shared copy is discovered and a violation is triggered. The recovery mechanism is simple: the thread is squashed and restarted.

DMT Memory RAW Resolution. The DMT proposal handles true memory dependences by tracking the loads and stores from each thread in separate per-thread load and store queues. These queues are used to handle intrathread memory dependences in a conventional manner, but are also used to resolve interthread dependences by conducting cross-thread associative searches of earlier threads' store queues whenever a load issues and later threads' load queues whenever a store issues. A match in the former case will forward the store data to the dependent load; a match in the latter case will signal a violation, since the load has already executed with stale data, and will cause the later thread to reissue the load and its dependent instructions. Effectively, the DMT mechanism achieves memory

renaming, since multiple instances of the same memory location can be in flight at any one time, and dependent loads will be satisfied from the correct instance as long as all the writes in the sequence have issued and are present in the store queues. Of course, if an older store is still pending, the mechanism will fail to capture dependence information correctly and the load will proceed with potentially incorrect data and will have to be restarted once the missing store does issue.

DEE Memory RAW Resolution. The DEE proposal describes a mechanism that is conceptually similar to the DMT approach but is described in greater detail. DEE employs an address-interleaved, high-throughput structure that is capable of tracking program order and detecting dependence violations whenever a later load reads a value written by an earlier store. Again, since these loads and stores can be performed out of order, the mechanism must logically sort them in program order and flag violations only when they actually occurred. This is complicated by the fact that implicit threads spawned by the DEE fetch policy can also contain stores and must be tracked separately for each thread.

Multiscalar ARB. The multiscalar address resolution buffer (ARB) is a centralized, multiported, address-interleaved structure that allows multiple in-flight stores to the same address to be correctly resolved against loads from future threads. This structure allocates a tracking entry for each speculative load as it is performed by a future thread and checks subsequent stores from older threads against such entries. Any hit will flag a violation and cause the violating thread and all future threads to be squashed and restarted. Similarly, each load is checked against all prior unretired stores, which are also tracked in the ARB, and any resulting data dependence is satisfied with data from the prior store, rather than from the data cache. It should be noted that such prior stores also form visibility barriers to older unexecuted stores, due to WAW ordering. For example, let's say a future thread n + 1 stores to address A. This store is placed in the ARB. Later on, future thread n + 2 reads from address A; this read is satisfied by the ARB from thread n + 1's store entry. Eventually, current thread n performs a store against A. A naive implementation would find the future load from thread n + 2, and squash and refetch thread n + 2 and all newer future threads. However, since thread n + 1 performed an intervening store to address A, no violation has actually occurred and thread n + 2 need not be squashed.



Implementation Challenges. The main drawback of the ARB and similar, centralized designs that track all reads and writes is scalability. Since each processing element needs high bandwidth into this structure, scaling to a significant number of processing elements becomes very difficult. The TLS proposal avoids this scalability problem by using standard caching protocols to filter the amount of traffic that needs to be tracked. Since only cache misses and cache upgrades need to be made visible outside the cache, only a small portion of references are ordered and checked against the other processing elements. Ordering within threads is provided by conventional load and store queues within the processor. An analogous cache-based enhancement of the ARB, the *speculative versioning cache,* has also been

proposed for multiscalar. Of course, the corresponding drawback of cache-based filtering is that false dependences arise due to address granularity. That is to say, since cache coherence protocols operate on blocks that are larger than a single word (usually 32 to 128 bytes), a write to one word in the block can falsely trigger a violation against a read from a different word in the same block, causing additional recovery overhead that would not occur with a more fine-grained dependence mechanism.

Other problems involved with memory dependence checking are more mundane. For example, limited buffer space can stall effective speculation, just as a full load or store queue can stall instruction fetch in a superscalar processor. Similarly, commit bandwidth can cause limitations, particularly for TLS systems, since commit typically involves writing modified lines back to memory. If a speculative thread modifies a large number of lines, writeback bandwidth can limit performance, since a future thread cannot be spawned until all commits have been performed. Finally, TLS proposals as well as more fine-grained proposals all suffer from the inherently serial process of searching for the newest previous write when resolving dependences. In the TLS proposal, this is accomplished by serially snooping the other processors in reverse thread creation order. The other IMT proposals suggest parallel associative lookups, which are faster, but more expensive and difficult to scale to large numbers of processing elements.

11.5.4 Concluding Remarks

To date, implicit multithreading exists only in research proposals. While it shows dramatic potential for improving performance beyond what is achievable with single-threaded execution, it is not clear if all the implementation issues discussed here, as well as others that may not be discovered until someone attempts a real implementation, will ultimately prevent the adoption of IMT. Certainly, as chip multiprocessor designs become widespread, it is quite likely that the simple enhancements required for thread-level speculation in such systems will in fact become available. However, these changes will only benefit applications that have execution characteristics that match TLS hardware and that can be recompiled to exploit such hardware. The more complex schemes—DEE, DMT, and multiscalar—require much more dramatic changes to existing processor implementations, and hence must meet a higher standard to be adopted in real designs.

11.6 Executing the Same Thread

So far, we have discussed both explicitly and implicitly multithreaded processor designs that attempt to sequence instructions from multiple threads of execution to maximize processor throughput. An interesting alternative that several researchers have proposed is to execute the same instructions in multiple contexts. Although it may seem counterintuitive, there are several potential benefits to such an approach. The first proposal to suggest doing so [Rotenberg, 1999], active-stream/redundant-stream simultaneous multithreading (AR-SMT), focused on fault detection. By executing an instruction stream twice in separate thread contexts and comparing

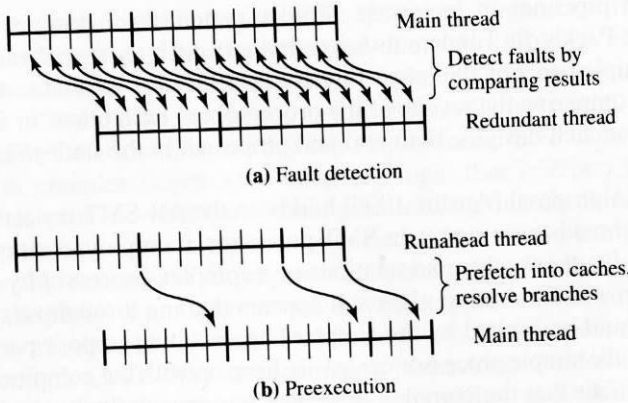
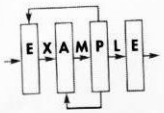


Figure 11.17
Executing the Same Thread.

execution results across the threads, transient errors in the processing pipeline can be detected. That is to say, if the pipeline hardware flips a bit due to a soft error in a storage cell, the likelihood of the same bit being flipped in the redundant stream is very low. Comparing results across threads will likely detect many such transient errors. An interesting observation grew out of this work on fault detection: namely, that the active and redundant streams end up helping each other execute more effectively. That is to say, they can prefetch memory references for each other and can potentially resolve branch mispredictions for each other as well. This cooperative effect has been exploited in several research proposals. We will discuss some of these proposals in the context of these benefits—fault detection, prefetching, and branch resolution—in this section. Figure 11.17 illustrates these uses for executing the same thread; Figure 11.17(a) shows how a redundant thread can be used to check the main thread for transient faults, while Figure 11.17(b) shows how a runahead thread can prefetch cache misses and resolve mispredicted branches for the main thread.

11.6.1 Fault Detection

As described, the original work in redundant execution of the same thread was based on the premise that inconsistencies in execution between the two thread instances could be used to detect transient faults. The AR-SMT proposal assumes a baseline SMT processor and enhances the front end of the SMT pipeline to replicate the fetched instruction stream into two separate thread contexts. Both contexts then execute independently and store their results in a reorder buffer. The commit stage of the pipeline is further enhanced to compare instruction outcomes, as they are committed, to check for inconsistencies. Any such inconsistencies are used to identify transient errors in the execution pipeline. A similar approach is used in real processor designs that place emphasis on fault detection and fault tolerance. For example, the IBM S/390 G5 processor also performs redundant execution of all instructions, but achieves this by replicating the pipeline hardware on chip and



running both pipelines in lock step. Similar system-level designs are available from Hewlett Packard's Tandem division; in these designs, two physical processor chips are coupled to run the same threads in a lockstep manner, and faults are detected by comparing the results of the processors to each other. In fact, there is a long history of such designs, both real and proposed, in the fault-tolerant computing domain.

The DIVA proposal [Austin, 1999] builds on the AR-SMT concept, but instead of using two threads running on an SMT processor, it employs a simple processor that dynamically checks the computations of a complex processor by re-executing the instruction stream. At first glance, it appears that the throughput of the pair of processors would be limited by the simpler one, resulting in poor performance. In fact, however, the simple processor can easily keep up with the complex processor if it exploits the fact that the complex processor has speculatively resolved all control and data flow dependences. Since this is the case, it is trivial to parallelize the code running on the simple processor, since all dependences are removed: All conditional branches are resolved, and all data dependences disappear since input and output operand values are already known. The simple processor need only verify each instruction in isolation, by executing with the provided inputs and comparing the output to the provided output. Once each instruction is verified in this manner, then, by induction, the entire instruction stream is also verified. Since the simple processor is by definition easy to verify for correctness, it can be trusted to check the operation of the much more complex and design-error-prone runahead processor. Hence, this approach is able to cover design errors in addition to transient faults.

Dynamic verification with a simple, slower processor does have one shortcoming that has not been adequately addressed in the literature. As long as the checker processor is only used to verify computation (i.e., ALU operations, memory references, branches), it is possible to trivially parallelize the checking, since each computation that is being checked is independent of all others. However, this relies on the complex processor's ability to provide correct operands to all these computations. In other words, the operand communication that occurs within the complex processor is not being checked, since the checker relies on the complex processor to perform it correctly. Since operand communication is one of the worst sources of complexity in a modern out-of-order processor, one could argue that the checker is focusing on the wrong problem. In other words, in terms of fault coverage, one could argue that checking communication is much more important than checking computation, since it is relatively straightforward to verify the correctness of ALUs and other computational paths that can be viewed as combinational delay paths. On the other hand, verifying the correctness of complex renaming schemes and associative operand bypassing is extremely difficult. Furthermore, soft errors in the complex processor's register file would also not be detected by a DIVA checker that does not check operand communication.

To resolve this shortcoming, the DIVA proposal also advocates checking operand communication separately in the checker processor. The checker decodes each instruction, reads its source operands from a register file, and writes its result

operands to the same checker register file. However, the process of reading and writing register operands that may have read-after-write (RAW), write-after-read (WAR), and write-after-write (WAW) dependences with instructions immediately preceding or following the instruction being checked is not trivial to parallelize. As explained in detail in Chapter 5, such dependences have to be detected and resolved with complex dependence-checking logic that is $O(n^2)$ in complexity with respect to pipeline width n. Hence, parallelizing this checking process will require hardware equivalent in complexity to the hardware in the complex processor. Furthermore, if, as the DIVA proposal advocates, the checker processor runs slower than the baseline processor, it will have to support a wider pipeline to avoid becoming the execution bottleneck. In this case, the checker must actually implement more complex logic than the processor it is checking. Further investigation is needed to determine how much of a problem this will be and whether it will prevent the adoption of DIVA as a design technique for enhancing fault tolerance and processor performance.

11.6.2 Prefetching

One positive side effect of redundant execution can be prefetching, since both threads are generating the same stream of instruction and data memory references. Whenever one thread runs ahead of the other, it prefetches useful instructions and data into the processor's caches. This can result in a net speedup, since additional memory-level parallelism is exposed. The key to extracting significant performance benefit is to maximize the degree of runahead, or slip, between the two threads. The *slipstream* processor proposal [Sundaramoorthy et al., 2000] does exactly that, by specializing the runahead thread; instead of redundantly executing all instructions in the program, the runahead thread is stripped down so that instructions that are considered nonessential are removed from execution. Nonessential instructions are ones that have no effect on program outcome or only contribute to resolving predictable branches. Since the runahead thread no longer needs to execute these instructions, it is able to get further ahead in the control flow of the program, increasing the slip between the two threads and improving the timeliness of the prefetches that it creates.

The principle of maximizing slip to ensure timeliness has been further refined in proposals for *preexecution* [Roth, 2001; Zilles, 2002; Collins et al., 2001]. In these proposals, profiling information is used to identify problematic instructions like branch instructions that are frequently mispredicted or load instructions that frequently cause cache misses. The backward dynamic data flow slice for such instructions is then constructed at compile time. The instructions composing that backward slice then form a speculative preexecution thread that is spawned at run time in an available thread context on an SMT-like processor. The preexecuted slice will then precompute the outcome for the problematic instruction and issue a prefetch to memory if it misses. Subsequently, the worker thread catches up to the preexecuted instruction and avoids the cache miss.

The main benefit of slipstreaming and preexecution over the implicit multithreading proposals discussed in Section 11.5 is that the streamlined runahead

thread has no correctness requirement. That is, since it is only serving to generate prefetches and "assist" the main thread's execution, and it has no effect on the architected program state, generating and executing the thread is much easier. None of the issues regarding control and data dependence resolution have to be solved exactly. Of course, precision in dependence resolution is likely to result in a more useful runahead thread, since it is less likely to issue useless prefetches from paths that the real thread never reaches; but this is a performance issue, rather than a correctness issue, and can be solved much more easily.

Intel has described a fully functional software implementation of preexecution for the Pentium 4 SMT processor. In this implementation, a runahead thread is spawned and assigned to the same physical processor as the main thread; the runahead thread then prefetches instructions and data for the main thread, resulting in a measurable speedup for some programs.

An alternative and historically interesting approach that uses redundant execution for data prefetching is the datascalar architecture [Burger et al., 1997]. In this architecture, memory is partitioned across several processors that all execute the same program. The processors are connected by a fast broadcast network that allows them to communicate memory operands to each other very quickly. Each processor is responsible for broadcasting all references to its local partition of memory to all the other processors. In this manner, each reference is broadcast once, and each processor is able to satisfy all its references either from its local memory or from a broadcast initiated by the owner of that remote memory. With this policy, all remote memory references are satisfied in a request-free manner. That is to say, no processor ever needs to request a copy of a memory location; if it is not available locally, the processor need only wait for it to show up on the broadcast interconnect, since the remote processor that owns the memory will eventually execute the same reference and broadcast the result. The net result is that average memory latency no longer includes the request latency, but consists simply of the transfer latency over the broadcast interconnect. In many respects, this is conceptually similar to the redundant-stream prefetching used in the slipstream and preexecution proposals.

11.6.3 Branch Resolution

The other main benefit of both slipstreaming and preexecution is early resolution of branch instructions that are hard to predict with conventional approaches to branch prediction. In the case of slipstreaming, instructions that are data flow antecedents of the problematic branch instructions are considered essential and are therefore executed in the runahead thread. The branch outcome is forwarded to the real thread so that when it reaches the branch, it can use the precomputed outcome to avoid the misprediction. Similarly, preexecution constructs a backward program slice for the branch instruction and spawns a speculative thread to preexecute that slice. The main implementation challenge for early resolution of branch outcomes stems from synchronizing the two threads. For instruction and data prefetching, no synchronization is necessary, since the real thread's instruction fetch or memory reference will benefit by finding its target in the instruction or data cache, instead

of experiencing a cache miss. In effect, the threads are synchronized through the instruction cache or data cache, which tolerates some degree of inaccuracy in both the fetch address (due to spatial locality) as well as the timing (due to temporal locality). As long as the prefetches are timely, that is to say they occur neither too late (failing to cover the entire miss latency) or too early (where the prefetched line is evicted from the cache before the real thread catches up and references it), they are beneficial.

However, for branch resolution, the preexecuted branch outcome must be exactly synchronized with the same branch instance in the real thread; otherwise, if it is applied to the wrong branch, the early resolution-based prediction may fail. The threads cannot simply synchronize based on the static branch (i.e., branch PC), since multiple dynamic instances of the same static branch can exist in the slip-induced window of instructions between the two threads. Hence, a reference-counting scheme must be employed to make sure that a branch is resolved with the correct preexecuted branch outcome. Such a reference-counting scheme must keep track of exactly how many instances of each static branch separate the runahead thread from the main thread. The outcome for each instance is stored in an in-order queue that separates the two threads; the runahead thread inserts new branch outcomes into one end of this queue, while the main thread removes outcomes from the other end. If the queue length is incorrect, and the two threads become unsynchronized, the predicted outcomes are not likely to be very useful. Building this queue and the associated control logic, as well as mechanisms for flushing it whenever mispredictions are detected, is a nontrivial problem that has not been satisfactorily resolved in the literature to date.

Alternatively, branch outcomes can be communicated indirectly through the existing branch predictor by allowing the runahead thread to update the predictor's state. Hence, the worker thread can benefit from the updated branch predictor state when it performs its own branch predictions, since the two threads synchronize implicitly through the branch predictor. However, the likelihood that the runahead thread's predictor update is both timely and accurate are low, particularly in modern branch predictors with multiple levels of history.

11.6.4 Concluding Remarks

Redundant execution of the same instructions has been proposed and implemented for fault detection. It is quite likely that future fault-tolerant implementations will employ redundant execution in the context of SMT processors, since the overhead for doing so is quite reasonable and the fault coverage can be quite helpful, particularly as smaller transistor dimensions lead to increasing vulnerability to soft errors. Exploiting redundant-stream execution to enhance performance by generating prefetches or resolving branches early has not yet reached real designs. It is likely that purely software-based redundant-stream prefetching will materialize in the near future, since it is at least theoretically possible to achieve without any hardware changes; however, the performance benefits of a software-only scheme are less clear. The reported performance benefits for the more advanced preexecution and slipstream proposals are certainly attractive; assuming that baseline SMT and

CMP designs become commonplace in the future, the extensions required for supporting these schemes are incremental enough that it is likely they will be at least partially adopted.

11.7 Summary

This chapter discusses a wide range of both real and proposed designs that execute multiple threads. Many important applications, particularly in the server domain, contain abundant thread-level parallelism and can be efficiently executed on such systems. We discussed explicit multithreaded execution in the context of both multiprocessor systems and multithreaded processors. Many of the challenges in building multiprocessor systems revolve around providing a coherent and consistent view of memory to all threads of execution while minimizing average memory latency. Multithreaded processors enable more efficient designs by sharing execution resources either at the chip level in chip multiprocessors (CMP), in a fine-grained or coarse-grained time-sharing manner in multithreaded processors that alternate execution of multiple threads, or seamlessly in simultaneous multithreaded (SMT) processors. Multiple thread contexts can also be used to speed up the execution of serial programs. Proposals for doing so range from complex hardware schemes for implicit multithreading to hybrid hardware/software schemes that employ compiler transformations and critical hardware assists to parallelize sequential programs. All these approaches have to deal correctly with control and data dependences, and numerous implementation challenges remain. Finally, multiple thread contexts can also be used for redundant execution, both to detect transient faults and to improve performance by preexecuting problematic instruction sequences to resolve branches and issue prefetches to memory.

Many of these techniques have already been adopted in real systems; many others exist only as research proposals. Future designs are likely to adopt at least some of the proposed techniques to overcome many of the implementation challenges associated with building high-throughput, high-frequency, and power-efficient computer systems.

REFERENCES

Adve, S. V., and K. Gharachorloo: "Shared memory consistency models: A tutorial," *IEEE Computer,* 29, 12, 1996, pp. 66–76.

Agarwal, A., B. Lim, D. Kranz, and J. Kubiatowicz: "APRIL: a processor architecture for multiprocessing," *Proc. ISCA-17,* 1990, pp. 104–114.

Akkary, H., and M. A. Driscoll: "A dynamic multithreading processor," *Proc. 31st Annual Int. Symposium on Microarchitecture,* 1998, pp. 226–236.

Austin, T.: "DIVA: A reliable substrate for deep-submicron processor design," *Proc. 32nd Annual ACM/IEEE Int. Symposium on Microarchitecture (MICRO-32),* Los Alamitos, IEEE Computer Society, 1999.

Burger, D., S. Kaxiras, and J. Goodman: "Datascalar architectures," *Proc. 24th Int. Symposium on Computer Architecture,* 1997, pp. 338–349.

Censier, L., and P. Feautrier: "A new solution to coherence problems in multicache systems," *IEEE Trans. on Computers,* C-27, 12, 1978, pp. 1112–1118.

Charlesworth, A.: "Starfire: extending the SMP envelope," *IEEE MICRO,* vol. 18 no. 1, 1998, pp. 39–49.

Collins, J., H. Wang, D. Tullsen, C. Hughes, Y. Lee, D. Lavery, and J. Shen: "Speculative precomputation: Long-range prefetching of delinquent loads," *Proc. 28th Annual Int. Symposium on Computer Architecture,* 2001, pp. 14–25.

Eickemeyer, R. J., R. E. Johnson, S. R. Kunkel, M. S. Squillante, and S. Liu: "Evaluation of multithreaded uniprocessors for commercial application environments," *Proc. 23rd Annual Int. Symposium on Computer Architecture,* Philadelphia, ACM SIGARCH and IEEE Computer Society TCCA, 1996, pp. 203–212.

Fillo, M., S. Keckler, W. Dally, and N. Carter: "The M-Machine multicomputer," *Proc. 28th Annual Int. Symposium on Microarchitecture (MICRO-28),* 1995, pp. 146–156.

Franklin, M.: "The multiscalar architecture," Ph.D. thesis, University of Wisconsin-Madison, 1993.

Hammond, L., M. Willey, and K. Olukotun: "Data speculation support for a chip-multiprocessor," *Proc. 8th Symposium on Architectural Support for Programming Languages and Operating Systems,* 1998, pp. 58–69.

Hill, M.: "Multiprocessors should support simple memory consistency models," *IEEE Computer,* 31, 8, 1998, pp. 28–34.

Krishnan, V., and J. Torrellas: "The need for fast communication in hardware-based speculative chip multiprocessors," *Int. Journal of Parallel Programming,* 29, 1, 2001, pp. 3–33.

Lamport, L.: "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Trans. on Computers,* C-28, 9, 1979, pp. 690–691.

Lovett, T., and R. Clapp: "STiNG: A CC-NUMA Computer System for the Commercial Marketplace," *Proc. 23rd Annual Int. Symposium on Computer Architecture,* 1996, pp. 308–317.

Olukotun, K., B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang: "The case for a single-chip multiprocessor," *Proc. 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII),* 1996, pp. 2–11.

Rotenberg, E.: "AR-SMT: A microarchitectural approach to fault tolerance in microprocessors," *Proc. 29th Fault-Tolerant Computing Symposium,* 1999, pp. 84–91.

Roth, A.: "Pre-execution via speculative data-driven multithreading," Ph.D. Thesis, University of Wisconsin, Madison, WI, 2001.

Smith, B.: "Architecture and applications of the HEP multiprocessor computer system," *Proc. Int. Society for Optical Engineering,* 1991, pp. 241–248.

Sohi, G., S. Breach, and T. Vijaykumar: "Multiscalar processors," *Proc. 22nd Annual Int. Symposium on Computer Architecture,* 1995, pp. 414–425.

Steffan, J., C. Colohan, and T. Mowry: "Architectural support for thread-level data speculation," Technical report, School of Computer Science, Carnegie Mellon University, 1997.

Steffan, J. G., C. Colohan, A. Zhai, and T. Mowry: "A scalable approach to thread-level speculation," *Proc. 27th Int. Symposium on Computer Architecture,* 2000.

Steffan, J. G., and T. C. Mowry: "The potential for using thread-level data speculation to facilitate automatic parallelization," *Proc. of HPCA,* 1998, pp. 2–13.

Storino, S., A. Aipperspach, J. Borkenhagen, R. Eickemeyer, S. Kunkel, S. Levenstein, and G. Uhlmann: "A commercial multi-threaded RISC processor," *Int. Solid-State Circuits Conference,* 1998.

Sundaramoorthy, K., Z. Purser., and E., Rotenberg: "Slipstream processors: Improving both performance and fault tolerance," *Proc. 9th Int. Conf. on Architectural Support for Programming Languages and Operating Systems,* 2000, pp. 257–268.

Tendler, J. M., S. Dodson, S. Fields, and B. Sinharoy: "IBM eserver POWER4 system microarchitecture," IBM Whitepaper, 2001.

Tera Computer Company: "Hardware characteristics of the Tera MTA," 1998.

Thornton, J. E.: "Parallel operation in the Control Data 6600," *AFIPS Proc. FJCC,* part 2, 26, 1964, pp. 33–40.

Tullsen, D., S. Eggers, J. Emer, H. Levy, J. Lo, and R. Stamm: "Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor," *Proc. 23rd Annual Symposium on Computer Architecture,* 1996, pp. 191–202.

Tullsen, D. M.: "Simultaneous multithreading," Ph.D. Thesis, University of Washington, Seattle, WA, 1996.

Uht, A. K., and V. Sindagi: "Disjoint eager execution: An optimal form of speculative execution," *Proc. 28th Annual ACM/IEEE Int. Symposium on Microarchitecture,* 1995, pp. 313–325.

Wang, W.-H., J.-L. Baer, and H. Levy: "Organization and performance of a two-level virtual-real cache hierarchy," *Proc. 16th Annual Int. Symposium on Computer Architecture,* 1989, pp. 140–148.

Yeager, K.: "The MIPS R10000 superscalar microprocessor," *IEEE Micro,* 16, 2, 1996, pp. 28–40.

Zilles, C.: "Master/slave speculative parallelization and approximate code," Ph.D. Thesis, University of Wisconsin, Madison, WI, 2002.

HOMEWORK PROBLEMS

P11.1 Using the syntax in Figure 11.2, show how to use the load-linked/store conditional primitives to synthesize a compare-and-swap operation.

P11.2 Using the syntax in Figure 11.2, show how to use the load-linked/store conditional primitives to acquire a lock variable before entering a critical section.

P11.3 A processor such as the PowerPC G3, widely deployed in Apple Macintosh systems, is primarily intended for use in uniprocessor systems, and hence has a very simple MEI cache coherence protocol. Identify and discuss one reason why even a uniprocessor design should support cache coherence. Is the MEI protocol of the G3 adequate for this purpose? Why or why not?

P11.4 Apple marketed a G3-based dual-processor system that was mostly used for running asymmetric workloads. In other words, the second processor was only used to execute parts of specific applications, such as Adobe Photoshop, rather than being used in a symmetric manner by the operating system to execute any ready thread or process. Assuming

a multiprocessor-capable operating system (which the MacOS, at the time, was not), explain why symmetric use of a G3-based dual-processor system might result in very poor performance. Propose a software solution implemented by the operating system that would mitigate this problem, and explain why it would help.

P11.5 Given the MESI protocol described in Figure 11.5, create a similar specification (state table and diagram) for the much simpler MEI protocol. Comment on how much easier it would be to implement this protocol.

P11.6 Many modern systems use a MOESI cache coherence protocol, where the semantics of the additional O state are that the line is shared-dirty: i.e., multiple copies may exist, but the other copies are in the S state, and the cache that has the line in the O state is responsible for writing the line back if it is evicted. Modify the table and state diagram shown in Figure 11.5 to include the O state.

P11.7 Explain what benefit accrues from the addition of the O state to the MESI protocol.

P11.8 Real coherence controllers include numerous transient states in addition to the ones shown in Figure 11.5 to support split-transaction busses. For example, when a processor issues a bus read for an invalid line (I), the line is placed in an IS transient state until the processor has received a valid data response that then causes the line to transition into the shared state (S). Given a split-transaction bus that separates each bus command (bus read, bus write, and bus upgrade) into a request and response, augment the state table and state transition diagram of Figure 11.5 to incorporate all necessary transient states and bus responses. For simplicity, assume that any bus command for a line in a transient state gets a negative acknowledge (NAK) response that forces it to be retried after some delay.

P11.9 Given Problem 11.8, further augment Figure 11.5 to eliminate at least three NAK responses by adding necessary additional transient states. Comment on the complexity of the resulting coherence protocol.

P11.10 Assuming a processor frequency of 1 GHz, a target CPI of 2, a per-instruction level-2 cache miss rate of 1% per instruction, a snoop-based cache coherent system with 32 processors, and 8-byte address messages (including command and snoop addresses), compute the inbound and outbound snoop bandwidth required at each processor node.

P11.11 Given the assumptions of Problem 11.10, assume you are planning an enhanced system with 64 processors. The current level-2 cache design has a single-ported tag array with a lookup latency of 3 ns. Will the 64-processor system have adequate snoop bandwidth? If not, describe an alternative design that will.

P11.12 Using the equation in Section 11.3.5, compute the average memory latency for a three-level hierarchy where hits in the level-1 cache take one

Ex.1019 / Page 63 of 66
TESLA, INC.

cycle, hits in the level-2 cache take 12 cycles, hits in the level-3 cache take 50 cycles, and misses to memory take 250 cycles. Assume a level-1 miss rate of 5% misses per program reference, a level-2 miss rate of 2% per program reference, and a level-3 miss rate of 0.5% per program reference.

P11.13 Given the assumptions of Problem 11.12, compute the average memory latency for a system with no level-3 cache and only 200 cycle latency to memory (since the level-3 lookup is no longer performed before initiating the fetch from memory). Which system performs better? What is the breakeven miss rate per program reference for the two systems (i.e., the level-3 miss rate at which both systems provide the same performance)?

P11.14 Assume a processor similar to the Hewlett-Packard PA-8500, with only a single level of data cache. Assume the cache is virtually indexed but physically tagged, is four-way associative with 128-byte lines, and is 512 KB in size. In order to snoop coherence messages from the bus, a reverse-address translation table is used to store physical-to-virtual address mappings stored in the cache. Assuming a fully associative reverse-address translation table and 4K-byte pages, how many entries must it contain so that it can map the entire data cache?

P11.15 Given the assumptions of Problem 11.14, describe a reasonable set-associative organization for the RAT that is still able to map the entire data cache.

P11.16 Given the assumptions of Problem 11.14, explain the implications of a reverse-address translation table that is not able to map all possible entries in the data cache. Describe the sequence of events that must occur whenever a reverse-address translation table entry is displaced due to replacement.

Problems 17 through 19

In a two-level cache hierarchy, it is often convenient to maintain inclusion between the primary cache and the secondary cache. A common mechanism for tracking inclusion is for the level-2 cache to maintain presence bits for each level-2 directory entry that indicate the line is also present in the level-1 cache. Given the following assumptions, answer the following questions:

- Presence bit mechanism for maintaining inclusion
- 4K virtual memory page size
- Physically indexed, physically tagged 2-Mbyte eight-way set-associative cache with 64-byte lines

P11.17 Given a 32K-byte eight-way set-associative level-1 data cache with 32-byte lines, outline the steps that the level-2 controller must follow whenever it removes a cache line from the level-2 cache. Be specific, explain each step, and make sure the level-2 controller has the information it needs to complete each step.

P11.18 Given a virtually indexed, physically tagged 16K-byte direct-mapped level-1 data cache with 32-byte lines, how does the level-2 controller's job change?

P11.19 Given a virtually indexed, virtually tagged 16K-byte direct-mapped level-1 data cache with 32-byte lines, are presence bits still a reasonable solution or is there a better one? Why or why not?

P11.20 Figure 11.8 explains read-set tracking as used in high-performance implementations of sequentially consistent multiprocessors. As shown, a potential ordering violation is detected by snooping the load queue and refetching a marked load when it attempts to commit. Explain why the processor should not refetch right away, as soon as the violation is detected, instead of waiting for the load to commit.

P11.21 Given the mechanism referenced in Problem 11.20, false sharing (where a remote processor writes the lower half of a cache line, but the local processor reads the upper half) can cause additional pipeline refetches. Propose a hardware scheme that would eliminate such refetches. Quantify the hardware cost of such a scheme.

P11.22 Given Problem 11.21, describe a software approach that would derive the same benefit.

P11.23 A chip multiprocessor (CMP) implementation enables interesting combinations of on-chip and off-chip coherence protocols. Discuss all combinations of the following coherence protocols and implementation approaches and their relative advantages and disadvantages. On-chip, consider update and invalidate protocols, implemented with snooping and directories. Off-chip, consider invalidate protocols, implemented with snooping and directories. Which combinations make sense? What are the tradeoffs?

P11.24 Assume that you are building a fine-grained multithreaded processor similar to the Tera MTA that masks memory latency with a large number of concurrently active threads. Assuming your processor supports 100 concurrently active threads to mask a memory latency of one hundred 1-ns processor cycles. Further assume that you are using conventional DRAM chips to implement your memory subsystem. Assume the DRAM chips you are using have a 30-ns command occupancy, i.e., each command (read or write) occupies the DRAM chip interface for 30 ns. Compute the minimum number of independent DRAM chip interfaces your memory controller must provide to prevent your processor from stalling by turning around a DRAM request for every processor cycle.

P11.25 Assume what is described in Problem 11.24. Further, assume your DRAM chips support page mode, where sequential accesses of 8 bytes each can be made in only 10 ns. That is, the first access requires 30 ns, but subsequent accesses to the same 512-byte page can be satisfied in 10 ns. The scientific workloads your processor executes tend to perform unit stride

accesses to large arrays. Given this memory reference behavior, how many independent DRAM chip interfaces do you need now to prevent your processor from stalling?

P11.26 Existing coarse-grained multithreaded processors such as the IBM Northstar and Pulsar processors only provide in-order execution in the core. Explain why or why not coarse-grained multithreading would be effective with a processor that supports out-of-order execution.

P11.27 Existing simultaneous multithreaded processors such as the Intel Pentium 4 also support out-of-order execution of instructions. Explain why or why not simultaneous multithreading would be effective with an in-order processor.

P11.28 An IMT design with distributed processing elements (e.g., multiscalar or TLS) must perform some type of load balancing to ensure that each processing element is doing roughly the same amount of work. Discuss hardware- and software-based load balancing schemes and comment on which might be most appropriate for both multiscalar and TLS.

P11.29 An implicit multithreaded processor such as the proposed DMT design must insert instructions into the reorder buffer out of program order. This implies a complex free-list management scheme for tracking the available entries in the reorder buffer. The physical register file that is used in existing out-of-order processors also requires a similar free-list management scheme. Comment on how DMT ROB management differs, if at all, from free-list management for the physical register file. Describe such a scheme in detail, using a diagram and pseudocode that implements the management algorithm.

P11.30 The DEE proposal appears to rely on fairly uniform branch prediction rates for its limited eager execution to be effective. Describe what happens if branch mispredictions are clustered in a nonuniform distribution (i.e., a mispredicted branch is likely to be followed by one or more other mispredictions). What happens to the effectiveness of this approach? Use an example to show whether or not DEE will still be effective.

P11.31 A recent study shows that the TLS architecture benefits significantly from silent stores. Silent stores are store instructions that write a value to memory that is already stored at that location. Create a detailed sample execution that shows how detecting and eliminating a silent store can substantially improve performance in a TLS system.

P11.32 Preexecution of conditional branches in a redundant runahead thread allows speedy resolution of mispredicted branches, as long as branch instances from both threads are properly synchronized. Propose a detailed design that will keep the runahead thread and the main thread synchronized for this purpose. Identify the design challenges and quantify the cost of such a hardware unit.