MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS

Kourosh Gharachorloo

Technical Report: CSL-TR-95-685

December 1995

This research has been supported by DARPA contract N00039-91-C-0138.
Author also acknowledges support from Digital Equipment Corporation.

Memory Consistency Models for Shared-Memory Multiprocessors

Kourosh Gharachorloo

Technical Report: CSL-TR-95-685

December 1995

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Gates Building A-408
Stanford, CA 94305-9040
pubs@shasta.stanford.edu

Abstract

The memory consistency model for a shared-memory multiprocessor specifies the behavior of memory with respect to read and write operations from multiple processors. As such, the memory model influences many aspects of system design, including the design of programming languages, compilers, and the underlying hardware. *Relaxed models* that impose fewer memory ordering constraints offer the potential for higher performance by allowing hardware and software to overlap and reorder memory operations. However, fewer ordering guarantees can compromise programmability and portability. Many of the previously proposed models either fail to provide reasonable programming semantics or are biased toward programming ease at the cost of sacrificing performance. Furthermore, the lack of consensus on an acceptable model hinders software portability across different systems.

This dissertation focuses on providing a balanced solution that directly addresses the trade-off between *programming ease* and *performance*. To address programmability, we propose an alternative method for specifying memory behavior that presents a higher level abstraction to the programmer. We show that with only a few types of information supplied by the programmer, an implementation can exploit the full range of optimizations enabled by previous models. Furthermore, the same information enables automatic and efficient *portability* across a wide range of implementations.

To expose the optimizations enabled by a model, we have developed a formal framework for specifying the low-level ordering constraints that must be enforced by an implementation. Based on these specifications, we present a wide range of architecture and compiler implementation techniques for efficiently supporting a given model. Finally, we evaluate the performance benefits of exploiting relaxed models based on detailed simulations of realistic parallel applications. Our results show that the optimizations enabled by relaxed models are extremely effective in hiding virtually the full latency of writes in architectures with blocking reads (i.e., processor stalls on reads), with gains as high as 80%. Architectures with non-blocking reads can further exploit relaxed models to hide a substantial fraction of the read latency as well, leading to a larger overall performance benefit. Furthermore, these optimizations complement gains from other latency hiding techniques such as prefetching and multiple contexts.

We believe that the combined benefits in hardware and software will make relaxed models universal in future multiprocessors, as is already evidenced by their adoption in several commercial systems.

Key Words and Phrases: shared-memory multiprocessors, memory consistency models, latency hiding techniques, latency tolerating techniques, relaxed memory models, sequential consistency, release consistency

Acknowledgements

Many thanks go to my advisors John Hennessy and Anoop Gupta for their continued guidance, support, and encouragement. John Hennessy's vision and enthusiasm have served as an inspiration since my early graduate days at Stanford. He has been a great source of insight and an excellent sounding board for ideas. His leadership was instrumental in the conception of the DASH project which provided a great infrastructure for multiprocessor research at Stanford. Anoop Gupta encouraged me to pursue my early ideas on memory consistency models. I am grateful for the tremendous time, energy, and wisdom that he invested in steering my research. He has been an excellent role model through his dedication to quality research. I also thank James Plummer for graciously serving as my orals committee chairman and my third reader.

I was fortunate to be among great friends and colleagues at Stanford. In particular, I would like to thank the other members of the DASH project, especially Jim Laudon, Dan Lenoski, and Wolf-Dietrich Weber, for making the DASH project an exciting experience. I thank the following people for providing the base simulation tools for my studies: Steve Goldschmidt for TangoLite, Todd Mowry for Dixie, and Mike Johnson for the dynamic scheduled processor simulator. Charles Orgish, Laura Schrager, and Thoi Nguyen at Stanford, and Annie Warren and Jason Wold at Digital, were instrumental in supporting the computing environment. I also thank Margaret Rowland and Darlene Hadding for their administrative support. I thank Vivek Sarkar for serving as a mentor during my first year at Stanford. Rohit Chandra, Dan Scales, and Ravi Soundararajan get special thanks for their help in proof reading the final version of the thesis. Finally, I am grateful to my friends and office mates, Paul Calder, Rohit Chandra, Jaswinder Pal Singh, and Mike Smith, who made my time at Stanford most enjoyable.

My research on memory consistency models has been enriched through collaborations with Phil Gibbons and Sarita Adve. I have also enjoyed working with Andreas Nowatzyk on the Sparc V9 RMO model, and with Jim Horning, Jim Saxe, and Yuan Yu on the Digital Alpha memory model.

I thank Digital Equipment Corporation, the Western Research Laboratory, and especially Joel Bartlett and Richard Swan, for giving me the freedom to continue my work in this area after joining Digital. The work presented in this thesis represents a substantial extension to my earlier work at Stanford.

I would like to thank my friends and relatives, Ali, Farima, Hadi, Hooman, Illah, Rohit, Sapideh, Shahin, Shahrzad, Shervin, Siamak, Siavosh, and Sina, who have made these past years so enjoyable. Finally, I thank my family, my parents and brother, for their immeasurable love, encouragement, and support of my education. Most importantly, I thank my wife, Nazhin, who has been the source of happiness in my life.

To my parents Nezhat and Vali
and my wife Nazhin

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Parallel architectures provide the potential for achieving substantially higher performance than traditional uniprocessor architectures. By utilizing the fastest available microprocessors, multiprocessors are increasingly becoming a viable and cost-effective technology even at small numbers of processing nodes.

The key differentiating feature among multiprocessors is the mechanisms used to support communication among different processors. Message-passing architectures provide each processor with a local memory that is accessible only to that processor and require processors to communicate through explicit messages. In contrast, multiprocessors with a single address space, such as shared-memory architectures, make the entire memory accessible to all processors and allow processors to communicate directly through read and write operations to memory.

The single address space abstraction greatly enhances the programmability of a multiprocessor. In comparison to a message-passing architecture, the ability of each processor to access the entire memory simplifies programming by reducing the need for explicit data partitioning and data movement. The single address space also provides better support for parallelizing compilers and standard operating systems. These factors make it substantially easier to develop and incrementally tune parallel applications.

Since shared-memory systems allow multiple processors to simultaneously read and write the same memory locations, programmers require a conceptual model for the semantics of memory operations to allow them to correctly use the shared memory. This model is typically referred to as a *memory consistency model* or *memory model*. To maintain the programmability of shared-memory systems, such a model should be intuitive and simple to use. Unfortunately, architecture and compiler optimizations that are required for efficiently supporting a single address space often complicate the memory behavior by causing different processors to observe distinct views of the shared memory. Therefore, one of the challenging problems in designing a shared-memory system is to present the programmer with a view of the memory system that is easy to use and yet allows the optimizations that are necessary for efficiently supporting a single address space.

Even though there have been numerous attempts at defining an appropriate memory model for shared-memory systems, many of the proposed models either fail to provide reasonable programming semantics or are biased toward programming ease at the cost of sacrificing performance. In addition, the lack of consensus on an acceptable model, along with subtle yet important semantic differences among the various models, hinder simple and efficient portability of programs across different systems. This thesis focuses on providing a balanced solution that directly addresses the trade-off between *programming ease* and *performance*. Furthermore, our solution provides automatic *portability* across a wide range of implementations.

1.1 The Problem

Uniprocessors present a simple and intuitive view of memory to programmers. Memory operations are assumed to execute one at a time in the order specified by the program and a read is assumed to return the value of the last write to the same location. However, an implementation does not need to directly maintain this order among all memory operations for correctness. The illusion of sequentiality can be maintained by *only* preserving the sequential order among memory operations to the *same location*. This flexibility to overlap and reorder operations to different locations is exploited to provide efficient uniprocessor implementations. To hide memory latency, architectures routinely use optimizations that overlap or pipeline memory operations and allow memory operations to complete out-of-order. Similarly, compilers use optimizations such as code motion and register allocation that exploit the ability to reorder memory operations. In summary, the uniprocessor memory model is simple and intuitive for programmers and yet allows for high performance implementations.

Allowing multiple processors to concurrently read and write a set of common memory locations complicates the behavior of memory operations in a shared-memory multiprocessor. Consider the example code segment shown in Figure 1.1 which illustrates a producer-consumer interaction between two processors. As shown, the first processor writes to location A and synchronizes with the second processor by setting location Flag, after which the second processor reads location A. The intended behavior of this producer-consumer interaction is for the read of A to return the new value of 1 in every execution. However, this behavior may be easily violated in some systems. For example, the read of A may observe the old value of 0 if the two writes on the first processor are allowed to execute out of program order. This simple example illustrates the need for clearly specifying the behavior of memory operations supported by a shared-memory system.[1]

Since a multiprocessor is conceptually a collection of uniprocessors sharing the same memory, it is natural to expect its memory behavior to be a simple extension of that of a uniprocessor. The intuitive memory model assumed by most programmers requires the execution of a parallel program on a multiprocessor to appear as some interleaving of the execution of the parallel processes on a uniprocessor. This intuitive model was formally defined by Lamport as *sequential consistency* [Lam79]:

Definition 1.1: Sequential Consistency
[A multiprocessor is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Referring back to Figure 1.1, sequential consistency guarantees that the read of A will return the newly produced value in all executions. Even though sequential consistency provides a simple model to programmers, the restrictions it places on implementations can adversely affect efficiency and performance. Since several

[1] Even though we are primarily interested in specifying memory behavior for systems with multiple processors, similar issues arise in a uniprocessor system that supports a single address space across multiple processes or threads.

```
Initially  A = FLAG = 0

  P1                P2

A = 1;
FLAG = 1;          while (FLAG == 0);
                   ... = A;
```

Figure 1.1: Example producer-consumer interaction.

processors are allowed to concurrently access the same memory locations, reordering memory operations on one processor can be easily detected by another processor. Therefore, simply preserving the program order on a per-location basis, as is done in uniprocessors, is not sufficient for guaranteeing sequential consistency. A straightforward implementation of sequential consistency must disallow the reordering of shared memory operations from each processor. Consequently, many of the architecture and compiler optimizations used in uniprocessors are not safely applicable to sequentially consistent multiprocessors. Meanwhile, the high latency of memory operations in multiprocessors makes the use of such optimizations even more important than in uniprocessors.

To achieve better performance, alternative memory models have been proposed that relax some of the memory ordering constraints imposed by sequential consistency. The seminal model among these is the *weak ordering* model proposed by Dubois et al. [DSB86]. Weak ordering distinguishes between ordinary memory operations and memory operations used for synchronization. By ensuring consistency only at the synchronization points, weak ordering allows ordinary operations in between pairs of synchronizations to be reordered with respect to one another.

The advantage of using a *relaxed memory model* such as weak ordering is that it enables many of the uniprocessor optimizations that require the flexibility to reorder memory operations. However, a major drawback of this approach is that programmers can no longer assume a simple serial memory semantics. This makes reasoning about parallel programs cumbersome because the programmer is directly exposed to the low-level memory reorderings that are allowed by a relaxed model. Therefore, while relaxed memory models address the performance deficiencies of sequential consistency, they may unduly compromise programmability. Programming complexity is further exacerbated by the subtle semantic differences among the various relaxed models which hinders efficient portability of programs across different systems.

1.2 Our Approach

The trade-offs between performance, programmability, and portability present a dilemma in defining an appropriate memory consistency model for shared-memory multiprocessors. Choosing an appropriate model requires considering three important factors. First, we need to determine how the model is presented to the programmer and how this impacts programming ease and portability. Second, we need to specify the restrictions the model places on the system and determine techniques required for correctly and efficiently implementing the model. Finally, we need to evaluate the performance of the model and consider the implementation complexity that is necessary to achieve this performance. We discuss our approach for addressing these issues below.

1.2.1 Programming Ease and Portability

Relaxed memory models such as weak ordering are specified as a set of conditions that define the allowable orderings among memory operations. Such specifications can be considered *system-centric* since they in effect expose the low-level system optimizations to the programmer. To address programmability, we have developed an alternative framework for specifying relaxed memory models that presents the programmer with a higher-level abstraction of the system. We refer to this type of specification as *programmer-centric*.

The premise behind a programmer-centric specification is that programmers prefer to reason with a simple memory model such as sequential consistency. Therefore, in contrast to the system-centric approach, the programmer is allowed to use sequential consistency to reason about programs. To enable optimizations, the programmer is required to provide information about the behavior of memory operations in sequentially consistent executions of the program. This information allows the system to determine the memory ordering optimizations that can be safely exploited without violating sequential consistency. Furthermore, the same information can be used to automatically and efficiently port the program to a wide range of implementations. Overall, providing this type of program-level information is simpler and more natural for programmers than directly reasoning with system-centric models.

In this thesis, we present a set of programmer-centric models that unify the optimizations captured by previous system-centric models. We show that this approach is easier to use and provides a higher performance potential than previous system-centric models.

1.2.2 Implementation Issues

To better understand the implementation needs of a memory consistency model, we must first specify the precise conditions that an implementation must obey to satisfy the semantics of the model. To address this issue, we have developed a formal framework for specifying the low-level system conditions imposed by a model. Specifications in this framework have the advantage of being formal and precise, and naturally expose the system optimizations that are allowed by a model. We use the above framework to express the sufficient conditions for implementing the programmer-centric models described in the previous section, in addition to the conditions for several of the system-centric models.

Based on the above specifications, we next consider a wide range of architecture and compiler techniques for correctly and efficiently supporting a given memory model.

1.2.3 Performance Evaluation

The primary motivation for studying relaxed memory models is to achieve higher performance. Evaluating the performance of such models is critical in deciding whether the benefits of a model outweigh its costs in terms of programming and implementation complexity. In addition, such a study sheds light on the features in a model that have the most impact on performance. Surprisingly, numerous relaxed memory models have been proposed without any attempt to quantify their performance effects.

We focus on evaluating performance gains from exploiting relaxed memory models in cache-coherent shared-memory architectures using a set of realistic parallel programs. By studying implementations with varying degrees of complexity, we identify the key architectural features for efficiently supporting relaxed memory models and relate the performance gains to the level of implementation complexity. Our results show that with a modest level of architectural support, relaxed memory models are extremely effective in hiding the large latency of memory operations in shared-memory machines.

1.3 Contributions

The primary contributions of this dissertation are as follows:

- We propose a set of programmer-centric models that unify the optimizations enabled by system-centric models. Compared to the system-centric models, the programmer-centric models are superior in their programming ease, portability, and performance potential. The models presented in this thesis extend our previous work on the *release consistency* model and *properly labeled* programs [GLL+90, GAG+92].

- We present a formal framework for specifying the implementation conditions that are required to maintain the semantics of a memory model. This framework leads to precise specifications that can be easily translated into correct and efficient implementations of both the architecture and the compiler. The framework presented in this thesis extends our previous work in this area [GGH93b, GAG+93]. We use it to specify the sufficient conditions for implementing our programmer-centric models, as well as to specify the conditions for several system-centric memory models. Expressing these models within a uniform and precise framework allows for a direct comparison of the optimizations enabled by each model. Furthermore, based on these specifications, we determine transformations for automatically porting programs across different models.

- We present a variety of practical implementation techniques for efficiently supporting memory models at the architecture and compiler level. Several of the architectural techniques discussed here have been successfully used in the design of the Stanford DASH [LLG+92] and FLASH [KOH+94] multiprocessors. We also describe a couple of aggressive techniques that can significantly boost the performance of models such as sequential consistency relative to straightforward hardware implementations of such models [GGH91b]. These latter techniques have recently been adopted by several commercial processors, including the Intel Pentium Pro (or P6) and the MIPS R10000.

- We provide a detailed performance evaluation of relaxed memory models at the architecture level. Our earlier studies in this area [GGH91a, GGH92] represent the first set of comprehensive results to quantify the performance gains from exploiting relaxed memory models.

1.4 Organization

Chapter 2 presents the background material for the thesis. We begin by describing the various components of a multiprocessor system that are affected by the memory consistency model. We next discuss the implications of sequential consistency on both architecture and compiler optimizations. Finally, we provide an overview of the previously proposed memory consistency models.

Chapter 3 presents the programmer-centric approach for specifying memory models. We discuss the abstraction that is presented to the programmer and describe the type of information that the programmer is required to convey about memory operations.

Chapter 4 presents the framework for specifying system requirements for different memory models. This framework is used to specify sufficient conditions for implementing the programmer-centric models proposed in Chapter 3. In addition, we provide specifications for several of the system-centric models and describe how programs that are written for programmer-centric models can be automatically ported to such models.

Chapter 5 presents a wide range of implementation techniques for efficiently supporting memory consistency models at both the architecture and the compiler level.

Chapter 6 presents a detailed evaluation of the performance gains from exploiting relaxed memory models. We identify the key optimizations and architectural enhancements that are necessary for achieving these gains in cache-coherent shared-memory architectures. In addition, we study the interaction of relaxed memory models with other latency hiding techniques such as prefetching and use of multiple contexts.

Finally, Chapter 7 summarizes the major results of this dissertation and discusses potential directions for future research.

Chapter 2

Background

This chapter provides the background information on memory consistency models. The first section describes the role of memory consistency models in specifying the behavior of memory operations and introduces sequential consistency as an intuitive model for shared-memory systems. We next consider the impact of common architecture and compiler optimizations on the behavior of shared memory operations. The third section illustrates the inherent difficulties in exploiting these optimizations in conjunction with sequential consistency. Finally, we explore some of the alternative models that have been proposed to remedy the performance inefficiencies associated with sequential consistency.

2.1 What is a Memory Consistency Model?

A *memory consistency model*, or *memory model*, for a shared-memory multiprocessor specifies how memory behaves with respect to read and write operations from multiple processors. From the programmer's point of view, the model enables correct reasoning about the memory operations in a program. From the system designer's point of view, the model specifies acceptable memory behaviors for the system. As such, the memory consistency model influences many aspects of system design, including the design of programming languages, compilers, and the underlying hardware.

This section begins with a brief description of the interfaces at which a memory model may be defined. We next present some basic terminology and assumptions for describing memory models. This is followed by a description of the sequential consistency model along with numerous examples that provide further familiarization with this model. Finally, we describe how different models or implementations may be compared based on the memory behaviors they allow.

2.1.1 Interface between Programmer and System

A memory model can be defined at any interface between the programmer and the system. Figure 2.1 shows a variety of possible interfaces in a typical multiprocessor system. At the lowest interface, the system consists

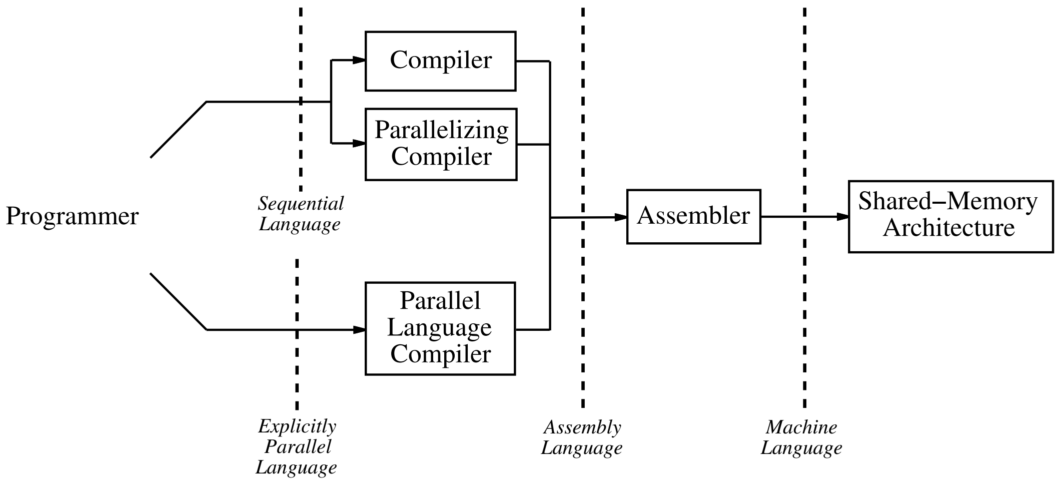


Figure 2.1: Various interfaces between programmer and system.

of the base hardware and programmers express their programs in machine-level instructions. At higher level interfaces, the system is typically composed of more components, including the compiler, assembler, and base architecture.

The definition of the memory model is often incorporated into the semantics of the programming language used at a given interface. Therefore, the memory model may vary for different interfaces or for different languages at the same interface. An intermediate system component such as the compiler is responsible for correctly mapping the memory semantics of its input language to that of its output language.

The majority of programmers deal with the highest interface, using a high-level programming language for specifying programs. At this level, a programmer typically has the choice of using either a traditional sequential language or an explicitly parallel language. The sequential language interface provides the familiar uniprocessor semantics for memory, thus relieving the programmer from reasoning about the multiprocessor aspects of the system. This interface is typically supported either by a traditional sequential compiler that generates a single thread of control to execute on the multiprocessor or by a parallelizing compiler that deals with the multiprocessor issues transparently to the programmer. On the other hand, programmers who opt to express their algorithms more directly using an explicitly parallel language are exposed to the multiprocessor semantics for memory operations in one form or another.

Many of the concepts that are discussed in this thesis apply across all interfaces. We will explicitly point out any concepts that are interface-specific.

2.1.2 Terminology and Assumptions

This section presents some of the basic terminology and assumptions we use to describe multiprocessor systems and memory consistency models. Our goal is to provide intuitive definitions; a more formal and general framework for describing memory models will be presented in Chapter 4.

A multiprocessor *program* conceptually consists of several parallel threads of control, each with its individual program consisting of a sequence of instructions. Since we are primarily concerned with the behavior of memory operations, our focus will be on instructions that generate memory accesses to shared data.

We assume a canonical multiprocessor system that consists of several *virtual processors* sharing a common memory. To execute a multiprocessor program, we assume there is a virtual processor per parallel thread of control in the application. Therefore, the semantics of a memory model should not be affected by the physical mapping of multiple threads to the same physical processor.

An *execution* consists of a sequence of shared memory operations that is generated as a result of executing the multiprocessor program. A *memory operation* corresponds to a specific instance of executing a memory instruction. Initializing the state of memory may be modeled by a set of initial write operations to memory that occur before the program execution begins.

We assume processors communicate with one another solely through read and write memory operations to the shared data memory. A *read* operation specifies a memory location and results in a single read event that returns the value of that memory location. Similarly, a *write* operation specifies a memory location along with a new value and results in a single write event that modifies the memory location to the new value. Two operations *conflict* if they are to the same location and at least one of them is a write [SS88]. Except for the events described above, there are no other side effects due to a read or write operation. For simplicity, we assume the instruction space is read only; Chapter 4 discusses how this assumption may be removed.

In general, an instruction may specify more than a single memory operation. Multiple memory operations generated by the execution of an instruction should appear to execute *atomically*; for any two instructions *i1* and *i2*, either all operations of *i1* appear to execute before any operation of *i2* or vice versa. Each instruction must also specify the order in which its constituent operations must execute. The above generality is useful for modeling read and write memory operations to multiple granularities of data (e.g., byte, word, double word) and for modeling atomic read-modify-write operations (e.g., test-and-set, compare-and-swap). For simplicity, we assume instructions can generate a single read operation, a single write operation, or a read followed by a write operation to the same location, all with the same data granularity. Extensions to this assumption will be discussed in Chapter 4.

The program for each virtual processor imposes a conceptual total order on the operations issued by the processor in a given execution. The *program order* is defined as a partial order on all memory operations that is consistent with the per-processor total order on memory operations from each processor [SS88].

Finally, we need to discuss the notion of *result* or *outcome* for an execution of a program. We will assume that the result of an execution comprises of the values returned by the read operations in the execution. The final state of memory can be included in this notion by modeling it through read operations to memory that occur after the actual program execution completes. The above notion of result does not capture the time or actual order in which operations execute; we can only deduce the order in which operations *appear* to execute. Therefore, the results of two executions are considered *equivalent* if the reads return the same values in both executions, regardless of whether the actual order of operations is the same in the two executions. Chapter 4 will discuss more realistic notions of result that include the effect of I/O devices.

2.1.3 Sequential Consistency

A natural way to define a memory model for multiprocessors is to base it on the sequential semantics of memory operations in uniprocessors. An intuitive definition would require executions of a parallel program on a multiprocessor to behave the same as some interleaved execution of the parallel processes on a uniprocessor.

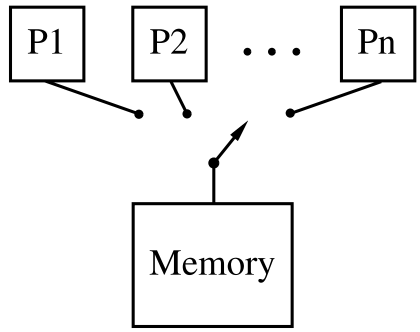


Figure 2.2: Conceptual representation of sequential consistency (SC).

Such a model was formally defined by Lamport as *sequential consistency* [Lam79] (abbreviated as SC). The definition below assumes a multiprocessor consists of several correctly-functioning uniprocessors, referred to as *sequential* processors [Lam79], that access a common memory. Another implicit assumption is that a read returns the value of the last write to the same location that is before it in the sequential order described above.

> **Definition 2.1: Sequential Consistency**
> [A multiprocessor is sequentially consistent if] the result of any execution *is the same as if* the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Sequential consistency maintains the memory behavior that is intuitively expected by most programmers. Figure 2.2 provides a conceptual representation of sequential consistency with several processors sharing a common logical memory. Each processor is required to issue its memory operations in program order. Operations are serviced by memory one-at-a-time; thus, they appear to execute atomically with respect to other memory operations. The memory services operations from different processors based on an arbitrary but fair global schedule. This leads to an arbitrary interleaving of operations from different processors into a single sequential order. The fairness criteria guarantees eventual completion of all processor requests. The above requirements lead to a total order on all memory operations that is consistent with the program order dictated by each processor's program.

An execution (or the result of an execution) of a program is sequentially consistent if there exists at least one execution on a conceptual sequentially consistent system that provides the same result (given the same input and same initial state in memory). Otherwise, the execution violates sequential consistency.

2.1.4 Examples of Sequentially Consistent Executions

Figure 2.3 provides some canonical sample programs to illustrate the semantics of sequential consistency. We use the following convention for specifying programs. We use uppercase letters (e.g., A, B, C) to denote shared memory locations and lowercase letters (e.g., u, v, w) to denote private memory locations. Register locations are denoted as r1, r2 . . . rn. Each instruction is given a unique label. Instructions from the same processor are sequenced by letters in alphabetical order with the processor number following the letter (e.g., *(a1)* is the first instruction on processor 1). We will often use the instruction labels to also identify the memory operation that results from executing the instruction. Unless specified otherwise, the initial values for all memory locations are *implicitly assumed* to be zero.

	P1		P2
a1:	A = 1;	*a2:*	u = B;
b1:	B = 1;	*b2:*	v = A;

(a)

	P1		P2
a1:	A = 1;	*a2:*	B = 1;
b1:	u = B;	*b2:*	v = A;

(b)

	P1		P2		P3
a1:	A = 1;	*a2:*	u = A;	*a3:*	v = B;
		b2:	B = 1;	*b3:*	w = A;

(c)

	P1		P2		P3		P4
a1:	A = 1;	*a2:*	u = A;	*a3:*	B = 1;	*a4:*	w = B;
		b2:	v = B;			*b4:*	x = A;

(d)

Figure 2.3: Sample programs to illustrate sequential consistency.

Consider the program segment in Figure 2.3(a). The result of executions of this program segment can be depicted by enumerating the values for variables (u,v). Consider an execution with the result (u,v)=(1,1). This execution is sequentially consistent because there is a total order on the operations, (a1,b1,a2,b2), that is consistent with program order and produces the same result. Now consider another execution with the result (u,v)=(1,0). This result violates the simple notion of causality since P2 observes the new value for B, but does not observe the new value for A even though the write to A is before the write to B on P1. One total order on operations that produces the same result is (b1,a2,b2,a1), but this total order violates the program order imposed by P1. Furthermore, there are no other total orders that are consistent with program order and yet produce the same result. Therefore, an execution that results in (u,v)=(1,0) violates sequential consistency. Figure 2.3(b) shows another two processor example, where an execution with the result (u,v)=(0,0) violates sequential consistency. Again, all total orders that produce the same result are inconsistent with the program order on at least one of the processors.

Figure 2.3(c) shows a three processor example. Consider the result (u,v,w)=(1,1,0), which is not sequentially consistent. This result violates causality across the three processors since P2 writes the new value of B after observing the new value of A, after which P3 observes the new value of B written by P2, but fails to observe the new value of A. And finally, in the four processor example of Figure 2.3(d), the result (u,v,w,x)=(1,0,1,0) violates sequential consistency. The result suggests that P2 observes the write of A happened before the write of B, while P4 observes the opposite order for the two writes. However, all processors should observe the same order on all writes in sequentially consistent executions.

An important attribute of the definition for sequential consistency is that it depends on the order in which operations *appear* to execute, not necessarily the order in which they *actually* execute in a system. For example, refer back to the program segment in Figure 2.3(a). Consider an execution on a system where the actual execution order on operations is (b1,a1,b2,a2) with the result (u,v)=(1,1). Even though the actual order above violates the program order on both processors, the execution is still considered sequentially consistent. This is because the result of the execution is the same as if the operations were done in the order (a1,b1,a2,b2) which is consistent with program order.

Figure 2.4 provides a couple of examples with multiple memory operations per instruction. The example in Figure 2.4(a) illustrates accessing data at multiple granularities. Assume A and B are consecutive words aligned at a double word boundary. The instruction on P1 represents an atomic double word write. We show the operations specified by the same instruction in a dotted box, and use an extended version of our labeling scheme to identify these operations (e.g., (a1.1) and (a1.2) are the operations specified by instruction (a1)). Since the two operations on P1 must appear atomic, the result (u,v)=(1,0) is disallowed under sequential consistency. Figure 2.4(b) illustrates the same example except the operations on P1 are now separated into two instructions. The result (u,v)=(1,0) is allowed under sequential consistency in the latter example since the total order (a1,a2,b2,b1) explains the result and is consistent with program order. Note that the analogous total order (a1.1,a2,b2,a1.2) for the first example violates the atomicity requirement for the multiple operations from instruction (a1).

Figure 2.4(c) illustrates the use of multiple operations per instruction to achieve an atomic read-modify-write. The example shows a fetch-and-increment of the same location on both processors. Again, atomicity requires all operations specified by one instructions to appear to execute before any operations specified by the other instruction. Therefore, (u,v)=(0,1) and (u,v)=(1,0) are possible outcomes, while (0,0) and (1,1) are

P1		P2	
a1.1:	A = 1;	*a2:*	u = A;
a1.2:	B = 1;	*b2:*	v = B;

(a)

P1		P2	
a1:	A = 1;	*a2:*	u = A;
b1:	B = 1;	*b2:*	v = B;

(b)

P1		P2	
a1.1:	u = A;	*a2.1:*	v = A;
a1.2:	A = u + 1;	*a2.2:*	A = v + 1;

(c)

P1		P2	
a1:	u = A;	*a2:*	v = A;
b1:	A = u + 1;	*b2:*	A = v + 1;

(d)

Figure 2.4: Examples illustrating instructions with multiple memory operations.

not possible. Figure 2.4(d) illustrates the same example except the operations on each processor are separated into two instructions. In contrast to the previous case, the outcome (u,v)=(0,0) is now possible. The total order on operations that explains this result is (a1,a2,b1,b2) and is consistent with program order.

Finally, our last set of examples illustrate the need for synchronization under sequential consistency. Even though sequential consistency maintains the sequential order among operations on each processor, it allows operations from different processors to be arbitrarily interleaved at the instruction granularity. Therefore, extra synchronization is required to provide atomicity across operations belonging to multiple instructions (e.g., by using locks). Similarly, synchronization is required to impose a deterministic order among operations on different processors (e.g., by waiting for a flag to be set). A number of mutual exclusion algorithms (e.g., Dekker's or Peterson's algorithm [And91]) have been developed for the sequential consistency model using single read and write operations. However, most current systems support atomic read-modify-write instructions, providing a simpler and more efficient mechanism for achieving mutual exclusion.

Consider the program segment in Figure 2.5(a). Assume A and B are two fields in a record and assume the desired semantics for the program is that reads and writes to the record should be done in a mutually exclusive fashion. Therefore, if P1 writes the two fields, and P2 reads the two fields, the only two acceptable results are (u,v)=(0,0) or (u,v)=(1,1) corresponding to either P1 or P2 accessing the record first. The program in Figure 2.5(a) fails to satisfy this requirement because the results (u,v)=(1,0) or (u,v)=(0,1) are possible under sequential consistency (possible total orders for these two results are (a1,a2,b2,b1) and (a2,a1,b1,b2), respectively). Figure 2.5(b) shows one way of guaranteeing atomic behavior by using a *test&set* instruction to achieve mutual exclusion; the two outcomes of (u,v)=(1,0) and (u,v)=(0,1) are no longer allowed.

Now consider the case where the programmer wants to communicate the new values of A and B to P2 in every execution as in a producer-consumer interaction. Therefore, we would like to guarantee the result (u,v)=(1,1) in every execution. Obviously mutual exclusion is not sufficient for guaranteeing this since if P2 gains access to the critical section before P1, then we can get the result (u,v)=(0,0). Figure 2.5(c) shows one possible way of supporting the desired producer-consumer semantics whereby P1 sets a flag after writing to the two locations and P2 waits for the flag to be set before reading the two locations. Other forms of

	P1		P2
a1:	A = 1;	*a2:*	u = A;
b1:	B = 1;	*b2:*	v = B;

(a)

	P1		P2
a1:	**while (test&set(L)==0);**	*a2:*	**while (test&set(L)==0);**
b1:	A = 1;	*b2:*	u = A;
c1:	B = 1;	*c2:*	v = B;
d1:	**L = 0;**	*d2:*	**L = 0;**

(b)

	P1		P2
a1:	A = 1;	*a2:*	**while (Flag==0);**
b1:	B = 1;	*b2:*	u = A;
c1:	**Flag = 1;**	*c2:*	v = B;

(c)

Figure 2.5: Examples illustrating the need for synchronization under SC.

synchronization (e.g., barriers) can also be easily implemented using read, write, and atomic read-modify-write operations to shared memory.

2.1.5 Relating Memory Behavior Based on Possible Outcomes

We can compare the memory behaviors of two different models, two different implementations, or a model and an implementation, by considering the set of possible results or outcomes (for executing various programs) that is allowed by each model or implementation.

Let A and B refer to different models or implementations. A and B are *equivalent* iff (abbreviation for if and only if) for any run of any program, the outcome in A is possible in B and vice versa. A is *stricter* than B iff for any run of any program, the outcome in A is possible in B but not vice versa. The above two relations are transitive by definition. A and B are *distinguishable* if they are not equivalent. A and B are *incomparable* iff they are distinguishable and neither is stricter than the other. Finally, a program that runs "correctly" on A is guaranteed to run correctly on B if B is equivalent to or stricter than A. This is because there will be no "unexpected outcomes"; all outcomes allowed by B are also possible on A. It is sometimes useful to define the equivalent and stricter relations over a subset of all programs. For example, two implementations may behave the same across a large subset of programs even though they are distinguishable if we consider all programs.

An implementation satisfies a given memory model if it is equivalent to *or* is stricter than the model. In practice, most implementations are actually stricter than the model they satisfy, i.e., they allow a subset of the outcomes allowed by the model. For example, a realistic design issues instructions at a set rate and does not arbitrarily vary the rate at which operations are issued, thus eliminating the possibility for some of the interleavings that are allowed by a conceptual model.

2.2 Impact of Architecture and Compiler Optimizations

This section considers the impact of common architecture and compiler optimizations on the behavior of shared memory operations. Many of the optimizations we discuss are critical to achieving high performance. However, most of them end up violating the semantics of a strict model such as sequential consistency. The issues presented here will be described in greater detail in Chapter 5.

2.2.1 Architecture Optimizations

Figure 2.6 shows the typical architecture for a scalable shared-memory multiprocessor. The processor environment (i.e., processor, write buffer, cache) within each node is quite similar to that of a uniprocessor. The nodes are connected using a general network (the figure shows a two-dimensional mesh network), and the shared memory is distributed among the nodes. The latency of memory operations is one of the key factors that determines the performance of a multiprocessor system such as the one shown above. Optimizations that reduce or hide the latency of memory operations are especially important in multiprocessors because of the following two reasons. First, memory latencies are typically larger than in a uniprocessor especially if the operation involves a remote node. Second, multiprocessors typically exhibit a larger cache miss rate relative to uniprocessors due to the inherent communication that takes place among the processors.

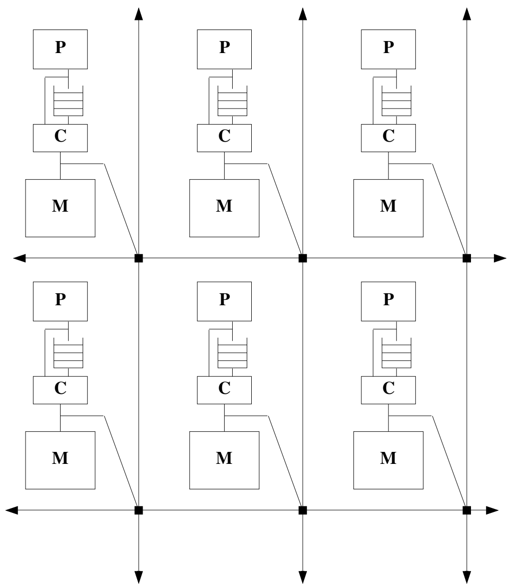


Figure 2.6: A typical scalable shared-memory architecture.

Aside from using caches, the typical way in which uniprocessors deal with large memory latencies is to overlap the service of multiple memory operations. For example, instead of stalling after a write operation, the processor simply places the write into a write buffer and continues to execute other instructions and memory operations. Even though such optimizations can potentially lead to a reordering of memory operations to different locations, the illusion of a sequential execution is still maintained. Maintaining this illusion is more difficult in multiprocessors because there is more than one processor that can observe the memory state. For example, the out-of-order execution of shared memory operations by one processor can potentially be detected by other processors.

To maintain sequential consistency, memory operations from individual processors must appear to execute in program order and all memory operations must appear atomic with respect to one another. We refer to these as the *program order* and *atomicity* requirements, respectively. Below, we describe how various architectural optimizations can violate the above two requirements. We will use the following notion of completion for memory operations. A read *completes* when its return value is bound (i.e., cannot be changed). A write *completes* when the corresponding location is updated with the new value. When there is more than a single copy of the same location (e.g., due to caching), the write is considered complete after all the copies have been affected. Explicit acknowledgement messages may be required for detecting the completion of write operations in a system such as the one shown in Figure 2.6.

Program Order Requirement

Memory operations from individual processors may complete out of program order due to a number of optimizations that attempt to overlap the servicing of such operations. We begin with optimizations at the processor level and move down in the memory hierarchy.

Operations may execute out of program order if the processor issues them out-of-order. This can easily occur in a dynamically scheduled (or out-of-order issue) processor, for example. Consider the program

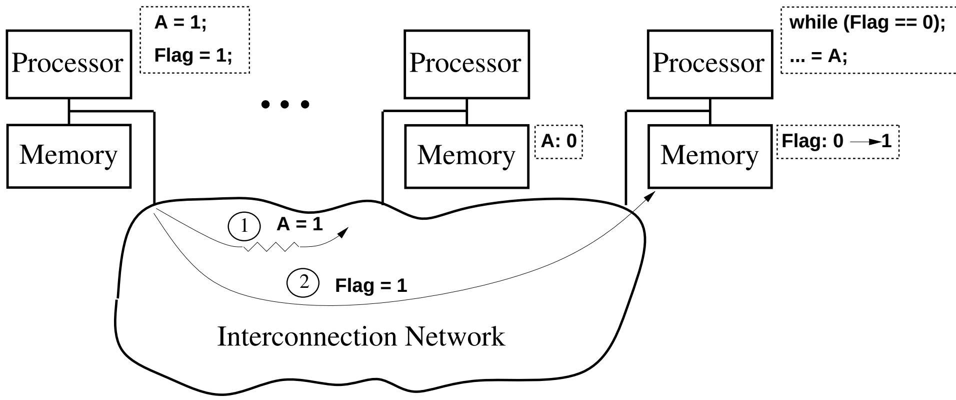


Figure 2.7: Reordering of operations arising from distribution of memory and network resources.

segment from Figure 2.3(a). Assume the processor issues the read of A before the read of B on P2, for example, because the effective address for A is available earlier. It is now possible for the execution to result in (u,v)=(1,0), which violates SC. An analogous situation can arise if the processor issues the writes on P1 out-of-order.

Even if the processor issues memory operations in program order, there is ample opportunity for them to get reordered due to optimizations within the memory hierarchy. Consider the write buffer. Optimizations such as read bypassing and forwarding have the effect of reordering a read with respect to previous write operations. Similarly, optimizations such as write merging (or coalescing) can lead to the reordering of writes. At the next level, a non-blocking or lockup-free cache [Kro81] can cause reordering. For example, consider two operations *o1* and *o2* in program order where *o1* misses in the cache and *o2* is a cache hit. Even if the two operations are issued to the cache in program order, operation *o2* completes earlier than operation *o1*.

Past the processor environment, reordering of memory operations can occur due to the distribution of memory system and network resources. Figure 2.7 shows an example of such reordering. The program segment we consider is the same as in Figure 1.1. As shown, the physical memory corresponding to locations A and Flag reside on two separate nodes. Assume the processor on the left issues the writes to these locations in program order. Since the write operations travel on separate paths to two different nodes, it is possible for the write of Flag to reach its memory module first. It is then possible for the processor on the right to observe the set flag and to read the old value of A from memory, all before the write to A reaches its memory module. Therefore, the presence of multiple paths in the network causes a reordering of the writes which leads to a violation of SC.

The reordering of memory operations from the same processor can be eliminated if a processor issues its operations in program order and waits for one to complete before issuing the next operation. While the above solution will satisfy the program order requirement, it fails to exploit many of the optimizations discussed above that are critical for dealing with large memory latencies.

Atomicity Requirement

Caching of shared data is an important optimization for reducing memory latency in shared memory systems. Of course, caching leads to multiple copies of data on different nodes which must be kept up-to-date on every

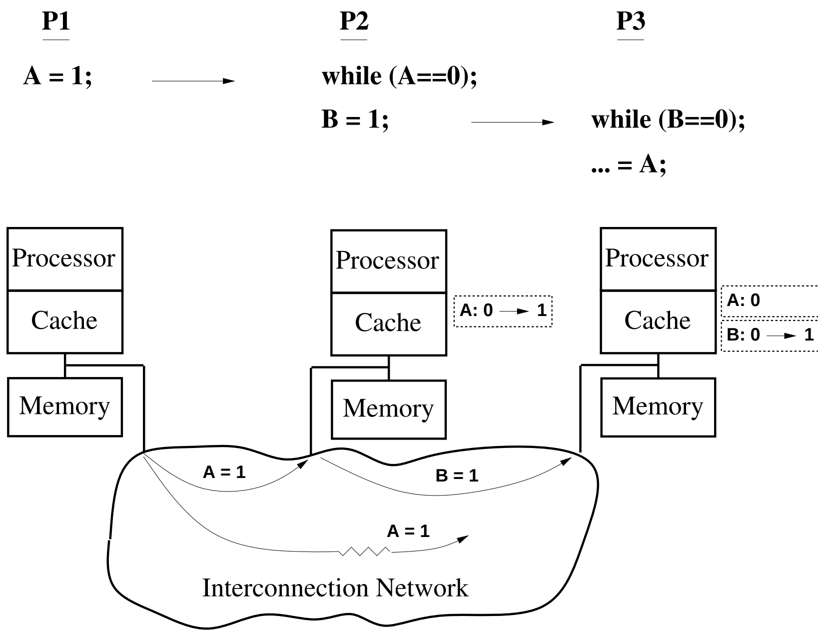


Figure 2.8: Non-atomic behavior of writes due to caching.

write. This can be achieved by either invalidating (i.e., eliminating) the copies or updating them to the new value. However, the distribution of the copies, along with the presence of multiple paths in the network, make it difficult to invalidate or update these copies in an atomic fashion.

Figure 2.8 shows an example of how caching can lead to a violation of the atomicity requirement for write operations. The program segment shown is similar to the one in Figure 2.3(c). Assume P2 initially has a copy of A and P3 initially has a copy of A and B in its cache. Furthermore, assume these copies are updated on every write to the corresponding location. Therefore, the write to A on P1 generates two update message, one for each copy of A. Because these update messages traverse different network paths, it is possible for the update to P2's cache to reach its target earlier. Therefore, P2 can observe the new value of A and can proceed with its write to B, which causes an update message to be sent to P3's copy of B. This latter update message can also reach P3 before the earlier update for A arrives there. It is now possible for P3 to proceed with its read of A which can return the old value of 0 that still resides in its cache. The above outcome violates sequential consistency and arises because the updates to the cached copies of A do not preserve the illusion of atomicity with respect to other operations. A similar situation can arise for the four processor example in Figure 2.3(d). As we will see later, there are efficient solutions to the above problem for systems that use invalidations to eliminate stale copies. However, solutions for update-based systems are inherently more difficult and inefficient.

Figure 2.9 shows another example of how caching can violate the atomicity requirement for writes to the same location. The program segment shown is similar to the one in Figure 2.3(d) except all operations are to the same location. Assume both P2 and P3 initially have a copy of A in their caches. Consider P1 and P4 attempting to write to A at the same time. Therefore, P1 will send an update message to P2 and P3 with the value of 1. Similarly, P4 will send an update message to P2 and P3 with the value 2. Due to the multiple paths in the network, the update messages may reach P2 and P3 in a different order. For example,

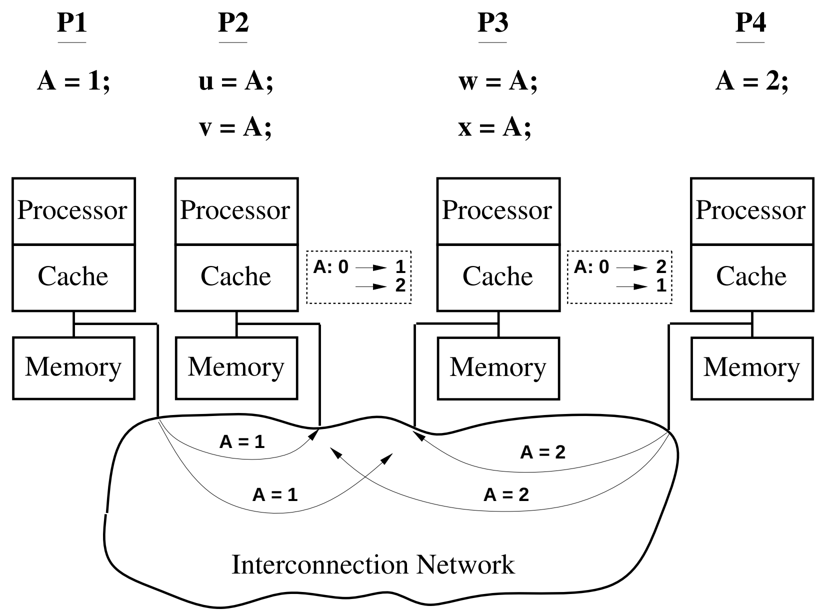


Figure 2.9: Non-atomic behavior of writes to the same location.

P2 may observe P1's write first followed by P4's write, while P2 may observe the writes in the opposite order. This can lead to the outcome (u,v,w,x)=(1,2,2,1) which violates sequential consistency. Furthermore, the two cached copies of A may remain permanently incoherent, with P2 caching the value of 2 and P3 caching the value of 1. Unlike the problem shown in Figure 2.8, this problem can be solved relatively efficiently for both invalidation and update-based protocols by serializing write operations on a per location basis.

In summary, many architectural enhancements that are commonly used to deal with large memory latencies affect memory behavior by either allowing operations to complete out of program order or making them appear non-atomic.

2.2.2 Compiler Optimizations

Analogous to architecture optimizations, many common compiler optimizations can also affect the behavior of shared memory operations. For example, optimizations such as register allocation, code motion, common sub-expression elimination, or loop interchange and blocking transformations, all have the effect of either reordering or eliminating memory operations. We briefly consider the effect of such optimizations in the context of compiling explicitly parallel programs.

In most cases, compiler optimizations lead to a violation of the program order requirement by reordering operations from the same processor. For example, consider applying code motion to the example program segment in Figure 2.3(a). Reordering the two writes on P1 or the two reads on P2 through code motion can change the behavior of this program by allowing the non-SC result of (u,v)=(1,0). Therefore, the overall effect is similar to hardware optimizations that reorder memory operations. Figures 2.10(a) and (b) show an example of how register allocation can lead to a similar reordering. The two program segments correspond to before and after register allocation. The result (u,v)=(0,0) is disallowed by the original program under SC. However, all executions of the program after register allocation result in (u,v)=(0,0). In effect, register

```
      P1                P2
a1:  B = 0;       a2:  A = 0;
b1:  A = 1;       b2:  B = 1;
c1:  u = B;       c2:  v = A;
```

(a) before

```
      P1                P2
a1:  r1 = 0;      a2:  r2 = 0;
b1:  A = 1;       b2:  B = 1;
c1:  u = r1;      c2:  v = r2;
d1:  B = r1;      d2:  A = r2;
```

(b) after

```
      P1                P2
a1:  A = 1;       a2:  while (Flag == 0);
b1:  Flag = 1;    b2:  u = A;
```

(c) before

```
      P1                P2
a1:  A = 1;       a2:  r1 = Flag;
b1:  Flag = 1;    b2:  while (r1 == 0);
                  c2:  u = A;
```

(d) after

Figure 2.10: Program segments before and after register allocation.

allocation reorders the read of B with the write of A on P1 and the read of A with the write of B on P2. Note that the above transformation would be considered safe if we treat each processor's code as a uniprocessor program.

Figures 2.10(c) and (d) illustrate yet another effect of optimizations such as register allocation that arise due to the elimination of memory operations. The while loop in the program segment on the left terminates in every SC execution. The program segment on the right shows the Flag location register allocated on P2, thus eliminating the multiple reads of Flag within the while loop. Consider an execution of the transformed program where the read of Flag on P2 returns the value 0. This execution violates SC because the while loop on P2 will not terminate. In effect, register allocating Flag disallows P2 from observing any changes to the location caused by the write on P1, making it appear as if the write of Flag on P1 is never observed by P2. Again, this optimization would be considered safe in a uniprocessor program.

In summary, the above examples show that common uniprocessor compiler optimizations can lead to "erroneous" program behavior if they are applied to explicitly parallel programs.

2.3 Implications of Sequential Consistency

This section provides a brief overview of the requirements that a system must satisfy to guarantee sequential consistency. We begin by considering the set of sufficient conditions proposed by Lamport [Lam79] and by Scheurich and Dubois [SD87]. We then briefly mention some of the more aggressive approaches that have been proposed for implementing sequential consistency more efficiently. The goal of this section is to provide a basic intuition for the relevant implementation issues. Chapter 5 provides a more thorough discussion of these issues.

2.3.1 Sufficient Conditions for Maintaining Sequential Consistency

In the same paper that introduces the notion of sequential consistency, Lamport proposes a set of system requirements that are sufficient for satisfying the semantics of this model [Lam79]. The conceptual system that is assumed consists of a set of processors having access to a set of memory modules; theoretically, there can be as many memory modules as there are unique memory locations. The two requirements are as follows.

Condition 2.1: Lamport's Requirements for Sequential Consistency
(a) Each processor issues memory requests in the order specified by its program.
(b) Memory requests from all processors issued to an individual memory module are serviced from a single FIFO queue. Issuing a memory request consists of placing the request in this queue.

Based on the above requirements, each processor needs to issue its memory operations in program order and needs to wait for each operation to get to the memory module, or its queue, before issuing the next memory operation. In architectures with general networks such as the one shown in Figure 2.7, detecting that a memory operation has reached its memory module requires an acknowledgement reply to be sent back from the memory module to the processor. Therefore, the processor is effectively delayed for the full latency of an operation before issuing its next operation. More efficient implementations are conceptually possible, especially with a more restrictive interconnection network such as a bus. The key observation made by Lamport's conditions is that, from an ordering point of view, a memory operation is effectively complete as soon as it is queued at its memory module (as long as the queue is FIFO). Therefore, a processor can conceptually overlap the period of time from when an operation reaches the queue to when the memory actually responds with the time to issue and service a later operation. The above observation does not hold for designs with caches, however. Furthermore, Condition 2.1 itself is not sufficient for correctness in the presence of data replication.

Scheurich and Dubois [SD87] provide a more general set of requirements that explicitly deals with optimizations such as caching. One of the key issues that arises in the presence of multiple copies is that writes may no longer behave atomically with respect to other memory operations. Scheurich and Dubois' sufficient conditions are summarized below. A read is considered complete when its return value is bound and a write is considered complete when all copies of the location are brought up-to-date (i.e., either through an invalidation or update mechanism).

Condition 2.2: Scheurich and Dubois' Requirements for Sequential Consistency
(a) Each processor issues memory requests in the order specified by its program.
(b) After a write operation is issued, the issuing processor should wait for the write to complete before issuing its next operation.
(c) After a read operation is issued, the issuing processor should wait for the read to complete, *and* for the write whose value is being returned by the read to complete, before issuing its next operation.
(d) Write operations to the same location are serialized in the same order with respect to all processors.

There are several important differences with Lamport's conditions that arise from dealing with multiple copies of the same location. For example, a write is considered complete only after all copies of the location are brought up-to-date, and a processor cannot issue operations following a read until the read return value is bound *and* the write responsible for that value is complete. Condition 2.2(c) and (d) are important for dealing with the inherently non-atomic behavior of writes, and address the problems depicted in Figures 2.8 and 2.9, respectively. One way to satisfy Condition 2.2(c) in an invalidation-based caching scheme is to disallow a newly written value from being read by any processor until all invalidations corresponding to the write are complete. Maintaining this condition for update-based protocols is more cumbersome, however, and typically requires a two phase interaction where cached copies are updated during the first phase followed by a second message that is sent to all copies to signal the completion of the first phase. These and other techniques are covered in detail in Chapter 5.

The conditions for satisfying sequential consistency are often confused with the conditions for keeping caches coherent. A *cache coherence protocol* typically ensures that the effect of every write is eventually made visible to all processors (through invalidation or update messages) and that all writes to the *same location* are seen in the same order by all processors. The above conditions are clearly not sufficient for satisfying sequential consistency. For example, sequential consistency ultimately requires writes to *all* locations to appear to be seen in the same order by all processors. Furthermore, operations from the same processor must appear in program order.

From an architecture perspective, the above conditions effectively require each processor to wait for a memory operation to complete before issuing its next memory operation in program order. Therefore, optimizations that overlap and reorder memory operations from the same processor may not be exploited. In addition, designs with caches require extra mechanisms to preserve the illusion of atomicity for writes. Similarly, from a compiler perspective, program order must be maintained among memory operations to shared data.

We next discuss how program-specific information can be used to achieve more aggressive implementations of SC.

2.3.2 Using Program-Specific Information

The sufficient requirements presented in the previous section maintain program order among all shared memory operations. However, for many programs, not all operations need to be executed in program order for the execution to be sequentially consistent. Consider the program segment in Figure 2.5(c), for example. The only sequentially consistent outcome for this program is (u,v)=(1,1). However, maintaining the program order between the two writes to A and B on P1 or the two reads to A and B on P2 is not necessary for guaranteeing a sequentially consistent result for this program. For example, consider an actual execution where the two writes on P1 are executed out of program order, with the corresponding total order on operations of (b1,a1,c1,a2,b2,c2) which is not consistent with program order. Nevertheless, the result of the execution is (u,v)=(1,1) since the writes to A and B still occur before the reads to those locations. This result is *the same as if* the total order was (a1,b1,c1,a2,b2,c2), which does satisfy program order. An analogous observation holds for the program segment in Figure 2.5(b). Of course, some program orders may still need to be maintained. For example, referring back to Figure 2.5(c), executing the write to Flag before the writes to A and B are complete can easily lead to the non-SC result of (u,v)=(0,0). Therefore, program-specific information must be used about memory operations in order to decide whether a given program order can be safely violated.

Shasha and Snir [SS88] have actually proposed an automatic method for identifying the "critical" program orders that are sufficient for guaranteeing sequentially consistent executions. This method may be used, for example, to determine that no program orders must be maintained in the example program segment in Figure 2.5(a). The reason is the outcomes (u,v)=(0,0), (0,1), (1,0), and (1,1) are all sequentially consistent outcomes, and allowing the operations to execute out of program order does not introduce any new outcomes that are not possible under sequential consistency. The information that is generated by such an analysis can be used by both the architecture and the compiler to exploit the reordering of operations where it is deemed safe. Similar to Lamport's work, Shasha and Snir assumed writes are atomic; therefore, their framework does not deal with issues of non-atomicity arising from the presence of multiple copies.

Unfortunately, automatically figuring out the critical set of program orders is a difficult task for general programs. The primary source of difficulty is in detecting conflicting data operations (i.e., operations to the same location where at least one is a write) at compile time. This is virtually the same as solving the well-known aliasing problem, except it is further complicated by the fact that we are dealing with an explicitly parallel program. The above problem is undecidable in the context of general programs. Furthermore, inexact solutions are often too conservative especially if the program uses pointers and has a complex control flow. Therefore, while the above techniques may work reasonably well for limited programs and programming languages, it is not clear whether they will ever become practical in the context of general programs.

The next section describes alternative techniques for enhancing system performance without violating the semantics of SC.

2.3.3 Other Aggressive Implementations of Sequential Consistency

This section presents a brief overview of a few other techniques that have been proposed to enhance the performance of sequentially consistent implementations. These techniques are specifically targeted at hardware implementations and are not applicable to the compiler. More detailed descriptions are presented in Chapter 5.

The first technique we discuss reduces the latency of write operations when the write has to invalidate or update other cached copies. A naive implementation of sequential consistency would acknowledge the invalidation or update message from each cache after the relevant copy has actually been affected. A more efficient implementation can acknowledge the invalidation or update earlier, i.e., as soon as the message is buffered by the target processor node. Thus, the issuing processor will observe a lower latency for the write. Sequential consistency is still upheld as long as certain orders are maintained with respect to other incoming messages. Afek et al. [ABM89, ABM93] originally proposed this idea, referred to as lazy caching, in the context of a bus-based implementation using updates and write-through caches. We present several extensions and optimizations to this idea in Chapter 5, making this technique applicable to a much wider range of implementations.

The above technique can be extended to more dramatically reduce the write latency in systems that use restrictive network topologies. Examples of such networks include buses, rings, trees, and hierarchies of such topologies. Landin et al. [LHH91] propose an implementation of SC that exploits the ordering guarantees in such networks to provide an early acknowledgement for writes. The acknowledgement is generated when the write reaches the "root" node in the network, which occurs before the write actually reaches its target caches. Chapter 5 presents our generalized version of this optimization in the context of various restricted network topologies.

Adve and Hill [AH90a] have also proposed an implementation for sequential consistency that can alleviate some of the write latency by allowing certain operations that follow the write to be serviced as soon as the write is serialized (i.e., receives ownership for the line) as opposed to waiting for all invalidations to be acknowledged.

Finally, a more promising way to enhance the performance of SC is to use a combination of the speculative read and automatic write prefetching techniques that we proposed in an earlier paper [GGH91b]. The idea behind speculative reads is to actually issue reads in an overlapped manner with previous operations from the same processor, thus allowing the operations to potentially complete out of program order. A simple mechanism is used to detect whether the overlapped completion of the operations can possibly violate sequential consistency. In the unlikely case that such a violation is detected, the speculative read and any computation that is dependent on it are simply rolled back and reissued. This technique is especially suited for dynamically scheduled processors with branch prediction capability since the required roll-back mechanism is virtually identical to that used to recover for branch mispredictions. The idea behind the second technique is to automatically prefetch the cache lines for write operations that are delayed due to the program order requirements. Thus, the write is likely to hit in the cache when it is actually issued. This latter technique allows multiple writes to be (almost fully) serviced at the same time, thus reducing the write latency as seen by the processor. The above two techniques have been adopted by several next generation processors, including the MIPS R1000 and the Intel Pentium Pro (both are dynamically scheduled processors), to allow for more efficient SC implementations.

Overall, except for the speculative read and the write prefetching techniques which are primarily applicable to systems with dynamically scheduled processors, the rest of the techniques described above fail to exploit many of the hardware optimizations that are required for dealing with large memory latencies. Furthermore, we are not aware of any analogous techniques that allow the compiler to safely exploit common optimizations without violating SC. Therefore, preserving the semantics of sequential consistency can still severely limit the performance of a shared-memory system. The following quotation from the two-page note by Lamport that originally proposed sequential consistency [Lam79] echoes the same concern: "The requirements needed to guarantee sequential consistency rule out some techniques which can be used to speed up individual sequential processors. For some applications, achieving sequential consistency may not be worth the price of slowing down the processors."

To achieve higher performance, many shared-memory systems have opted to violate sequential consistency by exploiting the types of architecture and compiler optimizations described in Section 2.2. This has led to a need for alternative memory consistency models that capture the behavior of such systems.

2.4 Alternative Memory Consistency Models

This section describes the various relaxed memory consistency models that have been proposed to capture the behavior of memory operations in systems that do not strictly obey the constraints set forth by sequential consistency. We begin the section by presenting a progression of relaxed models that enable more aggressive architecture and compiler optimizations as compared to sequential consistency. We next consider the relationship among these models in terms of the possible outcomes they allow for individual programs. Finally, the latter part of the section describes some of the shortcomings of relaxed models relative to sequential consistency.

2.4.1 Overview of Relaxed Memory Consistency Models

The basic idea behind relaxed memory models is to enable the use of more optimizations by eliminating some of the constraints that sequential consistency places on the overlap and reordering of memory operations.

While sequential consistency requires the illusion of program order and atomicity to be maintained for all operations, relaxed models typically allow certain memory operations to execute out of program order or non-atomically. The degree to which the program order and atomicity constraints are relaxed varies among the different models.

The following sections provide an overview of several of the relaxed memory consistency models that have been proposed. We have broadly categorized the various models based on how they relax the program order constraint. The first category of models includes the IBM-370 [IBM83], Sun SPARC V8 total store ordering (TSO) [SFC91, SUN91], and processor consistency (PC) [GLL+90, GGH93b] models, all of which allow a write followed by a read to execute out of program order. The second category includes the Sun SPARC V8 partial store ordering (PSO) model [SFC91, SUN91], which also allows two writes to execute out of program order. Finally, the models in the third and last category extend this relaxation by allowing reads to execute out of program order with respect to their following reads and writes. These include the weak ordering (WO) [DSB86], release consistency (RC) [GLL+90, GGH93b], Digital Equipment Alpha (Alpha) [Sit92, SW95], Sun SPARC V9 relaxed memory order (RMO) [WG94], and IBM PowerPC (PowerPC) [MSSW94, CSB93] models.

In what follows, we present the basic representation and notation used for describing the models and proceed to describe each model using this representation. Our primary goal is to provide an intuitive notion about the behavior of memory operations under each model. A more formal and complete specification of the models is provided in Chapter 4. Since many of the above models were inspired by the desire to enable more optimizations in hardware, our preliminary discussion of the models focuses mainly on the architectural advantages. Discussion of compiler optimizations that are enabled by each model is deferred to Section 2.4.6.

2.4.2 Framework for Representing Different Models

This section describes the uniform framework we use to represent the various relaxed memory models. For each model, our representation associates the model with a conceptual system and a set of constraints that are obeyed by executions on that system. Below, we use sequential consistency as an example model to motivate the various aspects of our representation.

Figure 2.11 shows the basic representation for sequential consistency. The conceptual system for SC consists of *n* processors sharing a single logical memory. Note that the conceptual system is meant to capture the behavior of memory operations from a programmer's point of view and does not directly represent the implementation of a model. For example, even though we do not show caches in our conceptual system, an SC implementation may still cache data as long as the memory system *appears* as a single copy memory (e.g., writes should appear atomic).

We represent shared memory read and write operations as R and W, respectively. A read operation is assumed to complete when its return value is bound. A write operation is assumed to complete when the corresponding memory location is updated with the new value. We assume each processor issues its memory operations in program order. However, operations do not necessarily complete in this order. Furthermore, unless specified otherwise, we implicitly assume all memory operations eventually complete.

Atomic read-modify-write operations are treated as both a read and a write operation. Most models require that it appears as if no other writes to the *same* location (from a different processor) occur between the read and the write of the read-modify-write. The TSO, PSO, and RMO models, however, require that no

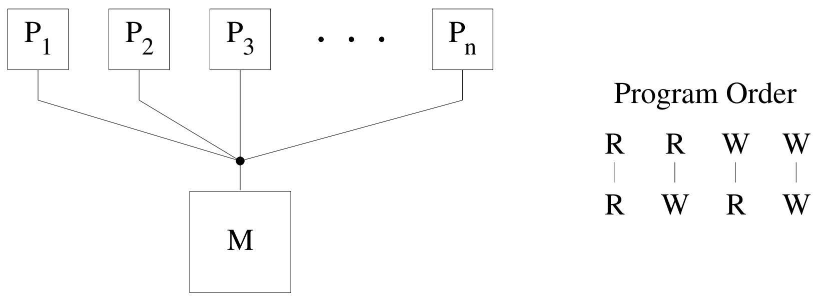


Figure 2.11: Representation for the sequential consistency (SC) model.

other writes to *any* location occur between the read and the write of the read-modify-write.

The first type of constraint on the execution of memory operations relates to program order and is referred to as the *program order requirement*. We use a pictorial representation to illustrate the program orders that are maintained by a model. As shown on the right side of Figure 2.11, we represent all possible pairs of read and write operations issued by the same processor that follow one another in program order: read followed by a read, read followed by a write, write followed by a read, and write followed by a write. The presence of a line between a given pair of operations signifies that program order should be maintained between such pairs, (i.e., the operations are required to complete in program order). As shown, SC requires program order to be maintained among all operation types.

The second type of constraint relates to the values returned by reads and is referred to as the *value requirement*. For sequential consistency, the value requirement is as follows: a read is required to return the value of the last write to the same location that completed in memory before the read completes. We will refer to this as the *memory value requirement*. Some of the models, such as TSO and PC, have a different value requirement that allows a processor to read the value of its own write before the write completes in memory. This latter semantics models the effect of optimizations such as *read forwarding* that allows a read to return the value of a write from a write buffer. We refer to this as the *buffer-and-memory value requirement*. Other models may impose additional types of constraints that will be discussed as we introduce each model.

The conceptual system shown in Figure 2.11 must obey the program order and memory value requirement described above to satisfy sequential consistency. An implementation obeys a given model if the result of any execution *is the same as if* the program was executed on the conceptual system. Therefore, a real system need not satisfy the constraints imposed on the conceptual system (e.g., program order) as long as the results of its executions appear as if these constraints are maintained. As a result, similar techniques to those described in Section 2.3 for implementing SC more aggressively are applicable to relaxed models as well. For simplicity, our discussion on the possible optimizations enabled by each model will concentrate on straightforward implementations. Chapter 5 provides more details about aggressive techniques that can be used to implement these models more efficiently.

2.4.3 Relaxing the Write to Read Program Order

The first category of relaxed models that we consider are those models that allow a write followed by a read to execute out of program order. The typical way hardware exploits this relaxation is through buffering writes and allowing a read to bypass the writes in the buffer. While maintaining sequential consistency typically

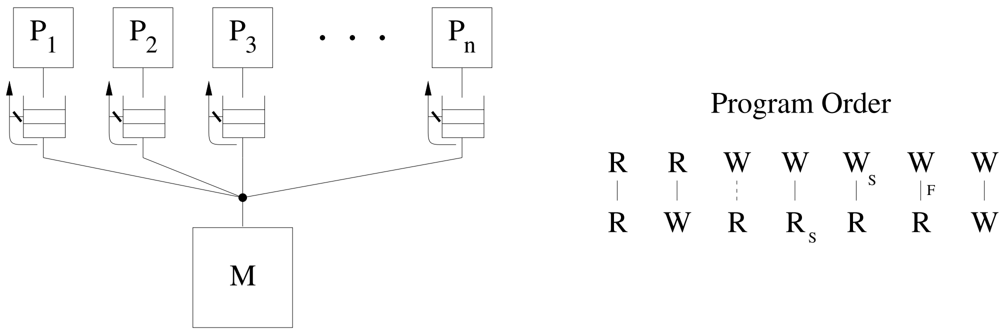


Figure 2.12: The IBM-370 memory model.

requires the processor to wait for a previous write to complete before completing the next read operation, this optimization allows the processor to continue with a read without waiting for write operations to complete. As a result, the write latency can be effectively hidden. Furthermore, many applications function correctly (i.e., provide sequentially consistent results) even if the program order from a write to a read is not maintained. Therefore, even though systems that exploit this optimization are not sequentially consistent, they appear sequentially consistent to a large class of programs.

The three models that we describe in this section are the IBM-370, TSO, and PC models. These models all provide the relaxation described above. The differences among them arise from the way they deal with the atomicity of memory operations and whether a processor is allowed to return the value of its own write before the write completes in memory.

IBM-370

Figure 2.12 provides the representation for the IBM-370 model [IBM83]. The IBM-370 model allows a write followed by a read to complete out of program order unless the two operations are to the same location, or if either operation is generated by a serialization instruction, or if there is a serialization instruction in program order between the two operations. The IBM-370 model has two types of serialization instructions: special instructions that generate memory operations (e.g., compare-and-swap) and special non-memory instructions (e.g., a special branch).

The conceptual system shown in Figure 2.12 is similar to that used for representing SC with a single logical memory shared by the processors. The main difference is the presence of a buffer between each processor and the memory. Since we assume that each processor issues its operations in program order, we use the buffer to model the fact that the operations are not necessarily issued in the same order to memory. Of course, the program order requirement places constraints on the reordering that can take place within the buffer. We also use the buffer to capture the behavior of models that allow a read to return the value of a conflicting write in the buffer before the write is actually issued to the memory.

The program order requirement for the IBM-370 model is shown on the right side of Figure 2.12. Similar to SC, the program orders between two reads, a read followed by a write, and two writes are maintained. The only difference is in the program order from a write to a read, which is only maintained in the three cases enumerated below. First, a write followed by a read to the same location must complete in the same order. We use a dashed line between the write-read pair to depict program order between operations to the same location.

Second, program order is maintained if either the write or the read are generated by a serialization instruction. We use R_S and W_S to depict read and write operations generated by serialization instructions; R and W still denote any read or write, including those generated by serialization instructions. Finally, program order is maintained if there is a non-memory serialization instruction between the write and the read. We generically refer to non-memory instructions that enforce program order as *fence* instructions [BMW85, GLL+90], and use a line with an "F" symbol (for fence) beside it to depict the presence of a program order enforced by a fence instruction. The program order constraints are transitive by definition. For example, if we have a sequence of (W,R_S,R) in program order, program order is automatically enforced between the first write and the last read.

The fact that the IBM-370 requires a write followed by a read to the same location to complete in program order implies that a read is not allowed to return the value of a write from the buffer. We depict this in Figure 2.12 by showing the reply path for a read from memory with the forwarding path from the buffer blocked. Therefore, the value requirement for IBM-370 is the same as for SC, i.e., a read returns the latest value in memory. Among the models we discuss in this section, IBM-370 is the only model that disallows optimizations such as read forwarding in an implementation.

To better understand the relationship between the SC and the IBM-370 models, we consider the possible outcomes under the two models for the program segments in Figure 2.3. Of the four programs, all except the example in Figure 2.3(b) provide the same set of outcomes on SC and the IBM-370 models. By allowing a read to be reordered with respect to a previous write, the IBM-370 model allows the non-SC outcome (u,v)=(0,0) for the example in Figure 2.3(b). Of course, the above difference with SC can be alleviated by using serialization or fence instructions to enforce the write-read program orders.

By requiring a write followed by a read to the same location to complete in program order, the IBM-370 model does not allow the latency of some writes to be hidden from the processor since the read must be delayed for the write to complete. The TSO model which is described next removes this restriction.

SPARC V8 Total Store Ordering (TSO)

The total store ordering (TSO) model is one of the models proposed for the SPARC V8 architecture [SFC91, SUN91]. Figure 2.13 shows the representation for this model. The main difference between the IBM-370 model and the TSO model is that TSO always allows a write followed by a read to complete out of program order. All other program orders are maintained. The conceptual system is almost identical to that of IBM-370 except the forwarding path from the buffer to a read is no longer blocked. Therefore, if a read matches (i.e., is to the same location as) a write in the write buffer, the value of the last such write in the buffer that is before it in program order is forwarded to the read. Otherwise, the read returns the value in memory, as in the SC and IBM-370 models.

Because the value of a write in the buffer is allowed to be forwarded to a read, the value requirement for TSO is different from the simple memory value requirement for SC and IBM-370. If we consider operations as executing in some sequential order, the *buffer-and-memory value requirement* requires the read to return the value of *either* the last write to the same location that appears before the read in this sequence *or* the last write to the same location that is before the read in program order, whichever occurs later in the sequence. This requirement captures the effect of forwarding the value of a write in the buffer in case of a match.

Figure 2.14 presents a couple of program segments that illustrate the differences between the TSO and

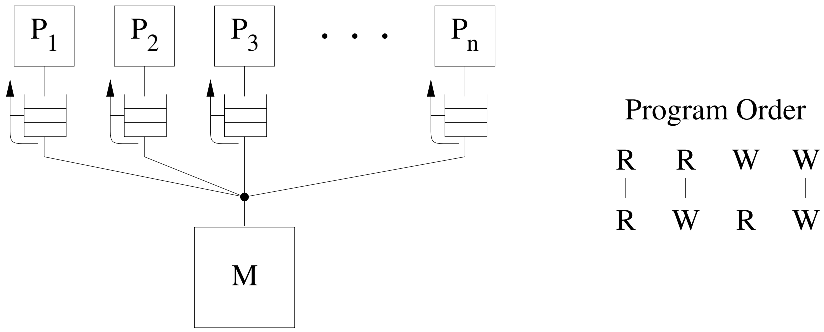


Figure 2.13: The total store ordering (TSO) memory model.

(a)

	P1		P2
a1:	A = 1;	*a2:*	B = 1;
b1:	u = A;	*b2:*	v = B;
c1:	w = B;	*c2:*	x = A;

(b)

	P1		P2
a1:	A = 1;	*a2:*	B = 1;
b1:	C = 1;	*b2:*	C = 2;
c1:	u = C;	*c2:*	v = C;
d1:	w = B;	*d2:*	x = A;

Figure 2.14: Example program segments for the TSO model.

IBM-370 or SC models. First consider the program segment in Figure 2.14(a). Under the SC or IBM-370 model, the outcome (u,v,w,x)=(1,1,0,0) is disallowed. However, this outcome is possible under TSO because reads are allowed to bypass all previous writes, even if they are to the same location; therefore the sequence (b1,b2,c1,c2,a1,a2) is a valid total order for TSO. Of course, the value requirement still requires b1 and b2 to return the values of a1 and a2, respectively, even though the reads occur earlier in the sequence than the writes. This maintains the intuition that a read observes all the writes issued from the same processor as the read. Figure 2.14(b) shows a slightly different program segment. In this case, the outcome (u,v,w,x)=(1,2,0,0) is not allowed under SC or IBM-370, but is possible under TSO.

Processor Consistency (PC)

The models we have considered up to now all provide the appearance of a single copy of memory to the programmer. Processor consistency (PC) [GLL+90, GGH93b] is the first model that we consider where the multiple-copy aspects of the memory are exposed to the programmer.[1] Figure 2.15 shows the representation for this model. The conceptual system consists of several processors each with their own copy of the entire memory. By modeling memory as being replicated at every processing node, we can capture the non-atomic effects that arise due to presence of multiple copies of a single memory location.

Since the memory no longer behaves as a single logical copy, we need to extend the notion of read and write memory operations to deal with the presence of multiple copies. Read operations are quite similar to before and remain atomic. The only difference is that a read is satisfied by the memory copy at the issuing processor's node (i.e., read from P_i is serviced by M_i). Write operations no longer appear atomic, however.

[1] The processor consistency model described here is distinct from the (informal) model proposed by Goodman [Goo89, Goo91].

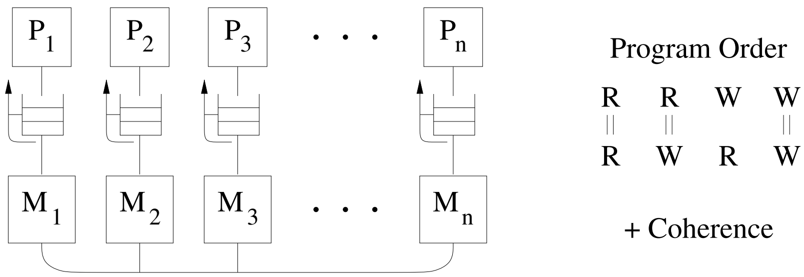


Figure 2.15: The processor consistency (PC) model.

Each write operation conceptually results in all memory copies corresponding to the location to be updated to the new value. Therefore, we model each write as a set of n sub-operations, W(1) ... W(n), where n is the number of processors, and each sub-operation represents the event of updating one of the memory copies (e.g., W(1) updates the location in M_1). Since we need to refer to the sub-operations of a write, we will also refer to a read as a single atomic sub-operation for uniformity (denoted as R(i) for a read from P_i).

Given that writes consist of multiple sub-operations now, our notion of maintaining program order changes slightly as well. Consider two writes in program order, for example. To maintain the program order among the two writes, we require all sub-operations of the first write to complete before any sub-operations of the second write complete. As before, a read sub-operation completes when its return value is bound. A write sub-operation W(i) completes when it updates its corresponding location in M_i. Figure 2.15 shows the program order requirement for PC. We use the *double lines* between a pair of operations to denote the fact that operations may no longer be atomic and that all sub-operations of the first operation must complete before any sub-operations of the second operation.

Since write operations are no longer atomic, processor consistency imposes an additional constraint on the order of write sub-operations to the same location. This requirement is called the *coherence requirement*. The coherence requirement constrains all write sub-operations to the same location to complete in the same order across all memory copies. In other words, given W1 and W2 are two writes to the same location, if W1(i) completes before W2(i), the coherence requirement requires W1(j) to also complete before W2(j) for all j. Therefore, there is conceptually a serialization point for all write operations to the same location. Finally, the value requirement for PC is an extension of the buffer-and-memory value requirement described for TSO. The following summarizes the conditions for PC.

Condition 2.3: Conditions for Processor Consistency (PC)
Memory sub-operations must execute in a sequential order that satisfies the following conditions:
(a) sub-operations appear in this sequence in the order specified by the program order requirement of Figure 2.15, and
(b) the order among sub-operations satisfies the coherence requirement, and
(c) a read sub-operation issued by R(i) returns the value of either the last write sub-operation W(i) to the same location that appears before the read in this sequence or the last write sub-operation to the location that is before the read in program order, whichever occurs later in the execution sequence.

To illustrate the effect of non-atomic writes, consider the program segments in Figures 2.3(c) and (d). Processor consistency allows the outcome (u,v,w)=(1,1,0) in the first example and the outcome (u,v,w,x)=(1,0,1,0) in the second example, while none of the previous models we have discussed allow these outcomes.

The fact that writes are not required to appear atomic makes PC suitable for update-based as well as invalidation-based designs (as discussed in Section 2.2.1, supporting atomic writes is difficult in update-based

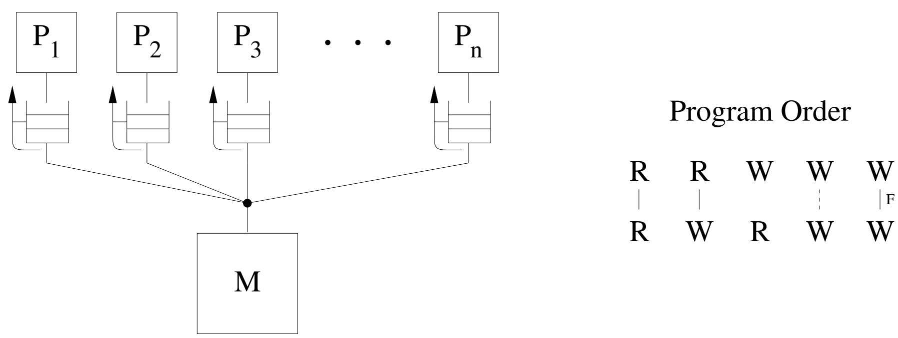


Figure 2.16: The partial store ordering (PSO) model.

designs). However, by far the major advantage of models such as PC, TSO, and IBM-370 over SC is the ability to hide the write latency by allowing reads to execute out of program order with respect to previous writes.

2.4.4 Relaxing the Write to Write Program Order

The second category of relaxed models that we consider allow two writes to execute out of program order in addition to allowing the reordering of a write followed by a read. This relaxation enables a number of hardware optimizations, including write merging in a write buffer and overlapping multiple write misses, all of which can lead to a reodering of write operations. Therefore, write operations can be serviced at a much faster rate. Below, we describe the characteristics of the partial store ordering (PSO) model which allows this optimization.

SPARC V8 Partial Store Ordering (PSO)

The partial store ordering model (PSO) [SFC91, SUN91] is an extension of the TSO model and is the second model provided by the SPARC V8 architecture. Figure 2.16 shows the representation for this model. The conceptual system is identical to that of TSO. The program order requirement is also similar except that writes to different locations are allowed to execute out of program order. We use the same notation of a dotted line, which we used for the IBM-370 model, to indicate program order is maintained between (write) operations to the same location. PSO also provides a fence instruction, called the store barrier or STBAR, that may be used to enforce the program order between writes.

Referring back to the program segment in Figure 2.5(c), the only outcome allowed by the previous models we have discussed is (u,v)=(1,1). However, since PSO allows writes to different locations to complete out of program order, it also allows the outcomes (u,v)=(0,0) or (0,1) or (1,0). For this specific example, it is sufficient to place a single store barrier immediately before the write to Flag on P1 in order to disallow all outcomes except (u,v)=(1,1).

2.4.5 Relaxing the Read to Read and Read to Write Program Order

The third category of relaxed models we consider provide a more general set of reorderings by also allowing two reads or a read followed by a write to complete out of program order. The hardware can exploit the overlap of a read operation with operations that follow it in program order to hide some of the read latency. We will consider the following five relaxed memory models in this section: weak ordering (WO), release consistency (RC), Alpha, Relaxed Memory Order (RMO), and PowerPC. Weak ordering [DSB86] is the seminal model among these. We extended the ideas in weak ordering to define the release consistency model; this model is supported by the Stanford DASH design [LLG+92]. Finally, the last three models have been proposed for use in commercial multiprocessors.

In what follows, we describe the characteristics of each of the above models. For simplicity, we ignore a few subtle ordering requirements, related to control dependences, that are imposed by some of the original definitions for these models.[2] We also have had to make some interpretations of our own to describe a few of the models whose original definitions are partially ambiguous. Finally, we do not attempt to model instruction fetches, I/O operations, or multiple granularity data operations, even though some of the commercial models describe the semantics for such operations. Formal specifications for these models are presented in Chapter 4 and Appendix I.

Weak Ordering (WO)

The weak ordering model (also known as weak consistency) was proposed by Dubois *et al.* [DSB86] and relies on maintaining program order only at the synchronization points in the program. The intuition behind weak ordering is that most programs are written using synchronization operations to coordinate memory operations on different processors and maintaining program order at such synchronization operations typically leads to correct outcomes (e.g., SC outcomes) for the program. For example, consider the program segment in Figure 2.5(b). In this example, locks are used to ensure that accesses to shared data occur within critical sections on each processor. If the test-and-set operation and the write to reset the lock variable on each processor are identified explicitly as synchronizations, weak ordering guarantees the executions of this program will be sequentially consistent.

Figure 2.17 shows the representation for weak ordering. The conceptual system is similar to that of PC since weak ordering also exposes the multiple-copy semantics of memory to the programmer.[3] The only difference is that, similar to the IBM-370 model, a read is not allowed to return the value of a previous write to the same location until the write completes (i.e., the forwarding path from the buffer is blocked).

Figure 2.17 also shows the program order requirements for WO. We use R_S and W_S to denote read and write operations that are identified as synchronization. As before, R and W denote any read or write operation, including synchronization operations. As with PC, the double lines between operations in program order denotes the fact that each operation may consist of multiple sub-operations and that all sub-operations of the first operation must complete before any sub-operations of the second operation. We also use a new notation consisting of *triple lines* between a read and a following operation in program order. This notation denotes that the read sub-operation and the sub-operations of the write (possibly from a different processor) whose value is returned by the read should complete before any of the sub-operations of the operation that follows the read in program order. Dubois *et al.* refer to this as the read being "globally performed" [DSB86]. As shown in the figure, weak ordering maintains the program order between a synchronization operation and

[2] See the description of the reach condition in Appendix F and Appendix I.

[3] The way multiple-copy semantics is exposed is subtle and is only apparent from the formal specification of WO presented in Appendix I.

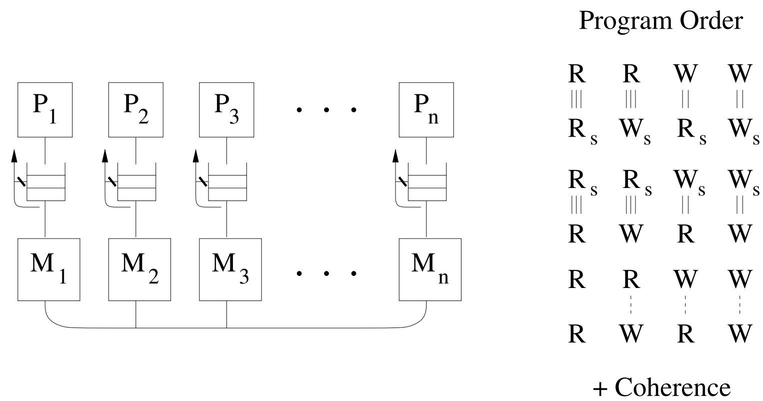


Figure 2.17: The weak ordering (WO) model.

operations that are before or after it in program order. Furthermore, program order is maintained among conflicting operations (i.e., operations to the same location with at least one being a write).

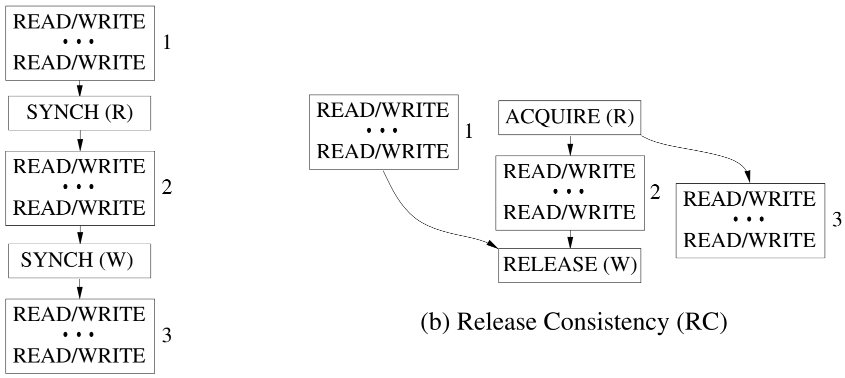
Similar to PC, the weak ordering model also constrains the order of writes to the same location by the coherence requirement. Therefore, writes to the same location must complete in the same order with respect to all memory copies.

Finally, the value requirement for WO is an extension of the memory value requirement: a read sub-operation R(i) returns the value of the last write sub-operation W(i) to the same location that appears before the read in the execution sequence. It turns out that the outcomes allowed by WO would be identical if we relax the program order requirement from a write to a read and use the extension of the buffer-and-memory value requirement defined for PC. Therefore, optimizations such as read forwarding can be safely exploited. Nevertheless, since the behavior of the model is the same without this relaxation, we do not need to expose the relaxation to the programmer. This same observation holds for the Alpha and PowerPC models.

Figure 2.18(a) illustrates the types of reordering allowed by WO by showing a sample set of operations from the same processor. Each block with a set of reads and writes represents a run of non-synchronization operations. Synchronization operation are identified separately. The figure shows that a synchronization operation (either a read or a write) must be delayed until all previous read and write operations complete, and read and write operations may not complete until the previous synchronization operation is complete. Weak ordering allows operations to different locations to be reordered in the region between two synchronization operations. Since synchronization operations are infrequent in many applications, this flexibility provides the opportunity to reorder and overlap operations across large regions of code. Note that by conservatively identifying all operations as synchronization, weak ordering trivially guarantees sequentially consistent executions for any program.

Release Consistency (RC)

Release consistency extends the ideas in weak ordering by further distinguishing among memory operations [GLL+90, GGH93b]. Specifically, release consistency further categorizes synchronization operations into *acquire* and *release* operations. An acquire is a read memory operation that is performed to gain access to a set of shared locations (e.g., a lock operation or spinning for a flag to be set). A release is a write operation



(a) Weak Ordering (WO)

(b) Release Consistency (RC)

Figure 2.18: Comparing the WO and RC models.

that is performed to grant the permission for accessing a set of shared locations (e.g., an unlock operation or the setting of a flag). This extra information is used to allow further reordering among memory operations. For simplicity, we describe only a subset of the operation categories used by release consistency. Chapter 4 presents the complete specification for this model.

Figure 2.18 shows the difference between WO and RC for the example we discussed in the previous section. The main idea behind release consistency is that read and write synchronization operations have different ordering requirements. The purpose of a write synchronization used as a release is to signal that previous accesses are complete and it does not have anything to say about ordering of accesses that follow it. Therefore, while the completion of the release is delayed until previous memory operations in program order complete, memory operations after a release are not delayed for the release to complete. Similarly, the completion of a read synchronization used as an acquire need not be delayed for previous memory operations to complete. This is because the acquire is not giving permission to any other process to read or write the previous pending locations. As shown in the figure, this allows for extra reordering and overlap of memory operations across acquires and releases.

Figure 2.19 provides the representation for release consistency. There are two flavors of release consistency that differ in the order that is maintained among synchronization operations. The first flavor maintains sequential consistency among synchronization operations and is referred to RCsc, while the second flavor maintains processor consistency among such operations and is called RCpc. Except for the program order requirements, these two models are identical. Considering the conceptual system, release consistency is similar to weak ordering except a read is allowed to return the value of a write (to the same location) that is in the buffer. The value requirement for both RCsc and RCpc is the same as that for PC. Finally, both models obey the coherence requirement.

Let us consider the program order requirements in more detail. As we mentioned above, release consistency provides a further categorization of operations. Both acquires and releases belong to a category called *competing* operations. Intuitively, competing operations are those operations that are involved in a race with other conflicting operations.[4] Competing operations also include a third category of operations that are not used to achieve synchronization. We use R_C and W_C to depict competing operations. R_{acq} and W_{rel} denote

[4]In release consistency, the requirement for a write operation to eventually complete applies to competing writes only.

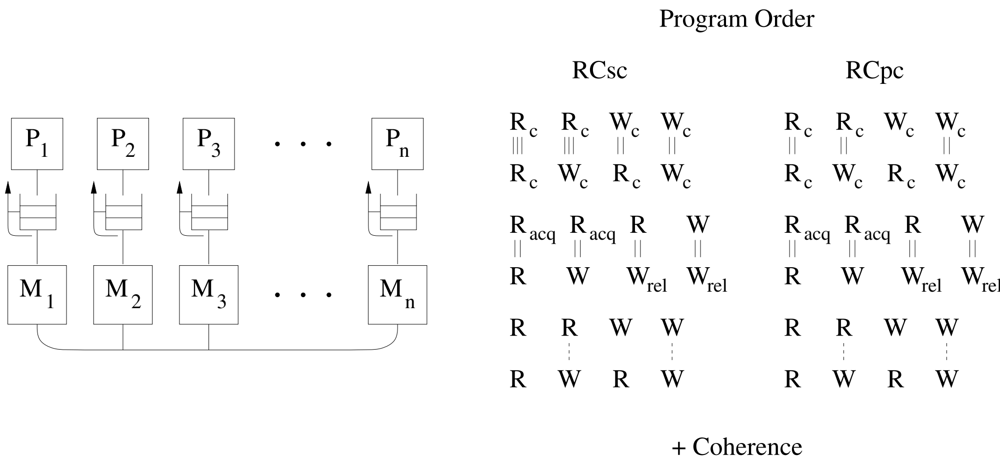


Figure 2.19: The release consistency (RC) models.

acquire and release operations. Finally, R and W still denote any read or write. For RCsc, all competing operations are required to complete in program order. In addition, the program order between an acquire and following operations, and between an operation and a following release, are maintained. Finally, for conflicting operations, program order is maintained from a read to a write and between two writes (shown with dotted lines); the order from a conflicting write to a read is not upheld, however. RCpc differs from RCsc only in the order maintained among pairs of competing operations.

Referring back to Figure 2.5(c), it is sufficient to identify the write to Flag as a release and the read of Flag as an acquire operation to ensure SC outcomes under both RCsc and RCpc models. The release consistency model was proposed in conjunction with the proper labeling (PL) framework [GLL+90], which is a programmer-centric specification that presents the programmer with a higher level abstraction of system behavior to simplify the task of programming with RC. This framework is described in detail in the next chapter.

DEC Alpha (Alpha)

In contrast to the WO and RC models which require memory operations to be identified or labeled according to a given categorization, the three commercial memory models we consider impose ordering solely through explicit fence instructions. We begin by considering the Alpha memory model [Sit92, SW95]. Figure 2.20 provides the representation for this model. The conceptual system is identical to that of the IBM-370 model. The program order constraints allow reads and writes to different locations to complete out of program order unless there is a fence instruction between them. The Alpha model supports two types of fences, the memory barrier (MB) and the write memory barrier (WMB); these are labeled as F1 and F2 in the figure for fence types 1 and 2, respectively. As shown, an MB imposes program order between all read and write operations, while the WMB only imposes program order between writes. Finally, memory operations to the same location, including reads to the same location, are required to complete in program order. Among the models discussed in this category (i.e., WO, RCsc, RCpc, Alpha, RMO, PowerPC), Alpha is the only model that enforces the program order between reads to the same location.

The value requirement for Alpha is the same as the memory value requirement used for SC or IBM-370,

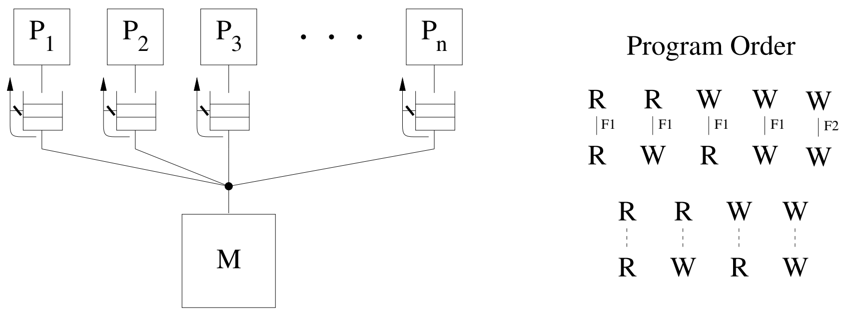


Figure 2.20: The Alpha memory model.

which requires a read to return the value of the last write operation to the same location. However, similar to WO, the semantics of the model is unaffected if we relax the program order requirement from a write to a read and use the buffer-and-memory value requirement of TSO.[5] Therefore, implementations can safely exploit optimizations such as read forwarding.

The Alpha memory model can be used to emulate several of the previous models we have discussed by appropriately using the explicit fence instructions. For example, we can guarantee SC by naively placing a fence instruction between any pair of memory operations. Similarly, we can preserve the requirements for WO by placing a fence instruction before and after every memory operation identified as a synchronization. Frequent use of fence instructions can incur a significant overhead, however, due to an increase in the number of instructions and the extra delay that may be associated with executing fence instructions. This observation holds for any model that requires the use of explicit fence instructions to impose orders (e.g., PSO, RMO, PowerPC).

SPARC V9 Relaxed Memory Order (RMO)

The relaxed memory order (RMO) model is an extension of the TSO and PSO models and was adopted for the SPARC V9 architecture [WG94]. RMO extends the PSO model to allow a read to complete out of program order with respect to following read and write operations. Our research on the release consistency model and quantifying its performance gains had a direct influence on the design of RMO, and we helped Sun with formalizing the semantics of this model. Figure 2.21 shows the representation for RMO. RMO provides four types of fences that allow program order to be selectively maintained between any two types of operations. A single fence instruction can specify a combination of the above fence types by setting the appropriate bits in a four-bit opcode. The program order constraints are shown in the figure, with the four fence types depicted as F1 through F4. RMO also maintains the program order between a read followed by a write or two writes to the same location. The value requirement for RMO is the same as for TSO and PSO.

Similar to the Alpha model, the RMO model can be used to provide sufficient implementations for several of the models we have discussed.

[5] We are not considering the effect of data accesses at multiple granularities (e.g., byte, word, long word, quad word) in this discussion. Capturing the behavior of the Alpha model with multiple granularity accesses would actually require the use of the buffer-and-memory value requirement, along with allowing a write followed by a read to the same location to complete out of program order [SW95].

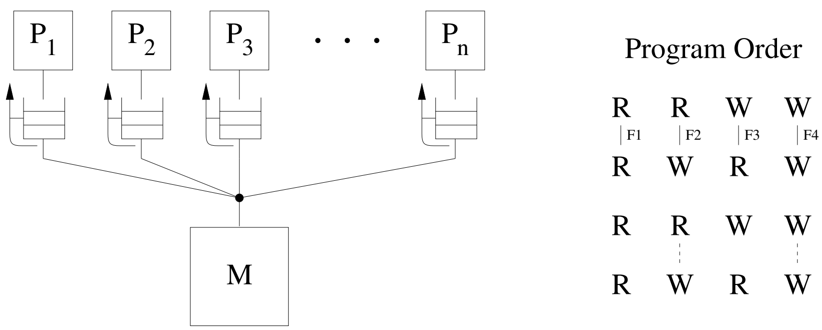


Figure 2.21: The Relaxed Memory Order (RMO) model.

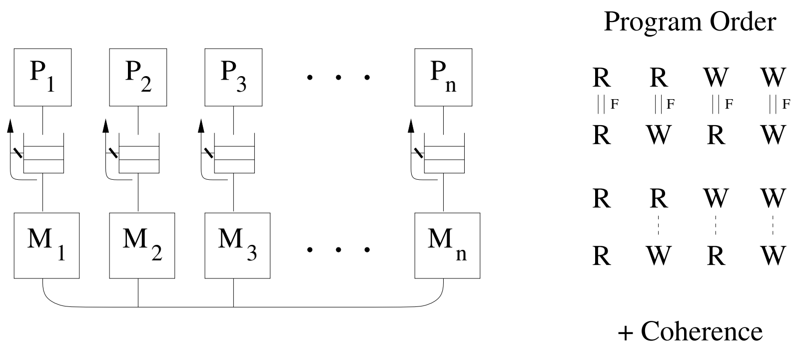


Figure 2.22: An approximate representation for the PowerPC model.

IBM PowerPC (PowerPC)

In contrast to the Alpha and RMO models, the IBM PowerPC model [CSB93, MSSW94] exposes the multiple-copy semantics of memory to the programmer. Figure 2.22 shows the representation for PowerPC. This representation is only approximate for reasons that will be described shortly. The conceptual system is identical to that of WO. PowerPC provides a single fence instruction, called a SYNC, that may be used to impose program orders. Remember that the double lines between operations in program order denotes the fact that each operation may consist of multiple sub-operations and that all sub-operations of the first operation must complete before any sub-operations of the second operation. Note that the program order from a read to its following operations does not delay for the read to be globally performed, which would have been depicted by a triple line. Program order is also maintained among conflicting operations. PowerPC also constrains the order of writes to the same location through the coherence requirement. The value requirement for PowerPC is the same as for WO: a read sub-operation R(i) returns the value of the last write sub-operation W(i) to the same location that appears before the read in the execution sequence. The semantics of the model is unaffected if we relax the program order requirement from a write to a read and use the extension of the buffer-and-memory value requirement used for PC. Therefore, optimizations such as read forwarding from the buffer are actually safe.

The representation shown in Figure 2.22 is stricter than the actual PowerPC model. Figure 2.23(a) shows an example program that illustrates this distinction. The representation in Figure 2.22 along with the constraints described above imply that the outcome (u,v)=(1,0) is disallowed under PowerPC. However, the formalism provided by Corella et al. [CSB93] implies that this outcome is actually allowed even though a

```
        P1                 P2                       P1                 P2

a1:  A = 1;        a2:  u = A;              a1:  Ptr = ...;    a2:  while (Ptr == 0);
                   b2:  sync;                                  b2:  sync;
                   c2:  v = A;                                 c2:  u = Ptr->...;

              (a)                                         (b)
```

Figure 2.23: Example program segments for the PowerPC model.

fence instruction is placed between the two reads on P2. Figure 2.23(b) shows a similar code segment to illustrate that this scenario can actually occur in practice. P2 waits for a null pointer to be set and then proceeds to dereference the pointer. Under PowerPC, it is possible for P2 to erroneously dereference a null pointer. Capturing this behavior in the simple framework we have been using is difficult. Similarly, designing a hardware implementation that satisfies PowerPC and yet allows the outcome (u,v)=(1,0) for the program in Figure 2.23(a) also seems quite difficult. Therefore, this behavior arises more as a side effect of the way PowerPC has been formalized, and actual implementations of the model will likely not behave in this way. Nevertheless, this subtle relaxation in PowerPC can make ports to this model more difficult and inefficient. Appendix I provides the precise specification for PowerPC using our more general framework.

2.4.6 Impact of Relaxed Models on Compiler Optimizations

In discussing the relaxed memory models, we have mainly concentrated on the performance differences among the models based on the types of architectural optimizations they enable. This section briefly discusses some of the implications for compiler optimizations.

The main flexibility required for doing compiler optimizations is to allow memory operations within each process or thread to be reordered with respect to their program order. As we discussed in Section 2.3, maintaining sequential consistency effectively disallows such reordering on shared memory operations. Compared to SC, the three categories of relaxed models that we discussed provide a higher degree of freedom for such reordering. However, not all the relaxed models provide sufficient flexibility to allow general compiler optimizations. For example, the first category of models that we discussed (e.g., TSO and PC) only allow reordering of reads with respect to previous writes in program order. While hardware can exploit this relaxation to hide write latency, exploiting this limited type of reordering is difficult for a compiler.

Enabling a reasonable set of compiler optimizations effectively requires full flexibility in reordering both reads and writes with respect to one another. Among the models we discussed, only the last category of models (i.e., WO, RC, Alpha, RMO, PowerPC) provide such flexibility. With WO or RC, the compiler is allowed to freely reorder operations in between consecutive synchronization points. Similarly, with Alpha, RMO, and PowerPC, memory operations between consecutive fence instructions can be arbitrarily reordered with respect to one another. Therefore, the compiler can perform optimizations such as register allocation and code motion on regions of code delimited by synchronization or fences. Chapter 5 will provide a more detailed discussion about compiler optimizations.

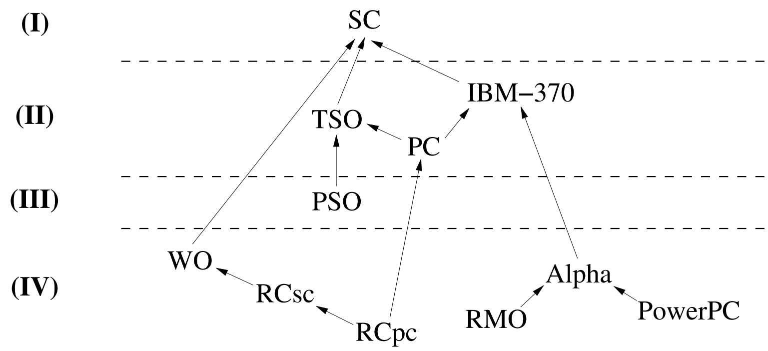


Figure 2.24: Relationship among models according to the "stricter" relation.

2.4.7 Relationship among the Models

Figure 2.24 provides a pictorial view of the "stricter" relation introduced in Section 2.1.5 among the models we have discussed so far. Given *A* and *B* refer to different models, *A* is *stricter* than *B* iff for any run of any program, the outcome in *A* is possible in *B* but not vice versa. The models are partitioned into four categories based on the level of flexibility they allow in reordering the program order among memory operations. In the figure, $A \longleftarrow B$ denotes that *A* is stricter than *B*. Since the stricter relation is transitive, the arrows can be followed transitively as well. For example, RCpc is stricter than SC. The lack of an arrow between two models means that the two models are incomparable. For example, even though a model such as WO allows a more general set of reorderings compared to TSO, neither is stricter than the other due to subtle differences among the two models. Similarly, IBM-370 is not stricter than TSO because TSO imposes stricter constraints on read-modify-write operations.

The relations in the figure are based on precise specifications of the models presented in Chapter 4 and Appendix I. We should note that we do not consider the semantics for operations such as instruction fetches and I/O operations or the effect of multiple granularity accesses. Some of the stricter relations shown in the figure depend on straightforward assumptions regarding the interpretation of fences or operation labels across different models. For example, we assume all SYNC instructions in PowerPC are treated as MB instructions in Alpha. Similarly, we assume competing operations in RCsc or RCpc are treated as synchronization operations in WO. We make no assumptions about transforming fences to labels or labels to fences, however.

A program written for model *A* is guaranteed to run "correctly" (i.e., all outcomes will be allowable by *A*) on another model *B* if the latter is stricter than the former. For example, any programs written assuming the PC model can be safely executed on the TSO, IBM-370, or SC models. However, the reverse is not necessarily true because PC violates some of the ordering requirements guaranteed by the other three models. Nevertheless, it is still possible to port programs between any two models by appropriately transforming the program. We will further discuss automatic techniques for porting program across different models in Chapter 4.

2.4.8 Some Shortcomings of Relaxed Models

Even though relaxed models enable desirable optimizations, their major drawback is increased programming complexity. Most programmers have implicit assumptions about the memory behavior of a shared-memory multiprocessor and use these assumptions when reasoning about the correctness of their programs. Correctness problems arise when certain orders that are implicitly assumed by the programmer are not maintained by the underlying memory model. The advantage of sequential consistency is that no matter what implicit assumptions a programmer makes regarding the program order or atomicity of memory operations, SC conservatively maintains all such orders. Therefore, the programmer's implicit assumptions are never violated.

In contrast to sequential consistency, relaxed memory models require programmers to abandon their implicit and intuitive understanding of how memory behaves. Most of the relaxed models we have described require the programmer to reason with low level (and non-intuitive) reordering optimizations to understand the behavior of their programs. In addition, many of the models have been defined using complicated terminology, and in some cases, the original definitions have ambiguities which leave the semantics open to interpretation. These factors further exacerbate the difficulties in programming these models.

Another difficulty with relaxed models is the lack of compatibility among the numerous models and systems in existence. Many of the subtle differences among models make little difference in the actual performance of a model. However, such differences make the task of porting programs across different systems quite cumbersome. Similarly, the variety of models in existence make it difficult for programmers to adopt a programming methodology that works across a wide range of systems.

With all their shortcomings, relaxed models are widely used in many commercial multiprocessor systems, including systems designed by major computer manufacturers such as Digital Equipment, IBM, and Sun Microsystems. The wide-spread use of these systems suggests that even though sequential consistency is simpler to use for programmers, performance often plays an important role in the ultimate choice made by system designers and programmers. Nevertheless, we would ideally like to provide the extra performance with as little programming complexity as possible.

2.5 How to Evaluate a Memory Model?

Designing a new memory model that provides a higher performance potential is in some ways quite trivial. After all, performance can often be enhanced by simply ensuring fewer program orders and reducing atomicity constraints on memory operations. The challenge lies in providing a balanced design that appropriately trades off diminishing programming ease for the extra performance that is attained. The lack of a metric for objectively quantifying programming ease further complicates the choice for a system designer.

In what follows, we discuss some important issues in evaluating the trade-off between programming ease and performance.

2.5.1 Identifying the Target Environment

The first step in evaluating a memory model is to identify the environment for which it is intended. Evaluating the programming ease of a memory model requires identifying the target programmers and the target applications. For example, sophisticated system programmers will likely have an easier time with complicated models because they are accustomed to dealing with low-level abstractions and system details. Similarly, it is easier to reason about the memory behavior of well-structured applications with synchronized access to data structures as compared to applications with unstructured access to shared data.

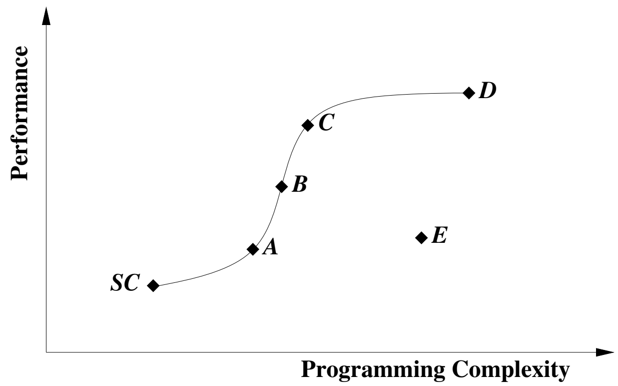


Figure 2.25: Trade-off between programming ease and performance.

Evaluating the performance of a memory model also requires identifying the target system and architecture. Specifically, it is important to identify the components in a system that may benefit from the flexibility provided by a relaxed model. The attainable performance gains can vary greatly if we are focusing on the hardware, the compiler, or both. Similarly, architectural assumptions play an important role; the performance gains from exploiting a relaxed model can be quite different depending on whether we are considering a tightly coupled shared-memory multiprocessor or a loosely coupled system that supports shared memory in software. Finally, the achievable gains can depend heavily on the characteristics of the target applications.

2.5.2 Programming Ease and Performance

Figure 2.25 shows the conceptual design space for memory models with respect to performance and programming complexity. The trends shown in this figure should be considered only as a coarse guideline for providing some intuition about the design space. The programming complexity axis is a measure of programmer effort required to develop a correctly functioning program that exploits the flexibility of a given model. The performance axis reflects the performance of the overall system, including both the compiler and underlying architecture. Each point in this space corresponds to a system that exposes some level of memory operation reordering and overlap to the programmer. The point labeled SC corresponds to a system that supports sequential consistency. Such a system has the advantage of low programming complexity, but provides low performance.

The figure shows a curve that corresponds to the *envelope of maximum performance* in the design space. In other words, systems that lie on the curve provide the highest performance for a given level of programming complexity. The shape of this curve can provide some intuition about the design choices. We have shown four hypothetical systems (denoted as *A*, *B*, *C*, and *D*) that lie on the curve, each successively exploiting more optimizations to achieve a higher performance. As we move from *SC* to *A*, there is a large increase in programming complexity before we observe any noticeable performance gains. The intuitive explanation for the sudden increase in programming complexity is that as soon as we move to a non-SC system, the programmer can no longer rely on the intuitive orders maintained by SC and is forced to explicitly reason with the low-level reordering optimizations. After this point, the curve is shown to be steep as we move to systems *B* and *C*. Since the programmer is already burdened with directly reasoning with reorderings, exploiting more memory reorderings can be quite effective because we can achieve large gains in performance

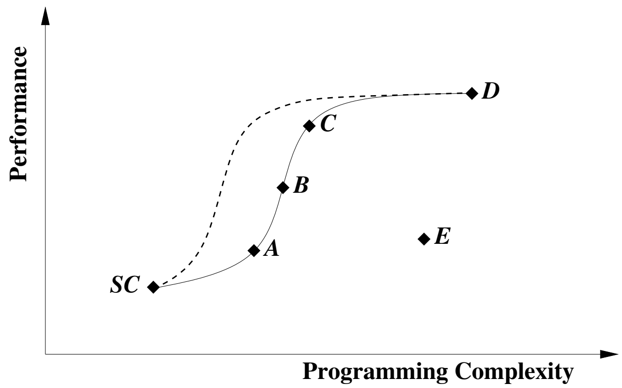


Figure 2.26: Effect of enhancing programming ease.

with a marginal increase in programming complexity. Eventually, exploiting more optimizations provides diminishing returns in performance (e.g., we have achieved most the gains that are possible from allowing memory operations to execute out of order) and can cause a jump in programming complexity since it becomes difficult to write correct programs given an extreme lack of order among memory operations. This effect is shown as a leveling off of the curve when we go from system *C* to system *D*. Given the non-linear shape of the curve, the most sensible design point for a non-SC system probably lies at the second knee of the curve, near point *C*.

It is important to note that not every system falls on the maximum performance curve (e.g., consider point *E*). In fact, one of the challenges in designing a relaxed model is to at least ensure that it provides close to the highest performance possible for its level of programming complexity. Given the large number of models that have been proposed in the literature, it is not difficult to find poorly designed models that fail in this respect.

2.5.3 Enhancing Programming Ease

The relaxed models we have discussed so far provide the programmer with an abstract specification for the low level reordering and overlap optimizations that are allowed among memory operations. Because these models directly expose the programmer to reordering optimizations in a system, we refer to them as *system-centric* models.

The next chapter will present an alternative category of memory models, called *programmer-centric* models. Programmer-centric models present the programmer with a higher-level abstraction of the system, thus relieving the programmer from directly reasoning with low-level reordering optimizations. To enhance programming ease, programmers are allowed to reason with sequential consistency. Programmers can enhance performance by providing intuitive program-level information about the behavior of memory operations in their programs (e.g., identifying synchronization operations). This information is in turn used by the system to determine the optimizations that may be safely exploited without violating sequential consistency for the given program. We will show that programmer-centric models can achieve better or comparable performance to system-centric models, and yet provide a much higher level of programming ease. This effect is conceptually depicted in Figure 2.26 as a shift of the performance curve towards lower programming complexity. By enhancing the programming ease of relaxed models, the programmer-centric approach makes it easier to justify the use of relaxed memory models over sequential consistency.

2.6 Related Concepts

This section presents a brief comparison of sequential consistency with other related correctness conditions such as *linearizability* [HW90] and *serializability* [Pap86].

Linearizability is a correctness condition proposed by Herlihy and Wing [HW90] for concurrent operations on shared objects. Shared objects are typically represented by abstract data types. For example, an object may represent a queue with corresponding operations such as enqueue and dequeue. Unlike sequential consistency, linearizability implicitly assumes the notion of an observable global time across all processes. Operations are modeled by an interval which consists of the period of time between the invocation and response for the operation and each operation is assumed to take effect instantaneously at some point within this interval. A processor does not issue an operation until it receives the response to its previous operation.

Linearizability can be trivially transformed to a correctness condition on memory operations by limiting objects to only represent shared memory locations with corresponding read and write operations. When interpreted in this way, linearizability is similar to sequential consistency because it requires valid executions to represent a total order on all operations that is consistent with individual program orders for each of the processes. However, linearizability is a stricter correctness condition than sequential consistency because it requires the actual order in which operations were executed in time to be reflected in this total order. For example, consider the program segment in Figure 2.3(a). Assume an actual execution of this program with operations executing in the time order (a1,b1,a2,b2) and with outcome (u,v)=(0,0). While this execution and outcome are sequentially consistent because it *appears as if* the operations were done in the order (a2,b2,a1,b1) to produce (u,v)=(0,0), the execution is not linearizable because the total order that explains the outcome is not consistent with the actual times at which the operations were executed. The stricter nature of linearizability naturally makes it a less efficient model to implement compared to sequential consistency. Furthermore, there are no compelling arguments that justify preserving linearizability over sequential consistency as the base correctness condition for multiprocessors.

Serializability is the correctness condition that is commonly assumed by most databases and distributed systems [Pap86]. In serializability, parallel threads consist of separate transactions each executing a finite sequence of operations to a set of objects shared with other transactions. An execution is considered *serializable* if it *appears as if* all transactions were executed in a sequential order with no interleaving within transaction boundaries. *Strict serializability* further constrains this condition by requiring this sequential order to correspond to the order in which the transactions are executed in real time [Pap86]. This difference between serializability and strict serializability is analogous to the difference we noted above between sequential consistency and linearizability.

Even though there are superficial similarities between sequential consistency and serializability, the two correctness conditions target different system and problem domains. Serializability is appropriate for systems such as databases and allows programmers to reason about transactions without considering the concurrency that the underlying system exploits. On the other hand, sequential consistency is targeted for programmers who write explicitly parallel programs and are willing to reason with the concurrency in a multiprocessor system. We refer the interested reader to Herlihy and Wing's paper [HW90] for a more thorough discussion of some of the theoretical differences among the correctness conditions discussed in this section.

2.7 Summary

This chapter presented the background information for the remainder of the thesis. We motivated the need for a memory consistency model for the purpose of specifying the behavior of memory operations, and introduced the notion of sequential consistency as an intuitive model for shared-memory multiprocessors. We next considered some of the architecture and compiler optimizations that are desirable in multiprocessors, and showed that the majority of these optimizations violate the semantics of sequential consistency. This led to the discussion of alternative memory models that enable a higher degree of performance by relaxing some of the constraints imposed by sequential consistency. Choosing among these models requires considering fundamental trade-offs between programmability, portability, implementation complexity, and performance. The remaining chapters in this thesis are devoted to a thorough analysis of each of these issues with the hope of clarifying and resolving the various trade-offs.

Chapter 3

Approach for Programming Simplicity

This chapter presents our approach for enhancing the programmability of relaxed memory models while maintaining the performance advantages associated with the system-centric models described in the previous chapter. We propose an alternative method for specifying memory behavior that presents a higher level abstraction to the programmer. Instead of requiring the programmer to deal with the complex semantics of system-centric models, we allow programmers to reason with sequential consistency and simply require them to provide program-level information about shared memory operations. This information is then exploited to provide higher performance. Models developed using this framework are referred to as programmer-centric models.

Section 3.1 provides an overview of the programmer-centric approach. We next present a set of related programmer-centric models that successively exploit more information about memory operations to achieve higher performance. Sections 3.3 and 3.4 compare the above programmer-centric models with system-centric models in terms of performance and ease of use. Section 3.5 describes practical ways that may be used by programmers to convey the required information about memory operations. We discuss the relevance of programmer-centric models to programs that exhibit frequent unsynchronized data accesses in Section 3.6. Finally, Sections 3.7 and 3.8 discuss possible extensions to this work and describe the related work in this area.

3.1 Overview of Programmer-Centric Models

Programmer-centric models provide an alternative approach to system-centric models for specifying the memory behavior of a system. The primary goal of programmer-centric models is to achieve the high performance associated with system-centric models while maintaining the programming ease of sequential consistency. We describe the basic foundation for this approach below.

The programmer-centric approach is based on the premise that programmers prefer to reason with an

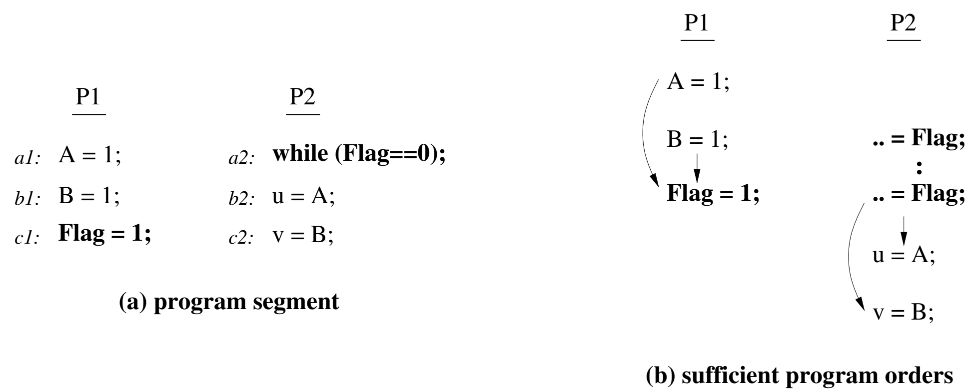


Figure 3.1: Exploiting information about memory operations.

intuitive model such as sequential consistency. Therefore, the programmer is allowed to reason with sequential consistency. To allow for higher performance, we require the programmer to provide program-level information about the behavior of shared-memory operations. This information is then used by the system to determine the memory reordering optimizations that can be exploited safely without affecting correctness.

Figure 3.1(a) shows an example program segment to motivate the use of program-level information about memory operations. The program shows a producer-consumer interaction between two processors, where P1 writes to two memory locations and sets a flag and P2 waits for the flag to be set before reading the two locations. A straightforward implementation of sequential consistency would maintain all program orders in this example, thus disabling overlap and reordering of memory operations within each processor. However, as we discussed in Section 2.3.2, maintaining all program orders is not necessary for achieving sequentially consistent results in most programs. Figure 3.1(b) shows a sufficient set of program orders that guarantee sequential consistency (SC) in this example. The multiple reads of Flag correspond to reading the old value multiple times before the set value is read. As shown, reordering the writes to A and B on P1, or the reads of A and B on P2, does not affect the correctness of the execution; the result of the execution with reordering is indistinguishable from the result with no reordering. The intuitive reason for this is that the flag synchronization ensures there is only one processor accessing locations A and B at any given time. Once P2 proceeds past its waiting loop, P1 is already done with its writes to A and B. Therefore, reordering the reads to A and B does not change the result of the execution since each read occurs after the write to the same location. A similar argument holds for reordering the writes to A and B on P1.

Ideally, we would like to automatically identify (e.g., through compiler analysis) the memory operations in a program that can be safely reordered and overlapped without violating sequential consistency. Shasha and Snir have proposed such an approach [SS88]. However, as we pointed out in Section 2.3.2, the success of such approaches has been limited to simple programs written with restrictive languages or programming models.

Instead of depending on automatic techniques, we require the programmer to provide explicit information about the usage and semantics of memory operations, which allows the system to easily determine the set of orders that may be safely violated. For example, referring back to Figure 3.1, simply knowing that the operations to the flag location function as synchronization allows us to determine that reordering the operations to the other locations is safe. The information we require is often naturally known by the programmer. Furthermore, with only a few types of information supplied by the programmer, we can exploit the full range of optimizations captured by the aggressive system-centric models.

Overall, the programmer-centric approach unifies the memory optimizations captured by the system-centric models without sacrificing programming ease or portability. Providing the extra program-level information is easier and more natural than reasoning directly with the system-centric models. In addition, as long as the information provided by the programmer is correct, the system guarantees sequentially consistent results and yet performs comparably to the most aggressive system-centric models.

3.2 A Hierarchy of Programmer-Centric Models

This section presents a hierarchy of programmer-centric models that exploit information about memory operations to provide higher performance. We describe the categorization that can be used to distinguish different memory operations based on their semantics and behavior in sequentially consistent executions of a program. This information is used by the system to determine whether a given operation can be safely executed out of program order or non-atomically without violating sequential consistency. The hierarchy of programmer-centric models described here is an extension of our earlier work on the PL [GLL+90] and PLpc [GAG+92] models which are described in more detail in Section 3.8.

We use the notion of a *label* associated with each memory operation as an abstraction for distinguishing the operation based on its categorization. Section 3.5 discusses practical techniques for allowing programmers to convey such information about memory operations. For now, we assume that the label is conveyed along with the static memory instructions in a program; the execution of the instruction generates a memory operation with the associated label. A program that provides correct labels (will be defined shortly) for its memory operations is referred to as a *properly-labeled (PL) program*. Given a properly-labeled program, a canonical programmer-centric model guarantees that all executions of the program will be sequentially consistent. We refer to programmer-centric models that are based on the above framework as *properly-labeled (PL) models*.

We will present three properly-labeled models that successively exploit more information about memory operations to achieve higher performance. The *first* type of information categorizes a memory operation based on whether it executes simultaneously with other operations to the same location. This information effectively identifies conflicting operations that are involved in a race; we refer to such operations as *competing* operations. The *second* type of information distinguishes between competing operations based on whether they are used to order or synchronize operations to other locations. Finally, the *third* type of information identifies a common use of competing operations for synchronization where one processor waits on a location until a particular value is written by another processor. Among the above three types of information, the first type is by far the most important in allowing the system to achieve higher performance relative to a straightforward implementation of SC. The latter two types of information become more relevant if competing operations are frequent. The categorization described here can be extended to include other useful information about memory operations. However, there is a trade-off between how easy it is for the programmer to provide the extra information and what incremental performance benefits it can provide.

Before presenting the models, we clarify some terminology and assumptions that will be used throughout this section. Some of this terminology was introduced in Chapter 2. Memory operations are modeled as a single read or a single write access to a memory location. Read-modify-write operations are modeled as a separate read and write operation. Two operations *conflict* if they are to the same location and at least one of them is a write operation. A system is considered an *SC system* if it obeys sequential consistency. An execution is an *SC execution* if it is possible on an SC system. By *every* SC execution, we mean all SC executions of the program for all valid inputs and initial states of memory. Memory operations appear to execute in some total order in an SC execution, referred to as the *execution order*. Given an execution order, the *conflicting order* ($\xrightarrow{co}$) is the order imposed on conflicting memory operations.[1] Finally, for every execution of a program, the *program order* ($\xrightarrow{po}$) represents a partial order on all memory operations that is consistent with the per-processor total order on memory operations imposed by the program for each processor.

The following sections describe each model in detail and provide intuition for how the extra information about memory operations can be exploited by the system to enable further reordering and overlap of memory operations. Chapter 4 provides the more precise set of system requirements for guaranteeing sequential consistency for properly-labeled programs, and describes how properly-labeled programs can be efficiently ported to systems that support a system-centric model.

3.2.1 Properly-Labeled Model—Level One (PL1)

The first programmer-centric model we consider requires the programmer to identify the memory operations that may be involved in a race. The following describes the formal categorization of shared-memory operations used by this model and provides intuition for the optimizations that can be exploited by a system based on such a categorization.

Categorization of Shared-Memory Operations for PL1

We use the example program from Figure 3.1 to illustrate the intuition behind competing operations, which are memory operations that are involved in a race. Figure 3.2 shows the program order and conflict order for one execution of this program. In the SC execution shown, the conflicting operations to location A, and to location B, are synchronized or ordered by operations to the Flag location. In contrast, conflicting accesses to the Flag location are not ordered through other operations. Therefore, we refer to the memory operations to Flag as *competing* operations and to A and B as *non-competing* operations. The formal definitions follow. Appendix A presents a slightly different definition for an ordering chain and explains why we chose Definition 3.1 below.

Definition 3.1: Ordering Chain
Given two conflicting operations *u* and *v* in an execution, an *ordering chain* exists from operation *u* to operation *v* if and only if
(a) $u \xrightarrow{po} v$, or
(b) $u \xrightarrow{po} w_1 \xrightarrow{co} r_1 \xrightarrow{po} w_2 \xrightarrow{co} r_2 \xrightarrow{po} w_3 \ldots \xrightarrow{co} r_n \xrightarrow{po} v$, where $n \geq 1$, w_i is a write access, r_j is a read access, and w_i and r_j are to the same location if $i = j$. If all accesses in this chain are to the same location, then *u* may be the same as w_1, and *v* may be the same as r_n, as long as there is at least one $\xrightarrow{po}$ arc in the chain.

Definition 3.2: Competing and Non-Competing Operations
Given a pair of conflicting operations *u* and *v* in an execution, operation *u competes* with operation *v* if and only if there is no ordering chain (as defined above) between *u* and *v* (either from *u* to *v* or vice versa). An operation *u* is a *competing operation* if and only if there exists at least one conflicting operation *v* in this execution that competes with *u*. Otherwise, the operation is a *non-competing operation*.

[1] This order is not transitive (i.e, not a partial order) because it does not hold between two reads to the same location.

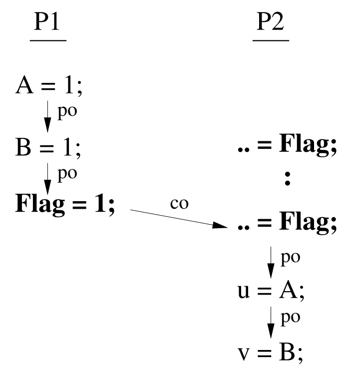


Figure 3.2: Example of program order and conflict order.

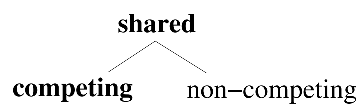


Figure 3.3: Categorization of read and write operations for PL1.

Referring back to Figure 3.2, there is an ordering chain between the conflicting operations to locations A and B in the execution that is shown. However, there is no ordering chain between the write and read operations to Flag. Therefore, operations to Flag are indeed competing while the remaining operations to the other locations are non-competing. Although the formal definition for competing operations seems complex, it essentially captures the fairly intuitive notion of a race. Thus, programmers should be able to identify such operations relatively easily.

Figure 3.3 shows the categorization of shared-memory operations discussed above. As we will see shortly, the PL1 model also distinguishes between competing read and competing write operations and exploits this distinction to provide more optimizations. However, since determining whether an operation is a read or a write is trivial and can be done automatically, we do not include it in the categorization that is visible to the programmer. Given the categorization shown in the figure, the valid labels for memory operations are competing and non-competing. Note that operations that access the same location can have differing labels; e.g., one operation may be competing while another is non-competing.

Since a system exploits the information conveyed by the labels to ensure correctness, a label needs to have a proper relationship to the actual category of a memory operation. We consider the labeling *correct* if every memory operation is labeled with the category that it actually belongs to. However, providing labels that exactly correspond to the actual category of an operation requires perfect information about the behavior of memory operations. Since it may be difficult for the programmer to obtain such exact information, we provide the flexibility of allowing *conservative labels*. Intuitively, a label is conservative if it leads to fewer reordering and overlap optimizations by a system as compared to the actual category of the operation. The conservative labels for PL1 are shown in bold in Figure 3.3. As we will see, the system exploits fewer optimizations for competing operations as compared to non-competing operations. Therefore, labeling an operation that is actually non-competing as competing still leads to correct (i.e., sequentially consistent) executions. In contrast, labeling an operation that is intrinsically competing as non-competing is not safe. While allowing conservative labels can substantially enhance programming ease, it is important to realize that frequent use

	P1		P2
a1:	**A = 1;**	*a2:*	**u = B;**
b1:	**B = 1;**	*b2:*	**v = A;**

(a)

	P1		P2
a1:	while **(test&set(L)==0);**	*a2:*	while **(test&set(L)==0);**
b1:	A = 1;	*b2:*	u = B;
c1:	B = 1;	*c2:*	v = A;
d1:	**L = 0;**	*d2:*	**L = 0;**

(b)

Figure 3.4: Program segments with competing operations.

of conservative labels can reduce system performance.

Below, we introduce the notion of proper labeling to formally capture the correctness property for labels. The subscript L is used to distinguish an operation's label from the operation's intrinsic category.

Definition 3.3: Properly-Labeled Execution—Level One (PL1)
An execution is a *properly-labeled (PL1) execution* if and only if all operations labeled *non-competing*$_L$ are *non-competing* (i.e., *non-competing*$_L \subseteq$ *non-competing*), and the remaining operations are labeled *competing*$_L$.

Definition 3.4: Properly-Labeled Program—Level One (PL1)
A program is a *properly-labeled (PL1) program* if and only if all sequentially consistent executions of the program result in properly-labeled (PL1) executions.

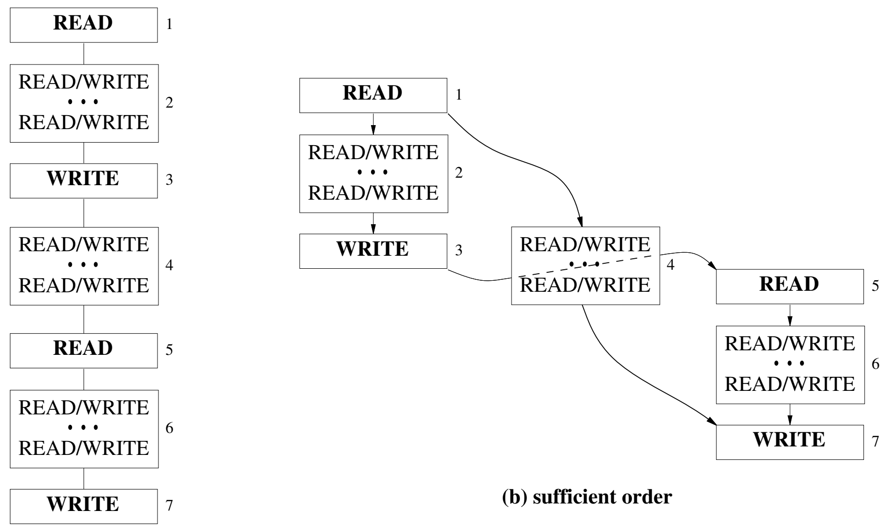
Note that the programmer can directly reason with sequential consistency to determine whether a program is properly-labeled.[2] Figure 3.4 shows a couple of program segments with competing operations. The memory instructions that can potentially generate a competing operation are shown in bold; i.e., there is at least one SC execution of the program where the operation that is generated by the instruction is competing. Below, we overload the competing label to refer to the instruction as well. Consider the program segment in Figure 3.4(a). For each instruction, there is at least one SC execution where the operation that is generated by the instruction competes with another conflicting operation. Therefore, all instructions are competing. Figure 3.4(b) shows a similar program segment except now the memory instructions are encapsulated within critical sections. There is still some non-determinism since either P1 or P2 may enter the critical section first. Nevertheless, in every SC execution, memory operations to locations A and B are non-competing because the lock synchronization provides an ordering chain separating the conflicting pairs. In contrast, the instructions that access location L are competing.

Given the above definitions, we can now define the properly-labeled (PL1) memory model. The basic idea behind this model is to guarantee sequential consistency for programs that are properly-labeled. The formal definition follows.

Definition 3.5: The Properly-Labeled Memory Model—Level One (PL1)
A system obeys the *PL1 memory model* if and only if for any PL1 program (defined above), all executions of the program on this system are sequentially consistent.

An interesting observation is that the above definition specifies the memory behavior for properly-labeled programs only; the behavior of programs that are not properly-labeled is unspecified. Of course, any practical implementation of the above model will have a specified behavior for all programs, regardless of whether the

[2]It is conceptually possible for an instruction in a program to never generate an operation in any SC execution. Naturally, the decision on whether the program is a PL1 program does not depend on the label on such instructions. Therefore, the programmer is free to use any label for such instructions.



(a) program order

(b) sufficient order

Figure 3.5: Possible reordering and overlap for PL1 programs.

program is properly-labeled. Nevertheless, the above under-specification provides implementations with an extra flexibility in the behavior they support for non-PL programs.

Possible Optimizations for PL1

As we discussed in Chapter 2, a simple and sufficient way to maintain sequential consistency is to (i) execute memory operations from each processor one at a time and in program order, and (ii) ensure write operations execute atomically (i.e., as if there is a single copy of memory). The information conveyed by the labels in a PL1 program allows the system to relax some of these constraints while maintaining sequential consistency.

The system can exploit the distinction between competing and non-competing operations, and between competing read and competing write operations. Figure 3.5 illustrates the overlap and reordering that is possible among the operations from a single processor. Figure 3.5(a) shows a sample sequence of memory operations in program order. The operations shown in bold are competing operations. Each block of multiple operations depicts a sequence of non-competing operations between two competing ones. The numbers beside each block uniquely identify the block. Figure 3.5(b) shows the sufficient program orders that can be maintained to satisfy the PL1 model. Within each block of non-competing operations, reads and writes to different locations can be overlapped or reordered. The system can trivially distinguish between competing read and competing write operations to allow extra overlap between competing and non-competing operations. A competing read is only ordered with respect to operations that follow it in program order, while a competing write is only ordered with respect to operations that precede it in program order. This is analogous to the distinction between acquire and release operations in the release consistency model (see Section 2.4 of the previous chapter), and arises from the fact that the interprocessor links in an ordering chain (Definition 3.1) are always from a competing write to a competing read. Finally, program order is maintained among competing operations. Regarding atomicity, non-competing writes are allowed to appear non-atomic.

PL1 programs can be easily ported to a system-centric model such as release consistency (RCsc) to exploit most of the above optimizations; competing reads and writes can be mapped to acquire and release operations, respectively, and non-competing operations can be simply mapped to ordinary data operations. Chapter 4 describes a slightly more aggressive set of system requirements (relative to RCsc) that still satisfies the PL1 model. Overall, the most important optimization is the ability to reorder memory operations in between competing operations, which can provide a substantial performance potential especially if competing operations are infrequent.

3.2.2 Properly-Labeled Model—Level Two (PL2)

The second programmer-centric model we consider requires extra information about memory operations to identify competing operations that are directly used for synchronizing non-competing operations. Below, we describe the formal categorization of operations and the types of optimizations that are enabled by this extra information.

Categorization of Shared-Memory Operations for PL2

This section extends the categorization of memory operations to identify the subset of competing operations that are used to order non-competing operations through ordering chains. We refer to such operations as *synchronization* (abbreviated to *sync*) operations and the remaining competing operations are referred to as *non-synchronization* (or *non-sync*) operations.

Consider the program segment in Figure 3.4(b) (from the previous section) that shows two processors accessing memory locations within critical sections implemented using test-and-set operations. As we discussed in the previous section, the operations to locations A and B are non-competing because in every SC execution, there is an ordering chain between conflicting pairs of operations to these locations. Therefore, the read and write of the test-and-set and the write to unset the lock are the only competing operations in the program. By considering SC executions of this program, it becomes apparent that while the write to unset the lock and the read of the test-and-set are required to form ordering chains among conflicting operations to locations A and B, the write of the test-and-set is not part of the ordering chains. Intuitively, the write to unset the lock acts as a *release* operation and informs the other processor that operations that appear before it in program order have completed. Similarly, the read of the test-and-set acts an *acquire* operation which, in conjunction with the while loop, delays future operations that follow it in program order until a release is done by another processor. In contrast, the write of the test-and-set is simply used to ensure mutual exclusion and does not function as either an acquire or a release. Therefore, we categorize the read of the test-and-set and the write to unset the lock as synchronization operations and the write of the test-and-set as a non-synchronization operation.

Figure 3.6 shows the categorization of shared-memory operations for the PL2 memory model. Given this categorization, the valid labels for memory operations (i.e., leaves of the categorization tree) are *sync*, *non-sync*, and *non-competing*. As discussed above, read synchronization operations function as an *acquire*, while write synchronization operations function as a *release*. This extra distinction between read and write synchronization operations will also be used to allow further optimization, but is not shown as explicit labels since the distinction between reads and writes is trivial and automatic. The conservative labels are shown in bold in Figure 3.6. For example, operations that are non-competing or non-sync can conservatively be

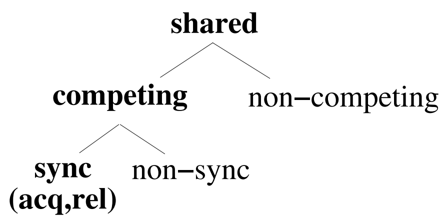


Figure 3.6: Categorization of read and write operations for PL2.

labeled as sync. Below, we formalize the notion of proper labeling based on the above categorization.

Definition 3.6: Sufficient Synchronization Ordering Chains
Given an execution, there are *sufficient synchronization ordering chains* if and only if for every operation u labeled as $non\text{-}competing_L$ and any operation v from a different processor that conflicts with u, there is at least one ordering chain (see Definition 3.1) either from u to v, or from v to u, such that every w_i and r_i in the chain is labeled as $sync_L$.

Definition 3.7: Properly-Labeled Execution—Level Two (PL2)
An execution is a *properly-labeled(PL2) execution* if and only if all operations labeled $non\text{-}competing_L$ are *non-competing* (i.e., $non\text{-}competing_L \subseteq non\text{-}competing$), enough operations are labeled $sync_L$ to ensure sufficient synchronization ordering chains (defined above), and the remaining operations are labeled as $non\text{-}sync_L$.

Definition 3.8: Properly-Labeled Program—Level Two (PL2)
A program is a *properly-labeled (PL2) program* if and only if all sequentially consistent executions of the program result in properly-labeled (PL2) executions.

Figure 3.7 shows a two-processor program segment that further illustrates the use of synchronization and non-synchronization operations. Recall that uppercase identifiers denote shared locations and lowercase identifiers denote private locations. The program depicts part of a branch-and-bound algorithm where each processor attempts to update the global bound in case the locally computed bound is smaller. This type of algorithm may be used to solve the traveling-salesperson problem, for example, with the bound representing the current shortest path among the cities. In this type of an algorithm, each processor may check the global bound many more times than it actually updates the bound. Therefore, for efficiency, each processor reads the current global bound in an unsynchronized fashion (as a hint) and obtains mutual exclusion for updating the bound only if the locally computed bound is smaller. The competing operations are shown in bold. In addition, we show a set of valid labels for the memory instructions in this program. The test-and-set is used to achieve mutual exclusion. As in our earlier discussion, the read of the test-and-set and the write to unset the lock are synchronization operations, while the write of the test-and-set is a non-synchronization competing operation. The read of Bound within the critical section is non-competing since in every execution there exists an ordering chain between it and any conflicting operations. In contrast, the read of Bound outside the critical section and the write of Bound within the critical section are competing; however, they are non-synchronization operations since they do not participate in any ordering chains for non-competing operations.

The definition of the PL2 model is similar to the definition in the previous section (Definition 3.5), except we use the new definition for PL2 programs now.

Definition 3.9: The Properly-Labeled Memory Model—Level Two (PL2)
A system obeys the *PL2 memory model* if and only if for any PL2 program (defined above), all executions of the program on this system are sequentially consistent.

```
        P1                                  P2
        ...                                 ...
a1: u = Bound;                      a2: u = Bound;
b1: if (local_bound < u) {          b2: if (local_bound < u) {
c1:     while (test&set(L)==0);     c2:     while (test&set(L)==0);
d1:     u = Bound;                  d2:     u = Bound;
e1:     if (local_bound < u)        e2:     if (local_bound < u)
f1:         Bound = local_bound;    f2:         Bound = local_bound;
g1:     L = 0;                      g2:     L = 0;
h1: }                               h2: }
        ...                                 ...
```

Labels

a1,a2: non−sync$_L$
c1,c2 (test): sync$_L$(acq)
c1,c2 (set): non−sync$_L$
d1,d2: non−competing$_L$
f1,f2: non−sync$_L$
g1,g2: sync$_L$(rel)

Figure 3.7: Program segment from a branch-and-bound algorithm

Possible Optimizations for PL2

The distinction between competing operations that are used for synchronization versus those that are not enables the system to further relax the order among memory operations relative to the PL1 model. The main optimization that is enabled by the sync/non-sync distinction is with respect to the program order between competing and non-competing operations. The program order between a sync read (acquire) and non-competing operations that follow it, and a sync write (release) and non-competing operations that precede it is maintained. However, the program order between competing and non-competing operations to different locations need not be maintained if the competing operations are non-sync operations. PL2 programs can be ported to a model such as release consistency (RCsc) to exploit the above relaxation; competing sync read and write operations should be mapped to acquire and release, respectively, competing non-sync operations should be mapped to non-sync, and non-competing operations should be mapped to ordinary data operations. Again, Chapter 4 describes a slightly more aggressive set of system requirements (relative to RCsc) that still satisfies the PL2 model.

Figure 3.8 provides an example to illustrate the overlap between operations. Figure 3.8(a) shows the program order among memory operations from a single processor. Assume the only competing operations in the sequence are the read and write of a test-and-set and the write that unsets the lock (shown in bold). The dashed box around the test-and-set signifies the atomicity of the read and write operations. As in our previous examples, assume the labels on the test and the write to unset the lock are sync$_L$ (acquire and release, respectively), and the label on the set is non-sync$_L$. Figure 3.8(b) shows the sufficient program orders for satisfying the PL2 memory model. Even though the write of the test-and-set is a competing operation, its program order with respect to non-competing operations need not be maintained since it is identified as a non-sync. One of the implications is that the test-and-set can be overlapped and reordered with respect to non-competing operations that precede it in program order. Note that this optimization is not necessarily safe unless we know that the set of the test-and-set does not play a role in the ordering chains among non-competing operations (i.e., it is not a sync).

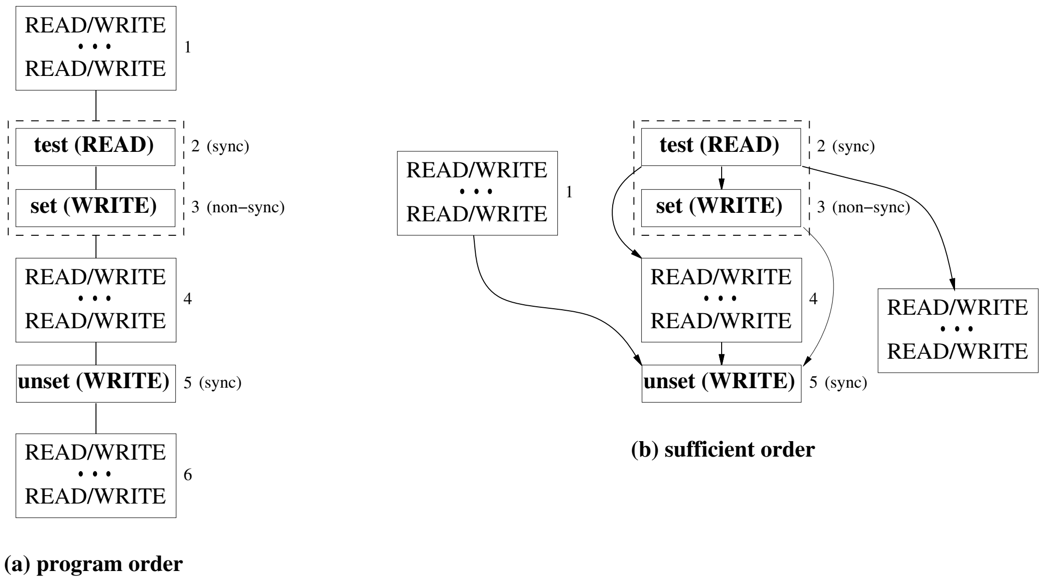


Figure 3.8: Possible reordering and overlap for PL2 programs.

3.2.3 Properly-Labeled Model—Level Three (PL3)

The third programmer-centric model we consider requires further information from the programmer to identify a common use of competing synchronization operations where one processor waits on a location until a particular value is written by another processor. The synchronization examples we have considered up to now, including the use of a flag to synchronize producer-consumer interactions or the use of test-and-set to implement locks, fall in this category. Below we describe the categorization of operations that allows us to correctly identify such synchronizations and discuss the optimizations that are enabled by such information.

Categorization of Shared-Memory Operations for PL3

To motivate the categorization described below, consider the program segment shown in Figure 3.1(a). Based on the categories we have already discussed, the operations to Flag are the only competing operations and are used to synchronize accesses to locations A and B. The way this synchronization is achieved is by P2 reading Flag within a loop until P1 writes the value of 1 to this location. We refer to such a loop as a *synchronization loop construct*. Below, we formalize a simple case of a synchronization loop construct in which the loop repeatedly executes a read or a read-modify-write to a specific location until it returns one in a set of specified values.[3]

Definition 3.10: Synchronization Loop Construct
A *synchronization loop construct* is a sequence of instructions in a program that satisfies the following:
(a) The construct executes a read or a read-modify-write to a specific location. Depending on whether the value returned is one of certain specified values, the construct either terminates or repeats the above.
(b) If the construct executes a read-modify-write, then the writes of all but the last read-modify-write store values that are returned by the corresponding reads.
(c) The construct terminates in every SC execution.

[3]The definition for a synchronization loop construct and for loop/non-loop reads and writes (Definitions 3.11 and 3.12) are similar to those used for the PLpc model [GAG+92, AGG+93], which will be discussed in Section 3.8.

The above definition captures many of the common uses for such synchronization constructs. Appendix B provides a more general set of conditions that, for example, allows implementations of locks using test-and-test-and-set [RS84] or back-off [MCS91] techniques to also be considered as synchronization loop constructs. Finally, Appendix F describes an optional restriction of synchronization loop constructs that slightly simplifies the required system conditions for supporting the PL3 model.

Given that a synchronization loop construct eventually terminates, the number of times the loop executes or the values returned by its unsuccessful reads cannot be practically detected and should not matter to the programmer. For example, in Figure 3.1(a), the number of time P2 reads Flag unsuccessfully, or even the values returned by the unsuccessful reads, should not matter as long as eventually a read of Flag returns the value 1 and terminates the loop. Therefore, *we do not consider the unsuccessful reads of a synchronization loop construct as part of the result or outcome for the program.* In other words, we assume that replacing the operations of a synchronization loop construct with *only* the last read or read-modify-write that causes the loop to terminate leads to an equivalent execution as far as outcomes are concerned. Note that this is a slightly relaxed interpretation of equivalence between two executions, and this relaxation allows more aggressive labels which in turn enable more optimizations. Finally, all executions that we consider (e.g., for deciding whether operation labels are correct) are assumed to be modified to replace every synchronization loop construct with only the last read or read-modify-write that causes the loop to terminate. We refer to this as a *modified execution*.

Operations associated with synchronization loop constructs often exhibit a special property that can be exploited by the system to achieve higher performance. For example, consider the write and read operations to Flag in Figure 3.1(a). In every SC execution of this program, the write to Flag by P1 must execute before P2's final read of Flag that terminates the synchronization loop. Therefore, while a competing pair of operations can in general occur in any order, certain write-read pairs such as the operations to Flag have a fixed order of execution. We refer to such write-read pairs as loop writes and loop reads. The formal definitions follow. As we will discuss shortly, the fact that the read effectively waits for the write can be exploited by the system to safely ignore certain program orders with respect to the read and to allow the write to behave non-atomically with respect to multiple copies.

Definition 3.11: Loop and Non-loop Reads
Given a competing read R in a modified execution, the read is a *loop read* if and only if
(a) it is the final read of a synchronization loop construct that terminates the construct,
(b) it competes with at most one write in the modified execution,
(c) if it competes with a write W in the modified execution, the write is necessary to make the synchronization loop construct terminate; i.e., the read returns the value of that write and the immediately preceding conflicting write in the execution order would not make the loop construct terminate, and
(d) let W' be the last conflicting write (if any) before W whose value could terminate the loop construct; if there exists any competing write W" (does not necessarily compete with R, W or W') to the same location between W' and W in the execution order (by definition, W" fails to terminate the loop construct), then there is an ordering chain from W" to R that ends with a $\xrightarrow{po}$.
A competing read that is not a loop read is a *non-loop read*.

Definition 3.12: Loop and Non-loop Writes
Given a competing write W in a modified execution, the write is a *loop write* if and only if
(a) it competes only with loop reads, and
(b) for any non-loop read R that conflicts with W and is after W, there is an ordering chain from W to R that ends with a $\xrightarrow{po}$.
A competing write that is not a loop write is a *non-loop write*.

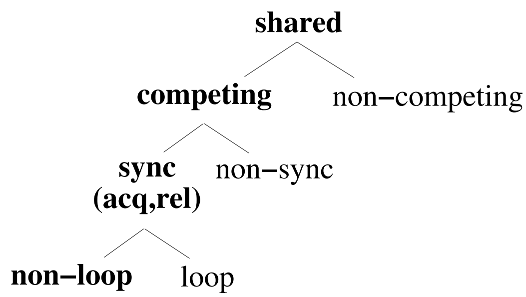


Figure 3.9: Categorization of read and write operations for PL3.

Appendix C motivates the need for the relatively complex conditions such as condition (d) for loop reads and condition (b) for loop writes above. In addition, the appendix describes a simplification of the PL3 model that forgoes some of the optimizations that are allowed by the more general form described here.

Figure 3.9 shows the extended categories of shared-memory operations. As shown, synchronization operations are now distinguished into two categories, loop and non-loop.[4] With this distinction, the valid labels (i.e., leaves of categorization tree) are *non-loop*, *loop*, *non-sync*, and *non-competing*. The conservative labels are shown in bold in the figure. As with the previous models, the system further distinguishes between read and write operations that are labeled as loop or non-loop. Again, since this distinction is trivial and automatic, we do not include it explicitly in the set of labels that is visible to the programmer. We next formalize the notion of correct labels.

Definition 3.13: Properly-Labeled Execution—Level Three (PL3)
A modified execution is a *properly-labeled (PL3) execution* if and only if all operation labeled *non-competing*$_L$ are *non-competing* (i.e., *non-competing*$_L \subseteq$ *non-competing*), all operations labeled *loop*$_L$ are either *loop* or *non-competing* operations (*loop*$_L \subseteq$ (*loop* $\cup$ *non-competing*)), enough operations are labeled *loop*$_L$ or *non-loop*$_L$ to ensure sufficient synchronization ordering chains (both loop and non-loop labels are considered as a sync in Definition 3.6), and the remaining operations are labeled as *non-sync*$_L$.

Definition 3.14: Properly-Labeled Program—Level Three (PL3)
A program is a *properly-labeled (PL3) program* if and only if all sequentially consistent executions (i.e., modified executions) of the program result in properly-labeled (PL3) executions.

Figure 3.10 shows the implementation of two common synchronization primitives that we will use to illustrate correct labels for PL3. Even though we show only two processors, the labels we discuss are still valid if the synchronization primitives are used across a larger number of processors. Figure 3.10(a) shows an implementation of locks using test-and-set and a write to unset the lock. The while loop on each processor that contains the test-and-set qualifies as a synchronization loop construct. Therefore, all operations (i.e., unsuccessful test-and-sets) except for the final successful test-and-set in an execution are ignored. In any such modified SC execution of this program, the test of a final test-and-set competes only with the write to unset the lock that is required for the loop to terminate. Therefore, the test is a loop read.[5] Similarly, the write to unset the lock competes only with a loop read (a successful test) and there are no non-loop reads that conflict with it; therefore, it qualifies as a loop write. Finally, the set of the final test-and-set does not compete

[4] It is also possible to further distinguish non-sync operations as loop and non-loop operations since the distinction between loop and non-loop operations is really orthogonal to whether the operation is a sync or non-sync. We chose not to do this for simplicity (and the fact that non-sync operations are typically infrequent), but the model can be trivially generalized in this way.

[5] This depends on the fact that we are discarding unsuccessful test-and-sets from the execution, since the test in unsuccessful test-and-sets do not qualify as loop reads, and the set of the unsuccessful test-and-sets would compete with the successful test thus disqualifying it as a loop read.

P1	P2	Labels
a1: while (**test&set(L)** == 0);	*a2:* while (**test&set(L)** == 0);	a1,a2 (test): $loop_L$ (acq)
accesses to data	*accesses to data*	a1,a2 (set): $non\text{-}competing_L$
b1: **L = 0;**	*b2:* **L = 0;**	b1,b2: $loop_L$ (rel)

(a)

```
P1                                        P2
accesses to data                          accesses to data
/* barrier code */                        /* barrier code */
a1: local_flag = ! local_flag;            a2: local_flag = ! local_flag;
b1: if (fetch&incr(Count) == N) {         b2: if (fetch&incr(Count) == N) {
c1:     Count = 1;                        c2:     Count = 1;
d1:     Flag = local_flag;                d2:     Flag = local_flag;
e1: } else {                              e2: } else {
f1:     while ( Flag != local_flag);      f2:     while ( Flag != local_flag);
g1: }                                     g2: }
accesses to data                          accesses to data
```

Labels

b1,b2 (fetch): $non\text{-}loop_L$ (acq)
b1,b2 (incr): $non\text{-}loop_L$ (rel)
c1,c2: $non\text{-}competing_L$
d1,d2: $loop_L$ (rel)
f1,f2: $loop_L$ (acq)

(b)

Figure 3.10: Example program segments: (a) critical section, (b) barrier.

with any operations (again, assuming we discard unsuccessful test-and-sets) and can therefore be labeled as non-competing.

Figure 3.10(b) shows the implementation of a barrier [MCS91] using an atomic fetch-and-increment operation. The while loop containing the reads of Flag forms a synchronization loop construct. Therefore, we ignore the unsuccessful reads of Flag. The write to Count is non-competing. In contrast, the fetch (read) and the increment (write) to Count and the write and the final read of Flag are competing. The read and write to Flag qualify as loop operations, while the fetch and the increment on Count are non-loop operations.

The formal definition for the PL3 model follows.

Definition 3.15: The Properly-Labeled Memory Model—Level Three (PL3)
A system obeys the *PL3 memory model* if and only if for any PL3 program (defined above), all modified executions of the program on this system are sequentially consistent.

Possible Optimizations for PL3

The distinction of sync operations into loop and non-loop categories enables the system to relax the program order among competing operations as compared to the previous two models we described. Specifically, given a competing write followed by a competing read, the program order between the write-read pair need not be maintained if either operation is identified with the loop label. In addition, competing writes that are identified with the loop label can be non-atomic with respect to multiple copies. It is possible to map PL3 programs

P1	P2	Labels
a1: A = 1;	*a2:* B = 1;	b1,b2: $loop_L$ (rel)
b1: **Flag1** = 1;	*b2:* **Flag2** = 1;	c1,c2: $loop_L$ (acq)
c1: while (**Flag2** == 0);	*c2:* while (**Flag1** == 0);	a1,a2,d1,d2: $non\text{-}competing_L$
d1: u = B;	*d2:* v = A;	

(a)

P1	P2	Labels
a1: A = 1;	*a2:* B = 1;	b1,b2: $loop_L$ (rel)
b1: **unset(L1);**	*b2:* **unset(L2);**	c1,c2 (test): $loop_L$ (acq)
c1: while (**test&set(L2)** == 0);	*c2:* while (**test&set(L1)** == 0);	c1,c2 (set): $non\text{-}competing_L$
d1: u = B;	*d2:* v = A;	a1,a2,d1,d2: $non\text{-}competing_L$

(b)

Figure 3.11: Example program segments with loop read and write operations.

to RCpc in order to exploit the above relaxations. Chapter 5 describes the mapping to RCpc, in addition to presenting a more aggressive set of constraints that still satisfy the PL3 model.

Figure 3.11 shows a couple of program segments to provide intuition for the program reordering optimization described above. The first example in Figure 3.10(a) shows two processors communicating data. Each processor produces a value, sets a flag, waits for the other processor's flag to be set, and consumes the value produced by the other processor. The operations to Flag1 and Flag2 are competing and are shown in bold. These operations all qualify as loop operations. The optimization discussed above allows the write of one flag and the read of the other flag to be overlapped on each processor, e.g., read of Flag1 can be reordered with respect to the write of Flag2 on P1. As long as we ignore unsuccessful reads of the flag locations on each processor, the above optimization yields sequentially consistent executions of the program.[6] Furthermore, the writes to the flag locations need not be atomic with respect to multiple copies; therefore, even in scalable systems (where it is difficult to make an update write appear atomic), it is possible to use a simple update protocol to more efficiently communicate the modification to each flag.

Figure 3.11(b) shows a program segment that is similar to the one in Figure 3.11(a), except we use locks and unlocks (implemented by test-and-set and write to unset the lock) instead of flags. As shown, the test of the test-and-set and the write to unset the lock on each processor are loop reads and writes, respectively, and the set is a non-competing operation. Thus, the acquisition of the lock can occur fully before the release of the previous lock on a given processor (i.e., if the lock being acquired is already free).

Figure 3.12 provides another example to illustrate the reordering and overlap that is enabled by PL3. The sequence of operations shown in Figure 3.12(a) is the same as those in Figure 3.5. As before, the competing

[6]In a sequentially consistent execution, it is impossible for both P1 and P2 to read the value of 0 for Flag2 and Flag1, respectively. However, this can occur with the optimization discussed above. Nevertheless, these unsuccessful reads do not need to be considered as part of the outcome for the (modified) execution, thus allowing us to consider executions with the optimization as being sequentially consistent.

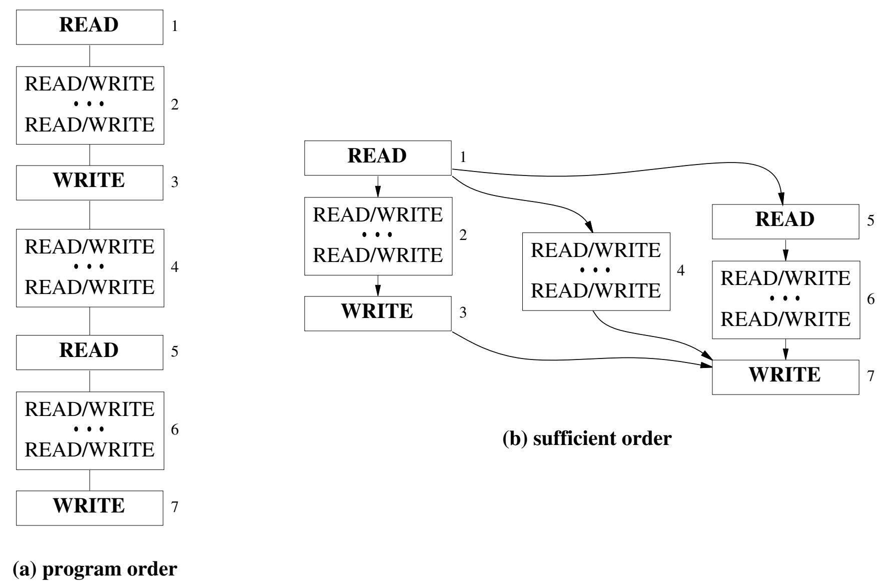


Figure 3.12: Possible reordering and overlap for PL3 programs.

operations are shown in bold. Assume all competing operations that are shown are identified with the loop label, and that blocks 1 to 3 and blocks 5 to 7 correspond to two critical sections. Compared to the overlap shown in Figure 3.5(b), the categorization of competing operations into loop and non-loop enables further overlap by allowing the write in block 3 (end of first critical section) to be reordered with respect to the read in block 5 (beginning of the second critical section). As a result, the two critical sections on the same processor can be almost fully overlapped.

3.2.4 Relationship among the Properly-Labeled Models

We have presented three programmer-centric models that successively exploit more information about memory operations to increase opportunities for overlap and reordering. The first model (PL1) requires information about operations that are competing (i.e., involved in a race). The second model (PL2) requires a further distinction among competing operation based on whether they are used to synchronize other memory operations. Finally, the third model (PL3) further distinguishes a common type of waiting synchronization construct. We expect that the majority of programmers will opt for the PL1 model due to its simplicity and the fact that the information required by PL1 enables by far the most important set of optimizations. The PL2 and PL3 models target a much smaller group of programmers. For example, the PL3 model may be used by system programmers who write the code for synchronization primitives such as locks and barriers.

Since the three properly-labeled models form a hierarchy, programs written for one model can be easily and automatically ported to another model in the group. We first consider porting a program in the direction of a more aggressive model. The categorization tree for memory operations (e.g., Figure 3.9) can be used to determine how to correctly transform operation labels in one model to labels in another model. For example, to port a program written for PL1 to PL2, any operations labeled as competing in PL1 can be trivially treated as sync in PL2. Of course, with some extra reasoning, the programmer may be able to more aggressively label some of the competing operations in PL1 as non-sync in PL2. Porting a program to a less aggressive model is similar. For example, a sync or non-sync label under PL2 must be treated as a competing label under PL1.

A subtle issue arises when porting a program from PL3 to either PL1 or PL2 because PL3 excludes unsuccessful operations (in synchronization loop constructs) from executions when deciding whether an operation is competing. As a result, some operations that are labeled as non-competing in PL3 may be considered as competing in PL1 and PL2 (e.g., the set in a test-and-set used within a lock primitive). Therefore, a simple transformation such as treating non-loop, loop, and non-sync operations in PL3 as competing operations in PL1 does not necessarily lead to a PL1 program. This means that the PL1 model does not *theoretically* guarantee that such a program will behave correctly. One possible remedy is to extend the PL1 and PL2 definitions to also exclude unsuccessful operations in a synchronization loop construct. This is *not necessary in practice*, however, since the memory ordering constraints enforced by systems that support the PL1 or PL2 models are typically a strict superset of the sufficient constraints required for supporting the PL3 model.[7]

Section 3.5 describes practical ways for programmers to convey information about memory operations to the system based on the proper labeling framework discussed in this section. As we will discuss, most application programmers deal with information at the level of the PL1 model only. Therefore, only a few system programmers or sophisticated application programmers may deal with the extra information required by the PL2 and PL3 models.

3.3 Relating Programmer-Centric and System-Centric Models

This section summarizes the memory optimizations enabled by properly-labeled models as compared to the system-centric models described in Chapter 2. The more formal sets of sufficient conditions for supporting the three properly-labeled models are presented in Chapter 4.

Table 3.1 summarizes the set of sufficient constraints for satisfying each of the three properly-labeled models described in the previous section. For each model, we show the labels used by the model and the sufficient program order and atomicity constraints that would satisfy the model. The program order constraints apply to operations to *different* locations. For simplicity, we no longer carefully distinguish an operation's label from the operation's intrinsic category. Furthermore, we use the names of categories (i.e., the non-leaf nodes in the category trees shown in Figures 3.3, 3.6, and 3.9) in addition to label names (i.e., the leaf nodes). For example, the competing category covers the sync and non-sync labels in PL2 and the non-loop, loop, and non-sync labels in PL3. We want to emphasize that the constraints we describe are only sufficient constraints; they are not necessary constraints for either supporting the given PL model or for guaranteeing sequentially consistent results for a given program.

Consider the PL1 model. Table 3.1 shows that it is sufficient to maintain program order between a non-competing operation followed by a competing write, a competing read followed by a non-competing operation, and two competing operations. Similarly, multiple-copy atomicity should be maintained for competing writes. Table 3.2 provides the complementary information to Table 3.1 by showing the operations for which program order and multiple-copy atomicity do not need to be maintained. As before, the program order relaxations apply to operations to different locations. For PL1, program order need not be maintained between two non-competing operations, a non-competing operation followed by a competing read, and a

[7] This property also holds for the sufficient conditions presented in Chapter 4 for supporting the above three models.

Table 3.1: Sufficient program order and atomicity conditions for the PL models.

Model	Labels	Program Order (sufficient): first op	Program Order (sufficient): second op	Multiple-Copy Atomicity (sufficient)
PL1	competing non-competing	non-competing	competing write	competing write
		competing read	non-competing	
		competing	competing	
PL2	sync non-sync non-competing	non-competing	sync write	competing write
		sync read	non-competing	
		competing	competing	
PL3	non-loop loop non-sync non-competing	non-competing	sync write	non-loop write non-sync write
		sync read	non-competing	
		competing read	competing read	
		competing read	competing write	
		competing write	competing write	
		non-loop or non-sync write	non-loop or non-sync read	

competing write followed by a non-competing operation. Similarly, multiple-copy atomicity need not be maintained for non-competing writes.

As shown in Tables 3.1 and 3.2, each PL model successively relaxes the program order and atomicity constraints of the previous level. Consider how program order is relaxed. The distinction between competing and non-competing operations in PL1 enables the most important class of optimizations by allowing for non-competing operations to be overlapped with respect to one another. Since non-competing operations constitute the large majority of operations in most program, relaxing the ordering constraints among them can provide substantial performance gains. The further distinction of competing operations into sync and non-sync in PL2 can improve performance by relaxing the program order between non-competing operations and competing operations that are categorized as non-sync. Finally, the distinction of sync operations into loop and non-loop in PL3 allows the system to relax the program order between a competing write and a competing read if either one is a loop operation. Relative to PL1, the extra optimizations enabled by PL2 and PL3 are important only if the program has a frequent occurrence of competing operations.

The above discussion shows that the information conveyed through labels can be used to exploit the same type of optimizations that are enabled by aggressive system-centric models. Below, we provide some intuition for how this information may be used to efficiently execute PL programs on system-centric models while still maintaining sequential consistency. The next chapter provides a more formal set of conditions for porting PL programs to system-centric models.

We begin by considering the first set of system-centric models introduced in the previous chapter (i.e., IBM-370, TSO, PC) that allow reordering of a write followed by a read in program order. Given the information conveyed by a PL1 program, this reordering is safe if either the write or the read is non-competing. The additional information provided by PL2 does not provide any additional cases. Finally, with PL3 information, the reordering is also safe between a competing write and a competing read as long as at least one is a loop operation. The second set of system-centric models that includes PSO can further exploit the information conveyed by labels by allowing the reordering of two writes as well. For example, the PL1 information makes the reordering of two writes safe as long as the second write is non-competing.

Table 3.2: Unnecessary program order and atomicity conditions for the PL models.

Model	*Labels*	*Program Order (unnecessary)* *first op*	*second op*	*Multiple-Copy Atomicity (unnecessary)*
PL1	competing non-competing	non-competing	non-competing	non-competing write
		non-competing	competing read	
		competing write	non-competing	
PL2	sync non-sync non-competing	non-competing	non-competing	non-competing write
		non-competing	sync read	
		sync write	non-competing	
		non-competing	non-sync	
		non-sync	non-competing	
PL3	non-loop loop non-sync non-competing	non-competing	non-competing	non-competing write loop write
		non-competing	sync read	
		sync write	non-competing	
		non-competing	non-sync	
		non-sync	non-competing	
		competing write	loop read	
		loop write	competing read	

The third and last category of system-centric models consists of models such as WO, RCsc, RCpc, Alpha, RMO, and PowerPC, that allow program reordering among all types of operations. Given WO, we can exploit the information conveyed by PL1 to allow non-competing operations to be reordered with respect to one another. The extent to which each model exploits the label information varies, however. For example, while WO cannot exploit the distinction between competing read and competing write operations, the other models (i.e., RCsc, RCpc, Alpha, RMO, and PowerPC) can use this distinction to safely relax the program order between certain competing and non-competing operations (e.g., between a competing write followed by a non-competing read).

Among the system-centric models, the RCsc model best exploits the label information provided by PL1 and PL2 programs, and the RCpc model best exploits the information provided by PL3 programs. This is partly because the definition of PL programs was closely linked to the design of release consistency as the implementation conditions for a system [GLL+90]. The next chapter provides an even more aggressive set of conditions, as compared with RCsc and RCpc, that still lead to sequentially consistent executions of PL programs. While the release consistency conditions are sufficiently aggressive for most practical hardware designs, the more aggressive set of conditions provide opportunity for higher performance in designs that support shared-memory in software (see Chapter 5).

3.4 Benefits of Using Properly-Labeled Models

The benefits of using properly-labeled models mainly arise from programming simplicity and easy and efficient portability among different systems. We briefly summarize some of these advantages below.

The primary advantage of properly-labeled models is that they are simpler to program with and reason with than the system-centric models. While system-centric models require the programmer to reason with low-level reordering optimization, programmer-centric models maintain the intuitive sequential consistency model as the base model and simply require extra information to be provided about the behavior of memory operations in sequentially consistent executions of the program. By far the most important piece of information is whether a memory operation is involved in a race (i.e., competing or non-competing). Since most parallel programs are written with the intent of disallowing races on shared data, the programmer already has to analyze the program for any races. Therefore, requiring this information to be made explicit leverages the fact that such information is already naturally known by the programmer. The other two types of information required by PL2 and PL3 (i.e., sync/non-sync and loop/non-loop, respectively) are also relatively intuitive, even though the formal definitions for these categories may seem complex.

An important attribute of the properly-labeled models is that they allow the programmer to provide *conservative* information about memory operations. This greatly simplifies the task of providing correct information about memory operations since if the exact information about some operation is not known, the programmer can simply provide the conservative information. For example, an operation may be safely labeled as competing if the programmer is not sure whether the operation is involved in a race. At the extreme, a program can be trivially labeled by providing conservative labels for all operations; for example, labeling all operations as competing trivially yields a PL1 program. Of course, conservative labels reduce the opportunity for doing optimizations, and in the extreme case where all labels are conservative, system performance degrades to the level of a typical sequentially consistent implementation. Therefore, from a performance perspective, it is best to limit the use of conservative labels.

Another benefit of allowing conservative labels is that programmers can focus on providing accurate information for the performance critical regions in a program. For example, if most of the program's execution time is spent in a handful of procedures, the programmer can concentrate on analyzing the memory operations in those procedures, and provide more conservative labels for the remaining memory operations. Similarly, the more detailed information required by PL2 or PL3 may be provided for critical synchronization algorithms used in a program, while the remaining parts may provide the less detailed information required by the PL1 program. Overall, the ability to provide conservative labels allows programmers to observe performance benefits that are proportional to the amount of effort spent by the programmer to properly label a program.

The information provided by properly-labeled programs also allows automatic and efficient portability of such programs across a wide range of systems. Contrast this to system-centric models. For example, consider porting a program that is originally written for the Alpha model to the RMO model. Recall that the Alpha model inherently maintains program order among read operations to the same location while the RMO model does not maintain this order. This subtle difference makes it difficult to efficiently port a program from Alpha to RMO because we must conservatively assume that the program orders between all pairs of reads to the same location are pertinent for correctness. In contrast, the labels in a PL program convey a lot more information about the operations, allowing us to efficiently match the orders required by the program to the orders provided by a given system.

Finally, a side benefit of operation labels is that they can serve as a form of documentation for the program by conveying information such as the set of memory operations that are involved in races or the set of operations that are used for synchronization. This type of information simplifies the task of understanding a program and can therefore enhance the maintainability of parallel programs.

Overall, the PL models provide programmers with a uniform programming strategy across a wide range of systems without sacrificing ease of use, efficiency, and portability.

3.5 How to Obtain Information about Memory Operations

The previous sections introduced the notion of operation labels as an abstraction for conveying information about memory operations. This section describes practical ways of obtaining such information. The programming interface between the programmer and the system plays a key role in determining who is responsible for conveying the labels and how the labels are conveyed. We begin by describing a number of possible interfaces between the programmer and the system, and discuss the responsibilities of the programmer at each interface. We next describe mechanisms that allow programmers to convey any required information at the language level. The actual mechanisms for conveying such information to the underlying hardware will be discussed in Chapter 5.

3.5.1 Who Provides the Information

The degree to which the programmer is exposed to the underlying relaxed memory model in a system is solely determined by the type of programming language interface. We will consider three broad classes of languages below: sequential languages with compiler-generated parallelism, explicitly parallel languages with implicit synchronization to ensure lack of races, and explicitly parallel languages with explicit synchronization under programmer control. For the first two classes of languages, the underlying memory model is transparent to the programmer. For the third class of languages, however, the programmer observes the relaxed memory model and may also be required to provide explicit information about memory operations to ensure correct program behavior.

Even though a language may not expose the underlying relaxed model to the application programmer, it is still beneficial to exploit relaxed models for achieving higher performance implementations of the language. This requires exposing the compiler and system programmers who implement the language and associated software libraries to the underlying relaxed model. Whether the operation labels are provided by the application programmer or the language library and compiler programmers, the proper labeling approach can be used to simplify the task of the programmer who is exposed to the underlying relaxed model.

Sequential Languages

The first category of languages we consider is traditional sequential languages. To exploit parallelism in such languages, the compiler is responsible for automatically parallelizing the program and coordinating the communication and synchronization among the parallel threads that it generates. Since the programmer's logical view of the system is that of a uniprocessor, all multiprocessor aspects including the multiprocessor memory model are transparent to the programmer.

To exploit the underlying relaxed memory model in a system, the compiler is responsible for properly labeling the memory operations in the parallel program that it generates. Since the compiler generates sufficient synchronization to ensure accesses to the original program's data structures are race-free, memory operations to these data structures will inherently be non-competing. The remaining operations in the parallel program are generated by the compiler to coordinate and synchronize the parallel tasks. The compiler must convey appropriate labels for these latter operations as well. For example, consider the lock and barrier synchronization primitives in Figure 3.10 with the corresponding labels. The effort in aggressively labeling such synchronization code is well worth it since the code is reused across multiple applications.

In summary, sequential languages can exploit relaxed memory models without increasing programming complexity for the application programmer.

Explicitly Parallel Languages with Implicit Synchronization

The languages in the second category are similar to sequential languages in that the programmer is allowed to reason with the traditional uniprocessor memory model. The only difference from a sequential language is that the programmer is required to provide information that allow the compiler to more effectively extract the parallelism in the program. With the parallelism exposed, the compiler is responsible for generating the parallel threads and imposing sufficient synchronization to ensure accesses to the original program's data structures are race-free (i.e., non-competing). Therefore, similar to sequential languages, the compiler is responsible for properly labeling the output program that it generates in order to exploit an underlying relaxed memory model.

Jade [RSL93] and FX-87 [JG88] are examples of languages designed to express parallel execution while preserving the serial semantics of the corresponding sequential program. In Jade, for example, the programmer identifies tasks that are useful to execute concurrently and specifies the data usage information for each task. This data usage information allows the compiler and run time system to execute tasks concurrently while preserving the data dependences between tasks; the system ensures this serial semantics by automatically inserting appropriate synchronization among the tasks. High Performance Fortran (HPF) [Lov93] is another example of a language in this category. In HPF, the programmer writes a serial program and annotates it with data distribution specifications. HPF also provides primitives to identify concurrent computation, such as an intrinsic data-parallel array operation or an explicit *forall* statement that identifies a loop as fully parallel.[8]

Explicitly Parallel Languages with Explicit Synchronization

The third category of languages we consider require the programmer to explicitly identify the parallelism and to coordinate the parallel tasks through explicit synchronization. Therefore, unlike the previous categories, the programmer is exposed to the multiprocessor semantics. The various options in this category trade off the level of control versus ease-of-use provided to the programmer.

The first option is to provide a set of predefined synchronization constructs and to require the programmer to use these constructs appropriately to ensure all memory operations are race-free or non-competing. These predefined synchronization constructs may either be part of the language or provided through parallel runtime libraries. For example, monitor-based languages such as Concurrent Pascal [Han77], Mesa [LR80], and Modula [Wir77] provide language-based synchronization constructs, while environments such as C-threads [CD88], Presto [BLL88], and the ANL [BBD$^+$87]/Parmacs [Bec90] macros extend conventional sequential languages through parallel runtime libraries.

As we mentioned above, the application programmer is responsible for ensuring that all memory operations are non-competing by using sufficient synchronization. To achieve this, the programmer needs a high-level

[8]These constructs have parallel copy-in/copy-out semantics.

semantic description of the synchronization constructs and a definition of non-competing based on these high-level constructs. Providing this type of description is relatively straightforward. For example, consider lock and unlock operations as part of the synchronization constructs. To define the notion of non-competing with lock and unlock constructs, one can simply adapt Definition 3.1 to describe a chain of unlock-lock pairs ordering conflicting operations. Given that the programmer is responsible for guaranteeing that all memory instructions that access program data structures are race-free, the compiler can automatically label such instructions as non-competing. Furthermore, as in the case of sequential languages, the memory instructions that comprise the predefined synchronization constructs are assumed to be labeled by the programmers who implement them and such labels can be passed along in the compiled code. Therefore, even though the application programmer is exposed to the multiprocessor semantics, no explicit operation labels are provided by the programmer.

The option described above may be limiting for some programmers since it disallows the use of synchronization primitives that are customized by the programmer and disallows the use of competing operations (e.g., refer back to the branch-and-bound algorithm of Figure 3.7 with the unsynchronized access to the bound variable). We can provide the programmer with more control at the cost of requiring the programmer to supply explicit labels for the memory operations in the program.

In summary, for languages with explicit synchronization, the programmer is responsible for providing extra information that helps the compiler in generating operation labels, ranging from guaranteeing that all memory operations are race-free by using predefined synchronization primitives to explicitly providing labels for all operations. Section 3.8.4 describes some programming environment tools and architectural features that can help the programmer provide correct information in the form of operation labels, while Section 3.6 further discusses the trade-offs in applying the proper labeling approach to programs with unsynchronized accesses.

3.5.2 Mechanisms for Conveying Operation Labels

As we discussed in the previous part, memory operation labels are provided either by language library and compiler programmers or by application programmers, depending on the language interface that is chosen. In this section, we describe language-level mechanisms that can be used by the programmer to convey the required information. The actual mechanisms for further conveying such information to the underlying hardware will be discussed in Chapter 5.

Operation labels as described in Section 3.2 are attributes for memory operations that appear in the dynamic memory reference stream corresponding to the parallel execution of a program. However, practical techniques for providing this information at the language level need to work within the context of the static program text. We describe two reasonable approaches for expressing operation labels. The first approach is to express the category information as an attribute of shared variables or memory locations. This can be achieved by either providing the information as part of the declaration for each program variable or specifying the information based on address ranges for memory locations; every operation in the dynamic execution that accesses the given variable or memory location inherits this information as a label. We refer to this as the *variable* or *address-based* approach. The second approach is to provide the appropriate information for each static memory instruction that may access a shared-memory location.[9] Analogous to the address-based

[9]Instructions that specify more than a single memory operation, such as read-modify-write operations, may conceptually require a

approach, the execution of a memory instruction leads to a dynamic instance of a memory operation that inherits this information as a label. We refer to this as the *instruction-based* approach.

Providing labels in the above two approaches may be done on a per address or per instruction basis, or it may be based on regions of addresses or instructions. The mechanisms through which this is achieved are naturally language dependent. For example, the address-based approach may be supported by associating the appropriate label to operations that access specific pages. Similarly, the instruction-based approach may be supported by providing multiple flavors of memory instructions, e.g., either through extra opcodes or through extra bits in the address (i.e., address shadowing) to achieve a similar functionality to extra opcodes. Or else, regions of instructions may be associated with a given label through special delimiter instructions.

In what follows, we discuss the relative merits of the address-based and instruction-based approaches for labeling operations.

Expressive Power

The address-based and instruction-based approaches are inherently different in their level of expressiveness. With the address-based technique, dynamic operations to different memory locations can inherit different operation labels. However, accesses to the same memory location inherit the same label. On the other hand, the instruction-based approach allows different memory instructions that possibly access the same memory location to inherit different labels, but invocations of the same memory instruction always inherit the same label even if the accesses are to different memory locations.

Both methods are theoretically less expressive than the conceptual approach of individually labeling every dynamic memory operation in a given execution, which would allow using a different label for different invocations of the same memory instruction or different accesses to the same memory location. For example, consider the execution of a single instruction that leads to two dynamic accesses to memory location A. Assume that individually labeling the operations in an execution would lead to the first operation being non-competing and the second operation being competing. To ensure correct labels, we would have to declare location A as competing or the instruction as competing depending on whether we use the address-based or instruction-based approach, respectively. In either case, the resulting execution of the program would lead to both memory operations being labeled as competing, which is more conservative than conceptually labeling the operations in an execution.

In practice, the instruction-based approach may be more appropriate than the address-based approach since it is often the case that operations generated by the same instruction require the same label, while different operations to the same location may require different labels.[10]

Default Labels

Default operations labels can be used to simplify the task of labeling a program or to reduce the chance of incorrect labels resulting from programmer error.[11] These two goals may often be conflicting. To achieve the latter goal, the conservative labels would be chosen as the default. For the PL1 model, for example, we

label per operation that they generate.

[10] For example, consider the test and set operations in Figure 3.7 (or Figure 3.10(a)) which access the same location but require different labels.

[11] Ideally, the mechanism that is chosen should allow the programmer to override a default label. Otherwise, it will not be possible in general to make a program properly labeled by simply providing correct labels.

would assume every operation is competing by default. Therefore, reordering optimizations would only be allowed if the programmer explicitly identifies an operation as non-competing. This can reduce the chances of incorrect behavior that may be caused if we implicitly assume an operation is non-competing.

To simplify the task of providing operation labels, an alternative approach would be to choose the more frequent labels as the default. Given PL1 programs, for example, the non-competing label is empirically more frequent than the competing label in most programs. Therefore, by assuming the non-competing label by default, we reduce the number of instructions or variables that must be explicitly labeled (i.e., as competing) by the programmer.[12] A related technique that is relevant to the instruction-based labeling scheme is to choose the default on a per instruction basis depending on the label that is most frequently used for the given instruction type. Assume the PL1 model again. Special instructions such as test-and-set or compare-and-swap are frequently associated with competing operations. Therefore, such instructions can be labeled as competing by default. On the other hand, ordinary load and store instructions would be labeled as non-competing since it is the more frequent label for such instructions. Again, these default choices can reduce the number of explicit labels that must be provided by the programmer.

Another interesting approach is to allow the programmer to choose potentially different default labels for different regions of code. For example, consider a scenario where the programmer focuses on providing accurate labels (i.e., in contrast to conservative labels) for only the performance critical regions in a program. In this case, it may be beneficial to choose the frequent labels as the default in the performance critical regions and choose the conservative labels as the default for the remaining sections of code. The above idea may not work well with the address-based approach for conveying labels since the same data structure may be used in both critical and non-critical regions of code.

In summary, providing flexible mechanisms for specifying default labels can be extremely effective in simplifying the task of properly labeling a program.

Implementation Issues

The information conveyed in the form of labels is used by the underlying system to enable various reordering optimizations. The instruction-based labeling approach has an advantage over the address-based approach in this respect since it provides the required information to the underlying system in a more direct and usable form.

Consider compiling a parallel program with explicit synchronizations, for example. To determine whether a given set of memory instructions may be safely reordered, the compiler must ultimately deduce the labels on individual instructions based on the static information provided by the program text. This information is directly provided by the instruction-based approach. On the other hand, the address-based approach provides labels as an attribute of a given memory address. Since the memory address accessed by an instruction may be dynamic (e.g., due to indirect addressing), it can be difficult to statically determine the appropriate label for a given instruction based on information provided by the address-based approach. This is especially true in languages such as C that are not strongly-typed. Therefore, the compiler may end up with conservative information in trying to transform labels on memory addresses to labels on memory instructions.

Unlike the compiler, the hardware has access to the dynamic execution of the program, allowing it

[12] For the other PL models, the sync label is empirically more frequent than the non-sync label and the loop label is more frequent than the non-loop label in most programs.

to accurately transform memory address labels to instruction labels. Nevertheless, the instruction-based approach still provides an advantage over the address-based approach simply because it directly notifies the hardware of the instruction label even before the instruction's effective address is computed. Assume an implementation of the PL1 model, for example. Consider a read instruction with a yet unresolved memory address that depends on a long series of previous computations. With the instruction-based approach, the hardware knows whether the read is competing or non-competing as soon as the instruction is fetched. In the case of a non-competing read, an aggressive hardware can safely service operations past the read potentially before the read address is computed. With the address-based approach, however, the hardware must conservatively assume that the read is competing until its effective address is resolved, thus losing the opportunity to overlap the time to compute the read's address with the servicing of operations that follow the read.

The above issues regarding implementation efficiency are by far the most important in comparing the address-based and instruction-based approaches, making the latter approach more desirable.

3.6 Programs with Unsynchronized Memory Operations

The majority of parallel programs are written with sufficient synchronization to ensure memory operations to data structures are race-free (i.e., non-competing). However, programmers sometimes avoid synchronization operations to achieve higher efficiency and performance. We loosely refer to such programs as *unsynchronized programs* and refer to memory operations that are not protected by synchronization as *unsynchronized operations* (similar to competing operations).

This section discusses the appropriateness of properly-labeled models for reasoning about unsynchronized programs. The section begins by further describing the incentive for writing unsynchronized programs. We next consider the trade-off of properly labeling unsynchronized programs, as compared to using system-centric models, from both a performance and an ease-of-use perspective.

3.6.1 Why Programmers Use Unsynchronized Operations

For most programs, eliminating races on program data structures is necessary for correctness even when we use a conservative memory model such as sequential consistency (e.g., refer back to Section 2.1.4). Furthermore, the presence of data races complicates the task of reasoning about program correctness since races often lead to a larger set of program behaviors that must be analyzed. Therefore, most programs use synchronizations to avoid races and the lack of sufficient synchronization to protect data structure is often considered a programming mistake.

There are some algorithms that can tolerate data races, however, and removing synchronization in such algorithms can potentially lead to higher performance and efficiency.[13] The main performance gains from removing synchronization arise from eliminating the serialization among tasks (e.g., one task waiting for another task to release a lock or reach a barrier) and eliminating the cost of synchronization (e.g., acquiring a free lock). In addition, since fine grain synchronization can sometimes incur a large memory overhead,

[13]The option of removing synchronization is only available if the programmer uses an explicitly parallel language with explicit synchronization, since a parallel program generated from a sequential language or a language with implicit synchronization will inherently have sufficient synchronization (refer back to the discussion Section 3.5.1).

eliminating such synchronization can lead to a more efficient use of the available memory space.

A common class of algorithms that can tolerate unsynchronized operations are iterative algorithms based on chaotic relaxation. An example is the parallel successive-over-relaxation (SOR) method for solving differential equations which leads to a nearest-neighbor computation among processors and is relaxed to tolerate old values from neighboring processors within each phase of the computation. Eliminating the synchronization in this algorithm is essential for exploiting parallelism even though it changes the semantics compared to the sequential SOR method. Since the algorithm iterates to convergence, old values are tolerated robustly and do not affect the correctness of the result. Many other iterative techniques based on chaotic relaxation work on a similar principle and can robustly tolerate unsynchronized memory operations.

In addition to the iterative algorithms described above, there are other types of algorithms that function correctly with unsynchronized memory operations. Refer back to the branch-and-bound algorithm shown in Figure 3.7, for example. As we mentioned in Section 3.2.2, each processor computes a new bound and compares it to the global bound to see whether the new bound is the lowest bound computed up to that point. The typical behavior is that the global bound is read frequently but is seldom written to. Therefore, acquiring a synchronization for every read can needlessly serialize the computation when several processors attempt to read the bound. This serialization can be alleviated if the global bound is read in an unsynchronized manner. The lock protecting the global bound is acquired only if a processor's computed bound is smaller than the global bound. This algorithm is sufficiently robust to tolerate the unsynchronized read of the global bound; the global bound monotonically decreases during the course of the computation and reading an old value simply implies that some processors will do extra work but does not affect the overall correctness of the results.

Finally, efficient high-level synchronization primitives such as the barrier synchronization shown in Figure 3.10(b) often require the use of unsynchronized operations. However, properly labeling such synchronization algorithms typically leads to efficient primitives, as is the case with the barrier example.

In summary, reasoning about the correctness of programs with unsynchronized accesses is a non-trivial task regardless of whether we assume a strict model such as sequential consistency or a relaxed memory model. Nevertheless, eliminating the synchronization can potentially provide a large enough performance advantage to warrant the use of unsynchronized memory operations in some programs. The next section describes the appropriateness of the properly-labeled models for reasoning about unsynchronized programs.

3.6.2 Trade-offs in Properly Labeling Programs with Unsynchronized Operations

Compared to the system-centric models, the properly-labeled (PL) models offer an advantage in ease of use, and for the majority of programs, provide a comparable or higher performance potential. The main performance gain from PL models arises from the ability to reorder and overlap memory operations that are non-competing. Therefore, PL models are especially well suited for programs that exhibit a high frequency of non-competing operations due to use of sufficient synchronization. Conversely, PL models are not well suited for programs that exhibit a high frequency of unsynchronized (i.e., competing) operations; even though the PL models still provide an advantage in ease of programming compared to system-centric models, using system-centric models may lead to a higher performance potential for such programs.

There are two options for programs with a high frequency of competing operations: (i) use the PL models at a possible loss in performance, or (ii) use system-centric models to possibly achieve higher performance at the cost of higher programming complexity. Before considering these two options in more detail, we first discuss some possible misconceptions regarding the frequency of competing operations in typical programs.

Possible Misconceptions about Unsynchronized Operations

Regardless of whether the programmer uses sequential consistency or a more relaxed memory model, most programs typically require sufficient synchronization in order to function correctly. The majority of programs have a relatively low frequency of competing memory operations, with races typically confined to only a few operations. The frequency of competing or unsynchronized operations may sometimes be over-estimated simply because of a misconception about what a competing operation is. Below, we present a few illustrative examples to clarify some of the common misconceptions about competing operations.

One potential source of confusion is that non-determinism in the order in which operations are executed may be mistaken as an indication of competing operations. Consider the example in Figure 3.4(b). The order in which the writes on P1 occur with respect to the reads on P2 to locations A and B is non-deterministic and depends on the order in which the two processors enter the critical section. However, this non-determinism does not constitute a race on locations A and B. In every execution, conflicting operations to these locations are ordered through an ordering chain consisting of a write to unset the lock on one processor and a test-and-set on the other processor. Thus, operations to locations A and B are actually non-competing.

Another source of confusion may arise from the use of ordinary reads and writes to achieve synchronization. Consider the program segment from Figure 3.1(a). Event synchronization is achieved with one processor writing to the Flag location while the other waits for the new value. Since this example does not use a conventional critical section (e.g., locks), some programmers may consider operation to locations A and B as competing. However, these operations are in fact non-competing since they are ordered in every execution by the competing operations to Flag.

Finally, the example in Figure 3.13 shows a more subtle interaction between two processors with no explicit synchronization. Assume Head_Ptr, Rec0_Ptr, and Rec1_Ptr are record pointers. The program segment shows P1 inserting a new record at the head of a linked list while P2 asynchronously reads the record at the head. The record at the head is initially the one pointed to by Rec0_Ptr, with P1 later inserting a different record pointed to by Rec1_Ptr at the head. The read of the head record by P2 can occur either before or after the insertion of the new record by P1. This type of optimized code may for example be used to add and look up records in a list with no deletions. The lack of any form of explicit synchronization in this example may lead to the conclusion that all the memory operations are competing. However, the only competing operations are the write on P1 and the read on P2 to the Head_Ptr. Consider the two possible sequentially consistent outcomes. In the first outcome, P2 reads the original record before P1 inserts the new record. Therefore, except for the write and read operations to the Head_Ptr, the remaining operations on P1 and P2 are to fields of two different records and cannot compete. In the second outcome, P2 reads the newly inserted record. The operations on P1 and P2 are now to fields of the same record. However, these operations remain non-competing because they are ordered through the ordering chain comprising of the write on P1 and the read on P2 to the Head_Ptr. Therefore, even though there is no explicit synchronization between the two processors, the operations to the record fields are in fact non-competing.

In summary, intuitive inspection of a program may often lead to an over-estimation of the actual frequency of competing operations that is revealed by a more careful analysis.

Initially Head_Ptr = Rec0_Ptr

P1		P2	
a1:	Rec1_Ptr–>field1 = 1;	*a2:*	ptr = **Head_Ptr;**
b1:	Rec1_Ptr–>field2 = 2;	*b2:*	r1 = ptr–>field1;
c1:	Rec1_Ptr–>next = Head_Ptr;	*c2:*	r2 = ptr–>field2;
d1:	**Head_Ptr** = Rec1_Ptr;	*d2:*	r3 = ptr–>next;

Figure 3.13: Example with no explicit synchronization.

Properly Labeling Unsynchronized Operations

Compared to a program written for relaxed system-centric models, the performance using the PL approach is often comparable or better when competing operations are infrequent. On the other hand, programs written for system-centric models may potentially perform better if competing operations are frequent. Of course, the system-centric approach may not provide an advantage if many of the conservative orders maintained (by the PL approach) among competing operations are actually necessary for the correctness of a given program and must therefore also be maintained by the system-centric model.[14]

We want to emphasize that the PL models may be used to reason about any program, regardless of whether the program has sufficient synchronization. The only requirement for properly labeling a program is to provide correct labels for all memory operations. For example, given the PL1 model, the only requirement is for every memory operation that is competing (i.e., unsynchronized) to be identified as competing$_L$. Therefore, properly labeling an unsynchronized program is conceptually no more difficult than properly labeling a program with sufficient synchronization. It is important to note that properly labeling an unsynchronized program *does not* require adding any extra synchronization to the program. In fact, it is likely that the programmer made a conscious decision to eliminate synchronizations to achieve higher performance; reinserting synchronization to eliminate the unsynchronized operations could therefore be the wrong choice (e.g., consider the chaotic relaxation or branch-and-bound algorithm described in Section 3.6.1).

Using System-Centric Models to Reason about Unsynchronized Operations

Algorithms that are consciously designed with insufficient synchronization are robust enough to tolerate the memory behavior caused by unsynchronized operations. Typically, the same robustness characteristics make many such algorithms immune to the reordering of memory operations that results from relaxing the memory model. As a result, many unsynchronized algorithms do not depend on sequential consistency for correctness. For example, the results of a chaotic over-relaxation algorithm may be considered correct if they satisfy specific convergence criteria, even though some of these results would not be possible on a sequentially consistent system. The above characteristics make unsynchronized algorithms suitable for systems with relaxed memory models since the system can achieve reordering and overlap optimizations that would not be possible if the results were required to be sequentially consistent.

For algorithms that do not depend on sequential consistency for correctness, there may be a performance

[14]For iterative algorithms, it is also important to consider the effect of relaxing the memory model on the number of iterations that are required for convergence.

advantage in using the system-centric approach instead of the properly-labeled approach because the latter can limit performance by guaranteeing sequentially consistent executions.[15] Of course, the correctness of unsynchronized programs still depends on maintaining certain ordering constraints. For example, the fact that an unsynchronized program functions correctly on a sequentially consistent system does not automatically imply that it will function correctly with any relaxed memory model. Therefore, the programmer must still ensure that the low-level reordering optimizations enabled by a given relaxed model do not affect the correctness of the program. The above can be a non-trivial task from an ease of programming perspective.

3.6.3 Summary for Programs with Unsynchronized Operations

Most programs use sufficient synchronization since this simplifies reasoning about the program. The proper labeling approach is well suited for such programs since it provides higher ease of programming and comparable or higher performance than the system-centric approach. However, there are a small class of programs that eliminate some synchronization operations in order to achieve higher performance. The system-centric approach may exhibit a performance advantage for such programs over the PL approach. Nevertheless, this performance advantage is accompanied with a large increase in programming complexity since the programmer has to reason about the correctness of the program given both the lack of synchronization in the algorithm and the low-level reordering that is exposed by a system-centric model.

3.7 Possible Extensions to Properly-Labeled Models

This section describes a number of possible extensions to the programmer-centric models we have presented in this chapter.

3.7.1 Requiring Alternate Information from the Programmer

Section 3.2 presented the three properly-labeled models along with the types of information about memory operations that is exploited by each model to achieve higher performance. Although we only described three specific types of information, our general framework may be extended to allow other types of information to be expressed by the programmer.

In assessing the utility of a new type of information, it is important to consider the trade-off between the extra performance that results from exploiting this information versus the extra complexity for the programmer to provide the information. As discussed in Section 3.3, among the three types of information used by the PL models, the first type of information that distinguishes between competing and non-competing operations allows the system to achieve most of the performance gain possible from exploiting relaxed memory models; in most cases, the further distinction of competing operations into sync/non-sync and loop/non-loop does not lead to substantial gains in performance. Therefore, it is important to consider the diminishing returns in performance when proposing new types of information and to balance this with the extra burden imposed on the programmer for providing such information.

[15] The programmer has the option of not properly labeling a program in order to allow non-SC outcomes. Even though the memory behavior for non-PL programs is theoretically unspecified under a PL model, any practical implementation will obey a set of sufficient conditions (similar to those specified by a system-centric model) that inherently specifies the memory behavior for all programs. These conditions can be used in a similar way to a system-centric model to reason about the behavior of a non-PL program.

The explicit relationship between data and synchronization operations in a program is an example of a type of information that is not captured by the labels we have presented thus far. Inherently, a synchronization operation protects a specific set of memory operations from competing with conflicting operations on other processors. For example, referring back to the program segment in Figure 3.1(a), the write to Flag on P1 and the read of Flag on P2 act as a synchronization to protect operations to locations A and B from competing. It is relatively straightforward to extend the programmer-centric models presented in this chapter to also require the relationship between data and synchronization to be expressed as part of the labeling process. This extra information can be exploited by the system to order synchronization operations only with respect to the memory operations that they protect as opposed to conservatively ordering them with respect to unrelated operations.

As discussed in Section 2.5, identifying the environment and the types of architectures that a model is intended for is an important factor in assessing the performance gains from exploiting a specific information. For example, consider the extra information discussed above for relating synchronization to the data they protect. Exploiting this information in a typical shared-memory hardware architecture can be quite difficult and may not yield enough of a performance gain to make it worthwhile. However, in the domain of software distributed-shared-memory (DSM) systems where more of the coherence protocol is managed in software, exploiting this type of information can enable important optimizations that would not be possible otherwise. In fact, Midway [BZ91] is an example software DSM system that successfully exploits this type of information (see Chapter 5).

3.7.2 Choosing a Different Base Model

The PL framework presented in this chapter uses sequential consistency as its base model. Therefore, programmers can simply reason with SC and are guaranteed that the result of their programs satisfy SC as long as they provide the correct information about the memory operations. The reason for choosing SC as the base model is not so much based on the fact that most programmers depend on this model, but because SC maintains sufficient orders among memory operations to eliminate any "unexpected" behavior that may arise due to overlap and reordering of memory operations. This relieves the programmer from considering the effect of low-level reordering optimizations on the correctness of a program.

However, the model that is implicitly assumed by most programmers is likely to be less strict than sequential consistency. For example, many programmers may simply expect some notion of causality to be upheld. Therefore, a model such as TSO which maintains causality may be sufficiently strict for such programmers. This suggests that extending the basic programmer-centric framework to explore the use of non-SC models as the base model may be a plausible option. Consider the effect of using an alternative model such as TSO as the base model. First, programmers would now reason with TSO instead of SC, and the system would guarantee that executions of the program would satisfy TSO. Second, some of the information required from the programmer to achieve certain optimizations with SC as the base model may no longer be required if the optimizations are already allowed by the alternative base model. For example, since the TSO model already allows reads to execute out of program order with respect to preceding writes, we no longer need the loop/non-loop information (as required by the PL3 model) to exploit this reordering between competing memory operations (i.e., competing write followed in program order by a competing read). Of course, the programmer is now exposed to the write-to-read reordering through TSO.

The programmer-centric framework can alternatively be thought of as a way of providing an equivalence between two different memory models; the programmer is presented with the more conservative model while the system achieves the performance associated with the more aggressive model. As an example, we can choose two system-centric models such as TSO and RMO. The programmer may be allowed to reason with TSO and is required to provide certain information about the behavior of memory operations in TSO executions of the program. The system can in turn use this information to exploit the optimizations possible under RMO while maintaining correctness according to TSO. The type of information required from the programmer will obviously be a function of the two models chosen.

3.7.3 Other Possible Extensions

The framework presented in this chapter may be extended in several other areas, a few of which are described below. One possible area is to explore extensions to the programmer-centric approach to provide better performance for unsynchronized programs that exhibit a high frequency of competing operations while still maintaining programming simplicity. This will likely require a deeper understanding of the behavior of such programs. Other possible areas to explore include extending programmer-centric models to deal with multiple granularity operations (e.g., operations at byte and word granularity), writes to the instruction space, and operations to I/O devices; the system specification and implementation issues related to the above are discussed in more detail in Chapters 4 and 5.

3.8 Related Work

This section provides an overview of the related research on programmer-centric models. The two seminal programmer-centric frameworks were independently developed by our group at Stanford (proper labeling (PL) framework [GLL+90]) and Adve and Hill at Wisconsin (data-race-free (DRF) framework [AH90b]) (both papers were published in the same conference). Even though some of the ideas that led to these two frameworks already existed, the PL and DRF work were the first to formally define the exact conditions on programs (e.g., formally defining the notion of correct labels) and to precisely define the set of possible executions for programs that satisfy these conditions (e.g., guaranteeing SC executions). The hierarchy of programmer-centric models described in this chapter is an extension of our original work on PL [GLL+90] and our joint work with Adve and Hill on PLpc [GAG+92]. We begin by relating the PL hierarchy presented here to the previous work on the PL and DRF frameworks, in addition to discussing other related work on programmer-centric models. The latter part of this section describes related research on programming environments and tools that can simplify the task of using programmer-centric models.

In what follows, we mainly focus on the specification of the various programmer-centric memory models. The related work on specifying sufficient implementation rules and efficient implementation techniques for supporting programmer-centric models are covered in Chapters 4 and 5.

3.8.1 Relation to Past Work on Properly-Labeled Programs

The proper labeling (PL) framework [GLL+90] was originally introduced to simplify the task of programming for system-centric models such as release consistency (i.e., RCsc). This framework categorizes operations into competing and non-competing, and further categorizes competing operations based on whether they are used for synchronization, in addition to distinguishing between synchronization read and write operations. We later extended the PL framework to further distinguish synchronization operations into loop and non-loop operations which captures the extra optimizations enabled by a model such as RCpc. This latter framework was developed jointly with Wisconsin and led to the PLpc model [GAG+92]. Finally, Gibbons *et al.* [GMG91, GM92] have also introduced variations of the original PL framework. Below, we compare the above definitions with the hierarchy of PL programs presented in this chapter.

The original definition of a PL program [GLL+90] most closely matches the requirements for PL2 programs in the hierarchy of PL programs presented in this chapter.[16] There are a few subtle differences between the two definitions, however. For example, the definition of PL2 makes the notion of an ordering chain explicit while this notion is only implicitly present in the original description of PL programs. This leads to some minor semantic differences between the two definitions.[17] Therefore, it is conceptually possible for a program with a given set of labels to be considered a PL program according to the original definition, but not satisfy the requirements for PL2 programs. Nevertheless, the differences between the two definitions is not expected to show through for most realistic programs and labelings.

The PLpc model [GAG+92] most closely matches the PL3 model among the hierarchy of program-centric models presented in this thesis (a later technical report [AGG+93] provides some minor fixes to the definitions in the original paper [GAG+92]). Aside from differences in the definition style,[18] the major difference between the two is that the PLpc model did not distinguish between sync and non-sync operations. Making this distinction in PL3 is mainly motivated by the desire to provide a clean hierarchy among the three different models we propose (i.e., since PL2 makes the distinction).

Finally, Gibbons *et al.* have proposed two variations of the original PL framework. The first one is the PL$_{br}$ framework ("br" stands for blocking reads) [GMG91]. The PL$_{br}$ framework is more restrictive than the original PL framework because it assumes that reads are blocking (i.e., processor stalls on reads). This removes the need for distinguishing between sync and non-sync reads. Furthermore, due to the blocking read restriction, PL$_{br}$ does not allow optimizations involving the overlap of a read with operations that follow it in program order. Gibbons and Merritt later proposed the PL$_{nr}$ ("nr" stands for non-blocking reads) framework that removes the restriction of blocking reads [GM92]. PL$_{nr}$ also extends the PL$_{br}$ work by relating synchronization operations to the data operations that they protect. The notion of a release set is used to identify the memory operations preceding the release operation in program order that are protected by the release. The above idea is also used to merge the notions of sync and non-sync operations; a non-sync operation is a degenerate form of a sync operation with a null release set. There is no corresponding notion of an acquire set, however, which is somewhat asymmetric. PL$_{nr}$ also generalizes the notion of sequential consistency by allowing the program order on each processor to be a partial as opposed to a total order; the paper does not describe how this extra information can be conveyed by the programmer, however. Among the PL models proposed in this chapter, PL$_{nr}$ is closest to the PL2 model except for the release set and program

[16] The non-sync label was referred to as nsync in the original definition.

[17] One superficial difference between the original PL definition and the PL2 definition is that the former requires synchronization operations (i.e., sync) to be explicitly distinguished as either an acquire or a release. PL2 does not make this categorization explicit to the programmer simply because it is trivial for the system to automatically generate the relevant information; a read synchronization is labeled as an acquire while a write synchronization is labeled as a release. The above does not lead to a semantic difference between the PL and PL2 definitions.

[18] The original definition of PLpc refers to the same operation across different executions. Since this is difficult to formalize, we use a different way of defining PL programs here.

order extensions that are described above.

3.8.2 Comparison with the Data-Race-Free Models

The data-race-free (DRF) framework [AH90b] was introduced by Adve and Hill at the same time that we introduced the proper labeling (PL) framework [GLL$^+$90]. The original paper by Adve and Hill [AH90b] described the Data-Race-Free-0 (DRF0) model. The data-race-free-1 (DRF1) model [AH93] was later introduced as an extension.

The data-race-free-0 (DRF0) framework [AH90b] requires the programmer to distinguish *data* and *synchronization* operations. Similar to the proper labeling framework, appropriate conditions are given for determining whether a program is data-race-free.[19] Specifically, the *program order* and the *synchronization order* (order between synchronization operations to the same address) are used to define an ordering relation among memory operation. This relation, called *happens-before*, is defined as the transitive closure of program order and synchronizations order. A program is considered to be data-race-free-0 if and only if (i) for any sequentially consistent execution, all conflicting accesses are ordered by the happens-before relation, and (ii) all synchronization operation in the program are recognizable by the hardware and each accesses exactly a single memory location.

Among the hierarchy of properly-labeled models described in this chapter, the data-race-free-0 model is most similar to the PL1 model. Intuitively, the notion of synchronization and data operations in DRF0 is similar to the notion of competing and non-competing operations in PL1. The way the notion of a race is defined in DRF0 is different from the way a competing operation is defined in PL1. For example, while the interprocessor conflict orders in an ordering chain for PL1 comprise solely of write to read pairs, the DRF0 model also allows a read after a read, a read after a write, or a write after a write synchronization order to appear in the happens-before relation. Therefore, given an SC execution, the operations that are considered competing under PL1 are a superset of those considered as a race under DRF0. This difference originally led to a stricter set of constraints for supporting DRF0 [AH90b] whereby the distinction between read and write synchronizations was not fully exploited as in PL1 implementations (e.g., reordering a data write with a following synchronization read). However, Adve later observed that such optimizations are actually safe to exploit in DRF0 programs [Adv93].

Adve and Hill later extended the data-race-free framework to capture similar ideas to those introduced by the original PL framework [GLL$^+$90]. This extended model is called the data-race-free-1 (DRF1) model [AH92b, AH93]. Similar to DRF0, the DRF1 model distinguishes between *data* and *synchronization* operations. Synchronization operations are further categorized into *paired* and *unpaired* operations; pairable synchronization operations comprise of a write synchronization (release) and a read synchronization (acquire) to the same location that returns the value of the write. The happens-before is then defined as the union of program order and synchronization order among paired operations. Compared to the categorization in the original PL framework (or the PL2 model), the distinction of data and synchronization operations in DRF1 roughly corresponds to the non-competing and competing categories and the pairable and unpairable synchronizations roughly correspond to sync and non-sync operations. Furthermore, similar to PL, DRF1

[19] The name "data-race-free" is non-ideal since it may be interpreted as saying that enough synchronization primitives must be used to avoid all races, while in fact it is perfectly fine to have unsynchronized operations as long as they are identified with the synchronization label.

exploits the distinction between pairable read and write synchronizations. Overall, the achievable performance potential for DRF1 programs is comparable to that of PL (or PL2) programs.

3.8.3 Other Related Work on Programmer-Centric Models

This section provides a brief overview of other efforts aimed at enhancing the programmability of relaxed models. The common technique used in many of the approaches described below is to identify a specific class of "good" programs that provide "correct" behavior on a given system. As we will see, the notions of "good" programs and "correct" behavior are often informal and sometimes lead to ambiguous definitions.

The memory model adopted for the VAX architecture requires that accesses to shared writable data be synchronized and provides several interlocked instructions that may be used to access memory for the purpose of synchronization [DEC81]: "Accesses to explicitly shared data that may be written must be synchronized. Before accessing shared writable data, the programmer must acquire control of the data structures." Similar to the programmer-centric framework presented in this chapter, the outcome of programs that do not comply to the model's specification are unspecified.

The requirement of ensuring multiple-reader single-writer mutual exclusion among accesses to shared data has been used in several other frameworks. The weak ordering model [DSB86, DS90b, Sch89] suggests using such a programming style (based on hardware-recognizable synchronization operations) to achieve "correct" results. Unlike the VAX description however, the behavior of the model is well defined for *all* programs (see Section 2.4.5), including programs that do not obey the mutual exclusion constraint. Similar to weak ordering, numerous other hardware-centric models, including Sun TSO, PSO, and RMO models [SUN91], the Digital Alpha model [Sit92], and the IBM PowerPC model [MSSW94], recommend protecting shared data through mutual exclusion for achieving correctness and portability. In addition to promoting the use of sufficient synchronization, the description of these hardware-centric models is often accompanied by recommended implementations of common synchronization constructs (e.g., locks) for the given model [SUN91, Sit92, MSSW94].

Although the models discussed above specify particular programming styles to enhance programmability, the informal nature of the specifications often lead to ambiguities and possible correctness problems. For example, the notion of accesses to shared data being synchronized lacks a formal definition. Furthermore, it is often not clear what memory model should be assumed when determining whether accesses are synchronized. Similarly, the behavior of programs that comply with the synchronization requirement is not formally specified.

Even though these efforts suffer from ambiguities, they provided valuable intuition for the ensuing research on programming-centric approaches such as the PL framework. The PL framework presented in this chapter attempts to alleviate ambiguities associated with previous approaches by formally specifying the criteria for programs through imposing well-defined constraints on sequentially consistent executions of the program. In addition, the behavior of programs that satisfy this criteria is explicitly specified to be the same as that of a sequentially consistent system. Another unique aspect of the PL framework is that *any* program can be transformed to a PL program by simply providing the appropriate labels that convey extra information about memory instructions. In contrast, ensuring that a program's memory operations obey mutual exclusion, as is required by the techniques discussed above, may sometimes require adding extra synchronization operations and changing the overall structure of the program. Overall, the PL framework has the advantage of a formal and unambiguous specification and is applicable across a larger range of programs and programming styles.

More recently, Hagit et al. [ACFW93] have suggested a few alternative techniques for enhancing the programmability of relaxed memory models by ensuring sequentially consistent results for certain programming styles. This work is in the context of hybrid consistency model [AF92, ACFW93], a system-centric model that has similarities to weak ordering with strong memory operations roughly corresponding to synchronization operations and weak operations roughly corresponding to data operations in weak ordering. One of the techniques they suggest is virtually the same as the data-race-free-0 (DRF0) model, so we do not discuss it further. They suggest two other alternative techniques. The first alternative requires all write operations in a program to be identified as strong. Even though it is simple to automatically label all writes in a program as strong, ensuring sequential consistency for such programs unduly restricts the use of important optimizations in hybrid consistency as compared to relaxed models such as weak ordering. For example, hybrid consistency constrains the reordering of weak reads with respect to each other, while such reordering is perfectly allowable in models such as weak ordering. The second technique is a dual of the above, requiring all read operations to be identified as strong. To ensure sequentially consistent results in this case, the programmer has to also guarantee that (i) every value written to the same location is unique, and (ii) every value written is returned by some read. These latter requirements are obviously overconstraining and can greatly limit the applicability of this technique to real programs. Furthermore, analogous to the case with strong writes, ensuring sequential consistency for programs with strong reads unduly constrains the reordering of weak writes in hybrid consistency.

3.8.4 Related Work on Programming Environments

Programmer-centric models place the burden of providing correct operation labels on either the programmer or compiler. While labels can be generated automatically for certain languages, languages that provide explicit synchronization often require the programmer to provide this information (see Section 3.5.1). We briefly describe programming environment tools and architectural features that have been proposed to help the programmer with this task. Appendix D presents these techniques in greater detail.

Appendix D primarily discusses two distinct options for helping programmers use programmer-centric models. The *first* option involves an extension to the debugging environment (based on data race detection work [AP87, DS90a, HKMC90, NM89, NM91]) that helps the programmer identify incorrect labels (e.g., whether a program is properly labeled according to Definition 3.4 for PL1). While this technique is useful during the debugging phase, it fails to protect the programmer from incorrect labels during normal program runs. On the other hand, the *second* option provides programmers with a mechanism to directly determine whether a program can lead to sequentially inconsistent executions on a given system. This is achieved through a simple hardware mechanism that can efficiently monitor normal executions of the program and notify the programmer if sequential consistency may be compromised due to the presence of incorrect labels [GG91]. This latter option is motivated by the observation that programmers ultimately care about achieving correct (i.e., sequentially consistent) results and ensuring correct labels is only a means to this end. Even though the above two options are related, there is a subtle but important difference between the two. A program that provides correct labels according to a programmer-centric model is indeed guaranteed to yield sequentially consistent executions on systems that support that model. However, correct labels are not a necessity for a given program to yield sequentially consistent executions on that system. This subtle distinction leads to different techniques for detecting violations of sequential consistency as compared to detecting incorrect labels, allowing the former to be used during normal execution while the latter is confined to executions during the debugging phase.

Techniques such as those described above are critical for enhancing the usability of programmer-centric models. However, work in this area is still in its infancy. In a more general context, there are various opportunities in providing a more supportive programming and debugging environment for relaxed models by exploring enhancements to the language, compiler, runtime environment, and even the architecture.

3.9 Summary

Programmer-centric models provide an alternative approach to system-centric models for specifying the memory behavior of a system to the programmer. The primary goal of programmer-centric models is to achieve high performance while maintaining the programming ease associated with a simple and intuitive model such as sequential consistency. To achieve this, the programmer is required to provide program-level information about the behavior of shared memory operations in sequentially consistent executions of the program. This information is in turn used to enable safe ordering optimization; as long as the information provided by the programmer is correct, the system guarantees sequentially consistent results.

We defined a set of three programmer-centric models, referred to as properly-labeled (or PL) models. The three PL models form a hierarchy, with each model exploiting an additional piece of information to achieve higher performance. We demonstrated that with only a few types of information supplied by the programmer, one can exploit the full range of optimizations captured by the aggressive system-centric models and yet maintain sequential consistency. Providing the type of information that is required by these programmer-centric models is easier and more natural for the programmer than reasoning with system-centric models directly and allows for simple and efficient portability of programs across a wide range of systems. Therefore, the above approach unifies the memory ordering optimizations captured by many of the system-centric models without sacrificing ease of use and portability.

The next chapter presents the sufficient requirements that a system must obey to support the PL models, and also describes how PL programs can be correctly ported to various system-centric models.

Chapter 4

Specification of System Requirements

To correctly and efficiently implement a memory model, a system designer must first identify the memory ordering optimizations that are safely allowed by the model. The specification of system requirements simplifies this task by directly defining such ordering constraints for a given model. System-centric models inherently provide these requirements by the nature of their specification. In contrast, it is difficult to directly deduce such requirements from the specification of programmer-centric models. Consequently, these models are also typically accompanied by a set of system requirements that are proven sufficient for correctness.

For both classes of models, the low-level system specification plays a key role by directly influencing hardware and system software implementations. Therefore, it is important for such system specifications to be unambiguous and relatively straightforward to convert into efficient implementations. This chapter presents a framework for specifying system requirements that satisfies the above goals.

Section 4.1 presents our framework for specifying system requirements. Section 4.2 uses this framework to express the sufficient requirements for supporting the properly-labeled programs presented in the previous chapter. Section 4.3 uses the same framework to express the system-centric models described in Chapter 2. Conditions for correctly and efficiently porting programs between programmer-centric and system-centric models are presented in Section 4.4. Section 4.5 considers a number of extensions to our specification framework. Finally, Section 4.6 provides a comparison with other related work. Techniques for transforming the system requirement specifications into efficient implementations are covered in Chapter 5.

4.1 Framework for Specifying System Requirements

Given the relative importance of system specifications in helping the designer determine optimizations that may be safely exploited under a specific model, there are a number of desirable properties for such specifications. First, the specification should be precise and complete. Any ambiguity arising from the specification is undesirable and can lead to incorrect implementations. Second, the specification should be general, allowing a wide range of system designs and optimizations. To achieve this goal, the specification should impose as few constraints as necessary to maintain the semantics of the model. Third, it should be relatively straightforward to convert the conditions into efficient implementations, and conversely, to verify if an implementation obeys the conditions. As will be further discussed in Section 4.6, many of the previously proposed specifications fall short of meeting these goals by being either overly restrictive or ambiguous.

This section presents a framework for specifying system requirements that meets the above criteria. Our abstraction extends previous work [Col92, SFC91, AH92a] by adequately modeling essential characteristics of a shared-memory system, such as replication of data and the non-atomicity of memory operations. In addition, a key attribute of our specification methodology is the exclusion of ordering constraints among operations to different locations by observing that such constraints are unnecessary for maintaining the semantics of a model. The above characteristics allow us to easily expose the inherent optimizations allowed by a given model. We begin by describing some terminology and assumptions. We next present our abstraction of a shared-memory system that forms the basis for the framework, and describe our methodology for specifying system requirements based on this abstraction. The framework presented here is an extension of our earlier work in this area [GAG+93].

4.1.1 Terminology and Assumptions for Specifying System Requirements

This section presents the terminology and assumptions used in our framework for specifying system requirements. We describe the notions of a shared-memory system and a shared-memory program, along with our assumptions about executions on canonical uniprocessors that comprise the multiprocessor system. In addition, we describe the notion of a result of an execution and discuss the implications of different interpretations of this notion. Finally we discuss some of the simplifying assumptions in our abstraction of a shared-memory system; we will later revisit these assumptions in Section 4.5.

A typical *shared-memory system* consists of a set of processes,[1] a set of private locations per process representing its private state (e.g., registers) and private memory, a set of shared-memory locations, a set of external I/O devices (e.g., disk, network, terminal) which we refer to as *external devices*, and a set of special I/O locations for communicating with the external devices. Definition 4.1 below describes our model for the system. While we model access to private process state and external I/O through read and write operations, the semantics of such operations may be substantially different from that of simple reads and writes to shared-memory locations. For example, a write to an external output state associated with a disk controller may have the complex semantics of initiating the transfer of data from the disk to memory.

> **Definition 4.1: Abstraction for a Shared-Memory System**
> A *shared-memory system* consists of a set of *processes* and a set of *external devices*. Processes communicate through read and write operations to a set of *shared-memory locations*. Each process also has access to a set of private locations representing its *private state*. Communication between processes and external devices occurs through operations to special locations associated with *external input state* and *external output state*. An external input (output) is written (read) by an external device and read (written) by a process. External devices are also allowed to perform read and write operations to the shared-memory locations. The semantics of read and write operations to the private state of each process and to external input and output states may be different (e.g., due to side-effects) from the simple memory-like behavior of operations to shared-memory locations.

Definition 4.2 describes our model for a *shared-memory program* and its constituent instructions. Each static *instruction* may specify a set of read and write operations to its process' private state, shared memory,

[1] For specification purposes, we assume a canonical system with a virtual processor per process. Therefore, we will use processor and process interchangeably. Chapter 5 describes implementation issues that arise in executing multiple processes per physical processor.

or external input and output states, along with partial ordering and atomicity relations (e.g., to specify an atomic read-modify-write) among these operations; the instruction fetch can be modeled as simply one such read operation. A dynamic instance of an instruction further specifies the actual locations and values that are read and written by the constituent operations. Each static instruction also specifies a set of possible next instructions within its process, which translates to a unique next instruction for a dynamic instance of the instruction (e.g., consider a conditional branch). This partial order among dynamic instructions from each process leads to the concept of *program order*, which is described in Definition 4.3. We have chosen to use the conventional definition of program order which imposes a total order on instruction instances from the same process [SS88], though our framework can be trivially extended to deal with a definition of program order that only imposes a partial order among instructions from the same process [GM92]. Definition 4.4 extends the notion of program order from instructions to memory operations.

Definition 4.2: Abstraction for a Shared-Memory Program
A *shared-memory program* specifies the following: (a) a sequence of static instructions for each process, and (b) the initial state for the system, including the per process private state, the state of shared-memory locations, and external input and output states. Each *static instruction* may specify the following: (i) read and write operations to its process' private state, shared memory, or external input and output states, (ii) a partial ordering among its read and write operations (we assume that at least the order of any conflicting operations is specified), and (iii) atomicity relations among its read and write operations. In addition, each instruction specifies (either implicitly or explicitly) a set of possible next instructions from the same process; the choice of a unique next instruction within this set is made dynamically in each run of the program based on the values returned by the read operations within the given process. A *dynamic instance of an instruction* provides further information regarding the dynamic locations and values for the read and write operations generated by executing the static instruction, in addition to specifying the next dynamic instruction instance within its process.

Definition 4.3: Program Order among Instruction Instances
The *program order*, denoted by $\xrightarrow{po}$, is a partial order on the dynamic instruction instances in a run of the program. The program order is a total order among instruction instances from the same process, reflecting the order specified by the "next instruction relation" described in Definition 4.2. Dynamic instruction instances from different processes are not comparable by program order.

Definition 4.4: Program Order among Memory Operations
Two memory operations, *o1* and *o2*, are ordered by program order ($o1 \xrightarrow{po} o2$) if either (a) their corresponding instruction instances, *i1* and *i2*, respectively, are ordered by $\xrightarrow{po}$ ($i1 \xrightarrow{po} i2$) as specified by Definition 4.3, or (b) they both belong to the same instruction instance and *o1* is partially ordered before *o2* by the instruction instance (as specified by Definition 4.2(ii)). In contrast to program order among instruction instances, the program order among memory operations from the same process is not necessarily a total order (because an instruction instance does not necessarily impose a total order among its own memory operations).

In specifying multiprocessor executions, we assume that the execution on each process at least obeys the semantics of a canonical uniprocessor by executing the instructions for the process correctly. Since our focus is on specifying multiprocessor behavior however, we do not attempt to formalize the notion of a canonical uniprocessor and rely instead on the intuitive notion. Condition 4.1 specifies what we refer to as the *uniprocessor correctness condition*. Intuitively, the condition requires the instruction instances from any process in a multiprocessor execution of a program to be the same as the instruction instances of a *conceptual* execution of the program for that process on a canonical uniprocessor, given that read operations to shared memory and external input state are made to return the same values as those in the original multiprocessor execution. Note that even though this condition imposes constraints on the set of possible executions, it does not constrain the behavior of operations to shared memory (which is the primary function served by a memory consistency model). Our methodology for constraining shared memory behavior is presented in the next section.

Condition 4.1: Uniprocessor Correctness Condition
The instructions specified by the program for each process are executed in the same way as by a *canonical uniprocessor*, with return values for read operations to shared memory and external input state assumed to be externally supplied. This implies that (a) the semantics for each instruction and its constituent operations are upheld, including the correct choice for the next instruction instance from the process, (b) the dynamic instruction sequence is solely a function of the values returned by read operations to private state, shared memory, and external input state, and (c) read operations to the process' private state return the semantically correct value based on previous operations to private state that appear in program order (for operations with memory-like behavior, the requirement is that a read should return the value of the previous write in program order that modified that location, or the initial value if there is no such write). One important implication of the above constraints is that the number of instruction instances that are in program order before a given instruction instance is required to be finite.

The remaining concept we need to define is the *result* of an execution of a program. The result intuitively comprises of the set of states that are considered *"observable"* during or after a run of the program. For example, the sequence of changes to external output state (e.g., messages printed on a terminal) may be considered as part of the result. Or the state of portions of the shared memory may be considered as observable during different stages of the execution and may therefore be considered part of the result. At the extreme, every state in the system may be considered observable in real time.

The notion of result plays a key role in comparing different memory model specifications or different implementations. Definition 4.5 describes how two specifications or implementations may be compared based on the results they allow (same terminology as in Section 2.1.5). Because the equivalence between two specifications is strongly dependent on the definition of result, it is possible to have two specifications that are equivalent under one interpretation of result and not equivalent under a different interpretation. Therefore, we need a clear definition for result.

Definition 4.5: Comparison of Memory Model Specifications or Implementations Based on Result
Given a formal definition of result, two memory model specifications or implementations *S1* and *S2* are *equivalent* iff for any run of any program, the result in *S1* is possible in *S2* and vice versa. *S1* is considered to be *stricter* than *S2*, and *S2* is considered *more relaxed* than *S1*, iff for any run of any program, the result in *S1* is possible in *S2* but not vice versa. *S1* and *S2* are *distinguishable* if they are not equivalent. *S1* and *S2* are *incomparable* iff they are distinguishable and neither is stricter than the other.

A definition of result that allows too much state to be observable is undesirable because it may limit various optimizations that would otherwise be possible. To better understand this, consider the example of a uniprocessor system with two different notions for result. Assume the first definition for result consists of the final state of memory when the program completes its execution, while the second definition also makes the real-time order in which memory operations are performed observable. With the second definition of result, a system that employs typical uniprocessor optimizations that change the order of memory operations will be considered distinguishable from a canonical uniprocessor that issues its memory operations in order. Meanwhile, the first definition considers the two systems to be indistinguishable as long as the final state of memory is the same, thus allowing a lot more optimizations.

Since the main focus of our specification framework is to define the behavior of operations to shared memory, we will make the *simplifying assumption* of excluding external devices from the our definition of the system. Therefore, we assume read and write operations are only to private state and shared memory, and all such operations are initiated by the processes (in effect ignoring communication between external devices and processes through external input/output state and shared memory). Based on the above assumption, we have adopted a simple notion of result, specified by Definition 4.6, that consists of the set of values returned by read operations in an execution, which uniquely determine the dynamic instruction instances for each process in a run of the program (by Condition 4.1).

Definition 4.6: Simple Notion for Result of an Execution
We assume the *result* of an execution of a shared-memory program consists of the set of values returned by read operations to private state and shared memory. Since we require that all executions obey the uniprocessor correctness condition (Condition 4.1), the above set of values uniquely determines the dynamic instruction instances for each process.

Even though the above definition for result is not adequate for modeling real systems due to the exclusion of external devices, it is extremely useful for isolating and specifying the behavior of shared memory. For this reason, many of the previous efforts in specifying memory models have either implicitly or explicitly assumed a similar notion of result. Section 4.5 considers extensions to our framework to encompass a more realistic notion of result. As we will see, many of the complexities that arise, such as the modeling of I/O devices, are also present for uniprocessors and are thus not unique to multiprocessors.

In addition to excluding external devices, we make a few other simplifying assumptions that are described below. The *first* assumption is a restriction on the possible atomicity relations among read and write operations generated by an instruction (Definition 4.2(iii)) and the granularity of data atomically accessed by these operations. We assume three atomic operations: read, write, and read-modify-write, all with the same granularity of access to data (e.g., a word). For an atomic read-modify-write, the read and write are assumed to be to the same location and the read is assumed to occur before the write in program order. The *second* assumption is regarding the locations or addresses dynamically generated by the read and write operations. We assume all locations are specified in terms of logical addresses to simplify identifying operations to the same location.[2] The *third* assumption is that the instruction space is read only. Therefore, we will initially not concern ourselves with instruction fetches being affected by the behavior of shared memory. Section 4.5 describes how each of the above assumptions may be relaxed.

4.1.2 Simple Abstraction for Memory Operations

This section introduces a simple abstraction for memory operations along with a description of how the behavior of shared-memory operations may be constrained to specify a model such as sequential consistency. The main purpose for this section is to build intuition for the more general abstraction that we will present in the next section.

The simple abstraction presented below is inspired by the conceptual model for shared memory depicted in Figure 4.1. As shown, there is conceptually a single copy of memory and each processor accesses this memory through either a read operation, denoted by R, that returns the value of a specific location, or a write operation, denoted by W, that modifies the value of the location.

An *execution* of a program (specified in Definition 4.7) consists of a set of dynamic instruction instances, a set of memory operations specified by the instruction instances, a partial order (i.e., program order) on the instructions and the memory operations, and a total order (called execution order) on all memory operations.

[2] We did not choose virtual addresses since they may have synonyms and we did not choose physical addresses since the correspondence for a given virtual address may vary during an execution.

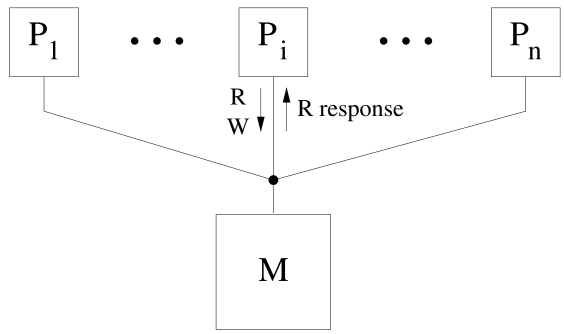


Figure 4.1: Simple model for shared memory.

Definition 4.7: Components of an Execution
An *execution* of a multiprocessor program consists of the following components.
(a) A (possibly infinite) set, *I*, of *dynamic instruction instances* from the run of the program that obeys the uniprocessor correctness condition (Condition 4.1).
(b) A set, *O*, of *memory operations* specified by the instruction instances in I. For every instruction instance *i* in I that reads (writes) a memory location, the set O contains a corresponding read (write) memory operation.
(c) A partial order, called the *program order* (denoted by $\xrightarrow{po}$), on the instruction instances in I (Definition 4.3) *and* on the memory operations in O (Definition 4.4).
(d) A total order, called the *execution order* (denoted by $\xrightarrow{xo}$), on the memory operations in O. The number of memory operations ordered before any given memory operation by the execution order ($\xrightarrow{xo}$) is finite.

To specify the behavior of a system, we will need to place various constraints on the execution order component. We first describe a couple of general conditions on the execution order. Condition 4.2 defines the constraint on the return values for read operations. This condition formalizes the intuitive notion that a read should return the latest value of a given memory location, as implied by Figure 4.1. Similarly, Condition 4.3 describes the notion of atomicity for atomic read-modify-write operations.

Condition 4.2: Return Value for Read Operations
A read memory operation returns the value of the last write operation to the same location that is ordered before the read by the execution order ($\xrightarrow{xo}$). If there is no such write ordered before the read, the initial value of the location is returned.

Condition 4.3: Atomicity of Read-Modify-Write Operations
If R and W are the constituent read and write operations for an atomic read-modify-write to a given location (R $\xrightarrow{po}$ W by definition), then there is no write operation W' to the same location such that R $\xrightarrow{xo}$ W' $\xrightarrow{xo}$ W.

We can specify various consistency models by further constraining the execution order. To illustrate this, Figure 4.2 specifies the conditions for sequential consistency. In addition to obeying the general conditions specified above, the execution order among memory operations is required to be consistent with program order. These requirements trivially (and conservatively) satisfy Scheurich and Dubois' conditions for sequential consistency (Condition 2.2 in Chapter 2).

We define a *valid execution order* as one that satisfies the constraints specified by a memory model. A *valid execution* is then an execution for which a corresponding valid execution order may be constructed.[3]

[3] Our framework covers infinite executions since an execution can comprise of an infinite set of dynamic instructions and the conditions for a model are specified as a restriction on the execution order among these instructions. In practice, one may want to test whether a partial execution (potentially leading to an infinite execution) generates a valid outcome. For this, we can define a partial execution as a set I' of instructions, a set O' of memory operations, and a program order and an execution order relation that represent the instructions

Conditions on $\xrightarrow{xo}$:
(a) the following conditions must be obeyed:
Condition 4.2: return value for read operations.
Condition 4.3: atomicity of read-modify-write operations.
(b) given memory operations X and Y, if X $\xrightarrow{po}$ Y, then X $\xrightarrow{xo}$ Y.

Figure 4.2: Conditions for SC using simple abstraction of shared memory.

A system correctly implements a model if the result of every execution is the same as the result of a valid execution.

Definition 4.8: Valid Execution Order
An *execution order* is *valid* for a memory model iff it satisfies the conditions (i.e., restrictions on execution orders) specified for that model.

Definition 4.9: Valid Execution
An execution is a *valid execution* under a memory model iff *at least one* valid execution order can be constructed for the execution.

Definition 4.10: Correct Implementation of a Memory Model
A system correctly implements a memory model iff for *every* execution allowed by the system, the result is *the same as* a result of a valid execution (as specified by Definition 4.9).

A typical implementation of a model obeys the constraints on execution order directly (and usually conservatively); i.e., constraints on the execution order are enforced in real time. A system designer is free to violate these constraints as long as the resulting executions *appear* to obey the constraints by generating the same results (Definition 4.10).[4] However, in practice it is difficult for a designer to come up with less restrictive constraints and yet guarantee that the allowable executions will produce the same results as with the original constraints. Therefore, it is important to initially provide the designer with as minimal a set of constraints as possible.

The next section presents a more general abstraction for memory operations that is better suited for providing aggressive specifications that further expose the optimizations allowed by a model, thus making it easier to come up with more efficient implementations. This generality also enables the specification of a much larger set of relaxed models.

4.1.3 A More General Abstraction for Memory Operations

Figure 4.3 depicts the conceptual model for memory that inspires our general abstraction. The abstraction consists of *n* processors, $P_1, \ldots, P_n$, each with a *complete* copy of the shared (writable) memory, denoted as M_i for P_i. Each processing node also has a conceptually infinite memory buffer between the processor and memory on that node. As before, processors access memory using read and write operations. However, due to the presence of multiple copies, we introduce the notion of sub-operations for each memory operation. A read operation R by P_i is comprised of two atomic sub-operations: an initial sub-operation $R_{init}(i)$ and a single

and operations that have been generated so far in the system. The partial execution generates a valid outcome or is valid if it could possibly lead to a valid complete execution; i.e., a partial execution is valid if its sets I' and O' are respectively subset of sets I and O of the instructions and operations of some valid execution. Note here that in forming I' and O', we should not include speculative instructions or operations that may have been generated but are not yet committed in the system.

[4] This would not be possible for models such as linearizability (see Section 2.6 in Chapter 2) where correctness depends on the real time when events occur. In contrast, for the models discussed here, we have assumed that the real time occurrence of events is unobservable to the programmer and is therefore not part of the correctness criteria (e.g., see Definition 4.6).

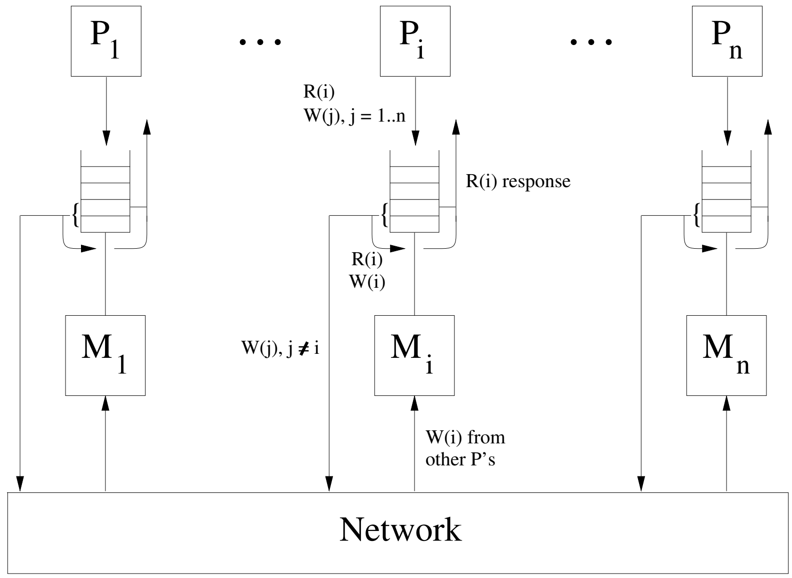


Figure 4.3: General model for shared memory.

read sub-operation *R(i)*. A write operation W by P_i is comprised of at most *(n+1)* atomic sub-operations: an initial write sub-operation $W_{init}(i)$ and at most *n* sub-operations *W(1),...,W(n)*. A read operation on P_i results in the read sub-operation R(i) being placed into P_i's memory buffer. Similarly, a write operation on P_i results in write sub-operations W(1),...,W(n) being placed in its processor's memory buffer. The initial sub-operations R_{init}(i) and W_{init}(i) are mainly used to capture the program order among conflicting operations. Conceptually, R_{init}(i) for a read corresponds to the initial event of placing R(i) into a memory buffer and W_{init}(i) for a write corresponds to the initial event of placing W(1),...,W(n) into the memory buffer.

The sub-operations placed in the buffer are later issued to the memory system (not necessarily in first-in-first-out order). A write sub-operation W(j) by P_i executes when it is issued from the buffer to the memory system and atomically updates its destination location in the memory copy M_j of P_j to the specified value. A read sub-operation R(i) by P_i executes when its corresponding read operation is issued to the memory system and returns a value. If there are any write sub-operations W(i) in P_i's memory buffer that are to the *same* location as R(i), then R(i) returns the value of the last such W(i) that was placed in the buffer (and is still in the buffer). Otherwise, R(i) returns the value of the last write sub-operation W(i) that executed in the memory copy M_i of P_i, or returns the initial value of the location if there is no such write.

The above abstraction is a conceptual model that captures the important properties of a shared-memory system that are relevant to specifying system requirements for memory models. Below, we describe the significance of the following features that are modeled by the abstraction: a complete copy of memory for each processor, several atomic sub-operations for a write, and buffering operations before issue to memory.

Providing each processor with a copy of memory serves to model the multiple copies that are present in real systems due to the replication and caching of shared data. For example, the copy of memory modeled for a processor may in reality represent a union of the state of the processor's cache, and blocks belonging to memory or other caches that are not present in this processor's cache.

The multiple sub-operations attributed to each write operation model the fact that updating multiple copies of a location may be non-atomic. Adve and Hill [AH92a] explain the correspondence to real systems as follows (paraphrased). "While there may be no distinct physical entity in a real system that corresponds to a certain sub-operation, a logically distinct sub-operation may be associated with every operation and a memory copy. For example, updating the main memory on a write corresponds to the sub-operations of the write in memory copies of processors that do not have the block in their cache. Also, while sub-operations may not actually execute atomically in real systems, one can identify a single instant in time at which the sub-operation takes effect such that other sub-operations *appear* to take effect either before or after this time." Note that in reality, write operations may actually invalidate a processor's copy instead of updating it with new data. Nevertheless, the event can be modeled as an update of the logical copy of memory for that processor.

The third feature, i.e., the memory buffer, seems necessary to capture the behavior of many multiprocessor system designs where a processor reads the value of its own write before any of the write's sub-operations take effect in memory. We refer to this feature as *read forwarding*. As with the other features, there may be no physical entity corresponding to this buffer in a real system. For example, this scenario can occur in a cache-coherent multiprocessor if a processor does a write to a location that requires exclusive ownership to be requested and allows a subsequent conflicting read (from the same processor) to return the new value from the cache while the ownership request is pending.

The conceptual model presented above provides the intuition for the formalism we present below. Definition 4.11 describes the components of an execution based on the general abstraction for memory operations. Compared to Definition 4.7, the only differences are the presence of memory sub-operations and the fact that the execution order is defined among these sub-operations as opposed to among memory operations.

Definition 4.11: Components of an Execution with the General Abstraction for Memory Operations
An *execution* of a multiprocessor program consists of the following components.
(a) A (possibly infinite) set, *I*, of *dynamic instruction instances* from the run of the program that obeys the uniprocessor correctness condition (Condition 4.1).
(b) A set, *O*, of *memory operations* specified by the instruction instances in I. For every instruction instance *i* in I that reads (writes) a memory location, the set O contains a corresponding read (write) memory operation.
(c) A set, O_{sub}, of *memory sub-operations* for the operations in O. For every read operation, R, in O by P_i, the set O_{sub} contains a corresponding initial sub-operation $R_{init}(i)$ plus a read sub-operation *R(i)* that denotes the read from the processor's memory copy. For every write operation, W, in O by process P_i, the set O_{sub} contains a corresponding initial write sub-operation, $W_{init}(i)$, plus zero or more corresponding write memory sub-operations *W(1),. . . ,W(n)* denoting writes to different memory copies. Memory sub-operations have the same location and value as their parent memory operations.
(d) A partial order, called the *program order* (denoted by $\xrightarrow{po}$), on the instruction instances in I (Definition 4.3) *and* on the memory operations in O (Definition 4.4).
(e) A total order, called the *execution order* (denoted by $\xrightarrow{xo}$), on the sub-operations in O_{sub}. The number of memory sub-operations ordered before any given memory sub-operation by the execution order ($\xrightarrow{xo}$) is finite.

We will be using the following notation in the specifications that follow. We use R and W to denote any read and write memory operations, respectively, and RW to denote either a read or a write. We use X and Y for either read or write memory operations, and X(i) or Y(i) denote either a read sub-operation R(i) or a write sub-operation W(i), excluding the initial read or write sub-operations which are always denoted explicitly. A condition like "X(i) $\xrightarrow{xo}$ Y(j) for all i,j" implicitly refers to pairs of values for i and j for which both X(i) and Y(j) are defined; *if such an order is imposed by any of the conditions in the specification, we also implicitly assume that X(i) and Y(j) appear in the execution for all such pairs.* For example, given R is issued by P_k, "W(i) $\xrightarrow{xo}$ R(j) for all i,j" reduces to "W(i) $\xrightarrow{xo}$ R(k) for all i" because R(j) is not defined for any value other than k. Finally, we define the conflict order relation. Two memory operations, or two sub-operations from different memory operations, conflict if they are to the same location and at least one is a write. The *conflict order* relation (denoted by $\xrightarrow{co}$) is defined as follows. For an execution order $\xrightarrow{xo}$ and two conflicting memory operations X and Y, X $\xrightarrow{co}$ Y iff X(i) $\xrightarrow{xo}$ Y(i) holds for any i. If X and Y are on different processors and X $\xrightarrow{co}$ Y, then X and Y are also ordered by the *interprocessor conflict order* X $\xrightarrow{co'}$ Y. Note that neither $\xrightarrow{co}$ nor $\xrightarrow{co'}$ are partial orders; for example, R $\xrightarrow{co}$ W and W $\xrightarrow{co}$ R' do not together imply R $\xrightarrow{co}$ R'.

We begin by describing a number of general constraints on the execution order that are assumed to be common among the specifications we present. The first two general conditions relate to the initiation of read and write operations and the termination of write operations. Condition 4.4 is a restriction on the execution order of initial read and write sub-operations with respect to one another. This condition effectively captures the program order among read and write operations to the same location. Meanwhile, Condition 4.5 intuitively requires that certain write sub-operations take effect in finite "time"; a memory model may restrict this requirement to a subset of all write sub-operations. As mentioned in the previous paragraph, a write sub-operation is also required to appear in the execution if any of the conditions in the specification impose an execution order on the given write sub-operation with respect to other sub-operations. We will shortly provide examples that illustrate the need for these two conditions for precisely specifying memory models. Appendix H presents a more aggressive form of the termination condition for the properly-labeled models.

Condition 4.4: Initiation Condition for Reads and Writes
Given memory operations by P_i to the same location, the following conditions hold: If R $\xrightarrow{po}$ W, then R_{init}(i) $\xrightarrow{xo}$ W_{init}(i). If W $\xrightarrow{po}$ R, then W_{init}(i) $\xrightarrow{xo}$ R_{init}(i). If W $\xrightarrow{po}$ W', then W_{init}(i) $\xrightarrow{xo}$ W'$_{init}$(i).

Condition 4.5: Termination Condition for Writes
Suppose a write sub-operation W_{init}(i) (corresponding to operation W) by P_i appears in the execution. The termination condition requires the other *n* corresponding sub-operations, *W(1),. . . ,W(n)*, to also appear in the execution. A memory model may restrict this condition to a subset of all write sub-operations.

The next two conditions are similar to Conditions 4.2 and 4.3 from the previous section, except they are extended to deal with multiple write sub-operations. Condition 4.6 formally captures the semantics of return values for reads that was presented in the intuitive description of our abstraction. Appendix E presents an alternative value condition that alleviates the need for the R_{init}(i) and W_{init}(i) sub-operations in the abstraction. Condition 4.7 simply states the atomicity of read-modify-write operations with respect to other writes to the same location.[5]

Condition 4.6: Return Value for Read Sub-Operations
A read sub-operation R(i) by P_i returns a value that satisfies the following conditions. If there is a write operation W by P_i to the same location as R(i) such that W_{init}(i) $\xrightarrow{xo}$ R_{init}(i) and R(i) $\xrightarrow{xo}$ W(i), then R(i) returns the value of the last such W_{init}(i) in $\xrightarrow{xo}$. Otherwise, R(i) returns the value of W'(i) (from any processor) such that W'(i) is the last write sub-operation to the same location that is ordered before R(i) by $\xrightarrow{xo}$. If there are no writes that satisfy either of the above two categories, then R(i) returns the initial value of the location.

Condition 4.7: Atomicity of Read-Modify-Write Operations
If R and W are the constituent read and write operations for an atomic read-modify-write (R $\xrightarrow{po}$ W by definition) on P_j, then for every *conflicting* write operation W' from a different processor P_k, *either* W'(i) $\xrightarrow{xo}$ R(i) and W'(i) $\xrightarrow{xo}$ W(i) for all i *or* R(i) $\xrightarrow{xo}$ W'(i) and W(i) $\xrightarrow{xo}$ W'(i) for all i.

The above abstraction is extremely general and allows us to specify the behavior for a wide range of memory models and implementations. In addition, it is trivial to "emulate" simpler abstractions. For

[5]This condition can be trivially extended to deal with operations such as the load-locked and store-conditional of the Alpha [Sit92]; the requirement would be to disallow conflicting writes from other processors between a load-locked and a corresponding *successful* store-conditional. See Section 5.3.7 for more details on such operations.

Conditions on $\xrightarrow{xo}$:

(a) if R is read operation with constituent sub-operations R_{init}(i) and R(i),
no other sub-operation is ordered between R's sub-operations by $\xrightarrow{xo}$.

(b) if W is a write operation with constituent sub-operations W_{init}(i) and W(1),. . . ,W(n)),
no other sub-operation is ordered between W's sub-operations by $\xrightarrow{xo}$.

(c) the following conditions must be obeyed:

Condition 4.5: termination condition for writes; applies to *all write sub-operations*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(d) given memory operations X and Y, if X $\xrightarrow{po}$ Y, then X(i) $\xrightarrow{xo}$ Y(j) for all i,j.

Figure 4.4: Conservative conditions for SC using general abstraction of shared memory.

example, to emulate an abstraction with a single copy of memory and no buffering (e.g., the abstraction depicted in Figure 4.1), we can add a condition that would require the constituent sub-operations for each read (R_{init}(i) and R(i)) and each write (W_{init}(i) plus W(1),. . . ,W(n)) to appear atomic in the execution order; i.e., no other sub-operations are ordered among the sub-operations of a read or write by $\xrightarrow{xo}$. Alternatively, to model multiple copies of memory without the buffer (e.g., as in Figure 2.17 for WO with no read forwarding), we can require that for every read R by P_i, the R_{init}(i) and R(i) sub-operations for that read appear atomic, and for every write W by P_i, the W_{init}(i) and W(i) sub-operations for that write appear atomic. Another way to achieve the above is to require W $\xrightarrow{po}$ R to imply W(i) $\xrightarrow{xo}$ R(i) for all i, which effectively defeats the read forwarding optimization. Similarly, to model the buffer without multiple copies (e.g., as in Figure 2.13 for TSO), we can require the sub-operations of a write except for the initial write sub-operation (i.e., W(1),. . . ,W(n)) to appear atomic. Finally, the initiation condition for reads and writes is only needed if the buffer is being modeled and the termination condition for writes is only needed if multiple memory copies are being modeled.

To illustrate the use of our general abstraction, we will present three different specifications of sequential consistency below that successively place fewer restrictions on the execution order while maintaining the same semantics with respect to the results they allow.

Conservative Specification of Sequential Consistency

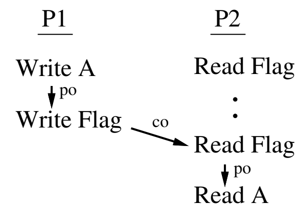
Figure 4.4 shows the conservative specification for sequential consistency based on the general abstraction. This specification is virtually identical to the specification in Figure 4.2 since conditions (a) and (b) effectively eliminate the effect of modeling multiple sub-operations for reads and writes. Therefore, we can effectively ignore the multiple memory copies and read forwarding in the abstraction. Similar to the conditions in Figure 4.2, the last condition requires the execution order among sub-operations to be consistent with the program order among memory operations. Conditions (a), (b), and (d) together alleviate the need for the initiation condition. This is achieved by requiring all sub-operations of one operation to be ordered before any sub-operations of another operation that proceeds it in program order.

To better understand the conservative conditions, consider the program segment in Figure 4.5. The example shows a processor writing a location and setting a flag, while the other processor waits for the flag to be set and then reads the location. Assume the values of A and Flag are initially zero. The right side of the figure shows the program order arcs ($\xrightarrow{po}$) and the conflict order arcs ($\xrightarrow{co}$) between the write and the

```
P1                      P2

a1:  A = 1;             a2:  while (Flag == 0);
b1:  Flag = 1;          b2:  u = A;
```

(a) program text



(b) subset of program order and conflict order

Figure 4.5: Example producer-consumer interaction.

successful read of the flag in an execution of the program. With sequential consistency, the read of A should return the value 1 in every execution. Let us consider how the conditions on execution order in Figure 4.4 ensure this. The program order condition (condition (d)) constrains $\xrightarrow{xo}$ as follows: $W_A(i) \xrightarrow{xo} W_{Flag}(j)$ and $R_{Flag}(i) \xrightarrow{xo} R_A(j)$ for all i,j. If we assume the successful read of the flag returns the value of the write of flag by the first processor, then $W_{Flag}(2) \xrightarrow{xo} R_{Flag}(2)$. Therefore, by transitivity, we have $W_A(i) \xrightarrow{xo} R_A(j)$ for all i,j, which denotes that all sub-operations of the write to A happen before the read A sub-operation, ensuring that the read returns the value 1 in all executions. We next present a slightly more aggressive set of conditions for sequential consistency.

Expressing Scheurich and Dubois' Conditions for Sequential Consistency

Figure 4.6 presents a specification for sequential consistency that is effectively a translation of Scheurich and Dubois' requirements for SC (Condition 2.2 in Chapter 2) to our general abstraction. To match Scheurich and Dubois' abstraction [SD87], conditions (a) and (b) of the specification alleviate the effect of read forwarding. However, we still model the non-atomicity of write sub-operations to different memory copies. Condition (d) of the specification captures the fact that all sub-operations of a read or write should appear in $\xrightarrow{xo}$ before any sub-operations of the next read or write in program order. The second chain (i.e., $W \xrightarrow{co} R \xrightarrow{po} RW$) captures the requirement that any operation that follows a read in program order appears in $\xrightarrow{xo}$ after all sub-operations of the write whose value is returned by the read. This formalizes the notion of "globally performed" [SD87] (see Condition 2.2(c)). Finally, condition (e) of the specification formalizes the cache coherence requirement (Condition 2.2(d)), requiring sub-operations of conflicting writes to occur in the same order in all memory copies. As with the previous specification, conditions (a), (b), and (d) together alleviate the need for the initiation condition.

We now consider the behavior of the program segment in Figure 4.5 under this specification. As in the previous specification, the effect of the program order condition in part (d) is to imply $W_A(i) \xrightarrow{xo} W_{Flag}(j)$ and $R_{Flag}(i) \xrightarrow{xo} R_A(j)$ for all i,j. The termination condition in part (c) guarantees that $W_{Flag}(2)$ (in fact W_{Flag} (i) for all i) will appear in the execution order. As before, we know $W_{Flag}(2) \xrightarrow{xo} R_{Flag}(2)$. By transitivity, this implies $W_A(i) \xrightarrow{xo} R_A(j)$ for all i,j. Therefore, the behavior of the program segment will satisfy sequential consistency. Note that we did not need to use the second chain in part (d) of the specification to reason about the correctness of this example. However, this constraint plays a key role in making writes appear atomic in examples such as those shown in Figure 2.3(c) and (d) in Chapter 2.

Conditions on $\xrightarrow{xo}$:

(a) if R is a read operation by P_i with constituent sub-operations R_{init}(i) and R(i), no other sub-operation is ordered between R_{init}(i) and R(i) by $\xrightarrow{xo}$.

(b) if W is a write operation by P_i with constituent sub-operations W_{init}(i) and W(i), no other sub-operation is ordered between W_{init}(i) and W(i) by $\xrightarrow{xo}$.

(c) the following conditions must be obeyed:

Condition 4.5: termination condition for writes; applies to *all write sub-operations*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(d) given memory operations X and Y, if X and Y are the first and last operations in one of

RW $\xrightarrow{po}$ RW, or

W $\xrightarrow{co}$ R $\xrightarrow{po}$ RW (or more conservatively: W $\xrightarrow{co}$ R)

then X(i) $\xrightarrow{xo}$ Y(j) for all i,j.

(e) if W $\xrightarrow{co}$ W', then W(i) $\xrightarrow{xo}$ W'(i) for all i.

Figure 4.6: Scheurich and Dubois' conditions for SC.

To illustrate that our second specification for SC is less restrictive than the previous one, we consider some of the differences between the orders the two impose on execution orders in our example program segment. The most distinct difference is that the previous specification requires the sub-operations of a write to appear atomic in $\xrightarrow{xo}$ with no intervening sub-operations from a different operation. Therefore, given $W_{Flag}(2)$ $\xrightarrow{xo}$ $R_{Flag}(2)$, the previous specification requires $W_{init:Flag}(1) \xrightarrow{xo} R_{Flag}(2)$ and $W_{Flag}(i) \xrightarrow{xo} R_{Flag}(2)$ for all i. However, the second specification is less restrictive. For example, it is ok for $W_{Flag}(1)$ to appear after $R_{Flag}(2)$ in $\xrightarrow{xo}$ (note that the second chain in part (d) requires that $W_{Flag}(i) \xrightarrow{xo} R_A(j)$ for all i,j). Similarly, given two conflicting writes on different processors, the previous specification requires all sub-operations of one write to occur in $\xrightarrow{xo}$ before any sub-operations of the second write. On the other hand, the second specification is less restrictive because it only requires that the sub-operations for the two writes occur in the same order for all memory copies (part (e) of the specification). By imposing fewer constraints on the execution order, this specification exposes more optimizations that may be exploited when implementing sequential consistency.

An Aggressive Specification of Sequential Consistency

This section presents a specification of sequential consistency which fully exploits the features of our general abstraction to place fewer restrictions on the execution order. To avoid execution order constraints that are unnecessary for maintaining the semantics of a model, we exploit the following observation. Given our simple notion of the result of an execution, any two executions that impose the same order among conflicting sub-operations appear equivalent in terms of results. Therefore, given a valid execution E (Definition 4.9) with a valid execution order $\xrightarrow{xo}$, any execution E' whose execution order $\xrightarrow{xo'}$ maintains the same order among conflicting sub-operations (to the same memory copy) as $\xrightarrow{xo}$ is also a valid execution. The reason is constructing a valid execution order for E' is trivial since $\xrightarrow{xo}$ is one such execution order.[6] Therefore,

[6] By exploiting the read forwarding optimization, the specification presented in this section goes further by actually allowing execution orders that do not maintain the same order among conflicting operations as any valid execution order allowed by the more conservative specifications. The reason is that given W $\xrightarrow{po}$ R on P_i to the same location, we will in some cases allow R(i) to occur before W(i) in the execution order, while this may never be allowed by the conservative specifications. Nevertheless, the initiation condition along with the value condition still require the read to return the value of the write, thus ensuring the results of the executions will be possible under the more conservative specifications.

it is sufficient to only constrain the execution order among conflicting operations in order to specify correct executions.

The previous conditions in Figures 4.4 and 4.6 impose constraints on the execution order between both conflicting and non-conflicting sub-operations. The key insight behind the more aggressive conditions is to impose constraints on the execution order *among conflicting sub-operations only*, as discussed above. Figure 4.7 shows these aggressive conditions for SC. The conditions are expressed in terms of some new relations that are used for notational convenience. The $\xrightarrow{spo}$ relation is a subset of $\xrightarrow{po}$ and the $\xrightarrow{sco}$ relation is composed of $\xrightarrow{co}$ relations (spo and sco stand for significant program order and conflict order, respectively). These relations capture certain $\xrightarrow{po}$ and $\xrightarrow{co}$ orders that are used to constrain the execution order.

Part (a) of the constraints on $\xrightarrow{xo}$ enumerates the four general conditions on $\xrightarrow{xo}$, including the initiation condition for reads and writes. Part (b) places further restrictions on conflicting sub-operations. Consider the various components in this part. In general, the constraint placed on the execution order for two operations X and Y is of the form X(i) $\xrightarrow{xo}$ Y(i) for all i. The first component is uniprocessor dependence which captures the order of operations from the same processor and to the same location. Together with the initiation condition for reads and writes (Condition 4.4) and the value condition (Condition 4.6), this component captures the fact that a read should return either the value of the last write to the same location that is before it in program order or the value of a subsequent write to that location. The second component is coherence which ensures that write sub-operations to the same location take effect in the same order in every memory copy. The third component, denoted as the multiprocessor dependence chain, consists of a set of conditions that capture the relation among conflicting operations that are ordered by $\xrightarrow{po} \cup \xrightarrow{co}$ chains. As we will see shortly, this set of conditions is the key component of our aggressive specification that differentiates it from a conservative specification (e.g., Figure 4.4). The notation "$\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}$+" denotes one or more occurrences of this pattern within the chain. This is similar to regular expression notation; we will also use "*" to denote zero or more occurrences of a pattern. In contrast to the conservative specifications, the aggressive specification imposes an order only among the conflicting endpoints of a $\xrightarrow{po} \cup \xrightarrow{co}$ chain as opposed to enforcing the order at every intermediate point within such a chain. The formal proof of equivalence between these aggressive conditions and Lamport's definition of SC appears in one of our previous publications [GAG$^+$93].

Consider the program segment in Figure 4.5 again, this time with the aggressive specification. The relevant constraint in Figure 4.7 that ensures the correct behavior for this program is the second condition under the multiprocessor dependence chain category. This condition applies as follows: given $W_A \xrightarrow{spo} W_{Flag} \xrightarrow{sco} R_{Flag} \xrightarrow{spo} R_A$, the condition implies W_A(i) $\xrightarrow{xo}$ R_A(i) for all i (this reduces to $W_A(2) \xrightarrow{xo} R_A(2)$ since R_A(i) is only defined for i=2). Therefore, every execution is required to return the value of 1 for the read of A. Note that there are many more valid execution orders possible under the aggressive specifications as compared to the previous conservative specifications. For example, the following execution order, $W_{Flag}(2) \xrightarrow{xo} R_{Flag}(2) \xrightarrow{xo} W_A(2) \xrightarrow{xo} R_A(2)$, is allowed under the aggressive but not under the conservative specifications (because program order is violated on the first processor). As we will see in the next chapter, this flexibility exposes optimizations that are not obvious with the conservative specifications.

We now consider a few examples to motivate the need for the termination and initiation conditions. First, consider the example in Figure 4.5. Without the termination condition, no other condition requires the sub-operation $W_{Flag}(2)$ to occur in the execution. Without this sub-operation appearing in the execution, the

define $\xrightarrow{spo}$**:** X $\xrightarrow{spo}$ Y if X and Y are to *different* locations and X $\xrightarrow{po}$ Y

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co'}$ Y

R $\xrightarrow{co'}$ W $\xrightarrow{co'}$ R

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to *all write sub-operations*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co'}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co'}$ R $\xrightarrow{po}$ RW

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure 4.7: Aggressive conditions for SC.

P1		P2	
a1:	A = 1;	*a2:*	B = 1;
b1:	u = B;	*b2:*	v = A;

(a)

P1	
a1:	A = 1;
b1:	u = A;

(b)

Figure 4.8: Examples illustrating the need for the initiation and termination conditions.

while loop on P2 would never terminate, which violates sequential consistency. Therefore, the termination condition plays an important role in the specification. Figure 4.8(a) provides a more subtle example for the termination condition. The outcome (u,v)=(0,0) does not occur in any SC execution of this program. As long as $W_A(2)$ and $W_B(1)$ appear in the execution, the second chain under the multiprocessor dependence chain in the specification would ensure that this outcome is disallowed. However, without the termination condition, no other condition would force these two sub-operations to appear in the execution and the outcome (u,v)=(0,0) would incorrectly be allowed by the specification.

Figure 4.8(b) shows an example to motivate the need for the initiation condition. In any sequentially consistent execution, the read of A should return the value of 1. However, without the initiation condition, our aggressive specification would fail to ensure this. To illustrate how the initiation condition works, consider the execution order: $W_{init:A}(1) \xrightarrow{xo} R_{init:A}(1) \xrightarrow{xo} R_A(1) \xrightarrow{xo} W_A(1)$. The above is a valid execution order under our aggressive specification (note that the uniprocessor dependence condition does not require $W_A(1) \xrightarrow{xo} R_A(1)$). However, even though the read sub-operation $R_A(1)$ occurs before $W_A(1)$, the initiation condition requires $W_{init:A}(1)$ to occur before $R_{init:A}(1)$, and along with Condition 4.6, this ensures that the read of A will return the value 1.

The important facets of our specification methodology are (a) the abstraction used for shared memory (e.g., modeling sub-operations) and (b) imposing execution order constraints only among conflicting sub-operations. Given this, there are many possible ways to represent the same set of constraints. Nevertheless, as we will see in the next chapter, our notation and representation does lend itself well to mapping the constraints into an implementation.

In summary, even though the aggressive specification in Figure 4.7 places much fewer constraints on the execution order, the resulting executions provide the same results as with the more conservative specifications. While the difference between the two sets of specifications is not discernable by the programmer, from a system design perspective, the aggressive specification exposes the optimizations that are safe thus making it easier for the system designer to implement the model efficiently. The next chapter considers efficient implementation techniques that exploit such aggressive specifications.

The specification framework we have described is general and can be used to express a variety of other memory models. In the following two sections, we use this framework to specify sufficient requirements for supporting properly-labeled programs and to formalize the specifications for various system-centric models.

4.2 Supporting Properly-Labeled Programs

This section specifies the sufficient system requirements for supporting the three programmer-centric models that were introduced in the previous chapter. The proof of correctness for these specifications is similar to the proof presented in our earlier paper [AGG+93] on sufficient conditions for supporting the PLpc model [GAG+92]. Section 4.6 describes related specifications and proofs for other programmer-centric models.

4.2.1 Sufficient Requirements for PL1

We begin by specifying the system requirements for supporting the PL1 model. As we discussed in the previous chapter, the system can exploit the distinction between competing and non-competing labels provided by this model to allow certain operations to be reordered and overlapped. The specification of system requirements is considered sufficient if for any PL1 program, all executions allowed by the specification are sequentially consistent (Definition 3.5 in Chapter 3). For programs that do not qualify as a PL1 program, the PL1 model does not impose any restrictions on the system. To build intuition, we begin with a conservative specification of system requirements that is sufficient for supporting PL1 programs. We next provide a more aggressive specification that imposes execution order constraints among conflicting operations only.

Figure 4.9 presents a conservative specification that is sufficient for supporting the PL1 model. Similar to the conservative specifications for SC presented in Figures 4.4 and 4.6, program order is enforced by imposing constraints on the execution order of both conflicting and non-conflicting operations. The $\xrightarrow{spo}$ and $\xrightarrow{spo'}$ relations identify the program orders that are significant for maintaining the appropriate ordering among memory operations. Rc and Wc denote competing read and write memory operations, respectively; this is based on the labels that are conveyed by a PL1 program. R and W represent any read or write operations, including competing ones. As before, RW denotes either a read or write; RWc is either a competing read or competing write. Below, we describe conditions (a) through (e) on the execution order.

define $\xrightarrow{spo}$,$\xrightarrow{spo'}$:

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

RWc $\xrightarrow{po}$ RWc

X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ RW

RW $\xrightarrow{po}$ Wc

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to write sub-operations for *all competing writes*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) uniprocessor dependence: if X $\xrightarrow{po}$ Y, X,Y conflict, and X=RW and Y=W, then X(i) $\xrightarrow{po}$ Y(i) for all i.

(c) coherence: if Wc $\xrightarrow{co}$ Wc', then Wc(i) $\xrightarrow{xo}$ Wc'(i) for all i.

(d) multiprocessor dependence: if X and Y are the first and last operations in one of

X $\xrightarrow{spo'}$ Y

X $\xrightarrow{spo}$ Y

Wc $\xrightarrow{co}$ Rc $\xrightarrow{spo}$ RW (or more conservatively: Wc $\xrightarrow{co}$ Rc)

then X(i) $\xrightarrow{xo}$ Y(j) for all i,j.

(e) reach: if R $\xrightarrow{rch}$ W, then R(i) $\xrightarrow{xo}$ W(j) for all i,j.

Figure 4.9: Conservative conditions for PL1.

Part (a) requires the general conditions (Conditions 4.4-4.7) to be obeyed. In contrast to the specifications for sequential consistency, the termination condition (Condition 4.5) is imposed only on competing writes. Referring back to Figure 4.5, the program would be considered a PL1 program as long as the write of Flag on P1 and the read of Flag on P2 are labeled as competing. If the example is properly-labeled, the termination condition guarantees that the while loop will eventually terminate by requiring the write sub-operation of Flag in P2's memory copy to appear in the execution order. On the other hand, if the write of Flag on P1 is incorrectly labeled as a non-competing operation, this guarantee is no longer upheld. This is alright, however, since the specification need not guarantee sequentially consistent executions for non-PL1 programs.

The uniprocessor condition (part (b)) captures the order among conflicting operations from the same process and to the same location (similar to the uniprocessor condition in Figure 4.7 for SC). The coherence requirement (part (c)) represents the order among competing writes; its effect on the execution order is to ensure competing write sub-operations to the same address execute in the same order in all memory copies. Even though non-competing writes are not directly covered by the coherence requirement, the other conditions ensure that the above property is upheld for all writes as long as the program is properly labeled.

The multiprocessor dependence condition (part (d)) identifies other important program orders and conflict orders. This condition effectively formalizes the sufficient program order and atomicity conditions for supporting PL1 (e.g., see Table 3.1 in Chapter 3). The first two components identify the cases where the execution order has to be consistent with the program order. The first component requires that all sub-operations of one competing operation occur before any sub-operations of a second competing operation that proceeds it in program order. The second component imposes a similar execution order from a competing read to any memory operation that follows it in program order and from any memory operation to a competing write that follows it. The third component in this condition effectively captures the need for making competing writes appear atomic with respect to competing reads that proceed them in the execution order.

Finally, the reach condition (part (e)) captures the order among operations arising from uniprocessor data and control dependences and certain multiprocessor dependences. This condition introduces a new relation, called the reach relation ($\xrightarrow{rch}$), that in some cases orders a read with respect to a write that proceeds it in program order. The next section will motivate the need for and discuss the characteristics of the reach condition.

Consider the example in Figure 4.5 with the conditions for PL1. Assume the read and write to Flag are labeled as competing. Therefore, we have $W_A \xrightarrow{spo} W_{Flag}$ and $R_{Flag} \xrightarrow{spo} R_A$. Given the above, the rules under multiprocessor dependence require $W_A(i) \xrightarrow{xo} W_{Flag}(j)$ and $R_{Flag}(i) \xrightarrow{xo} R_A(j)$ for all i,j. Assume R_{Flag} returns the value of W_{Flag}, which implies $W_{Flag}(2) \xrightarrow{xo} R_{Flag}(2)$. Therefore, by transitivity, we know $W_A(2) \xrightarrow{xo} R_A(2)$, which ensures that the read of A will always return the value 1. Now consider the effect of incorrect labels. For example, assume the write to Flag is labeled as a non-competing operation. In this case, $W_A \xrightarrow{spo} W_{Flag}$ no longer holds, and the conditions no longer guarantee that the read of A would always return the new value, i.e., $R_A(2) \xrightarrow{xo} W_A(2)$ becomes a valid execution order. An analogous situation arises if the read of Flag is incorrectly labeled as non-competing. Again, both of the above cases are fine since the conditions are not required to guarantee SC outcomes for non-PL1 programs.

In what follows, we will describe the reach condition and a relaxation of the uniprocessor correctness condition (Condition 4.1). We will then present the more aggressive specification for supporting the PL1 model. Appendix H describes a more aggressive version of the termination condition for the PL models.

Motivation for the Reach Condition

The main purpose for the reach condition is to disallow anomalous executions that arise if we allow certain "speculative" write sub-operations to take effect with respect to other processors. This type of optimization is possible for systems that have the ability to overlap or reorder a read operation with respect to a following write operation.[7] The need for the reach condition is best described through examples. Figure 4.10 shows four program segments, all of which are properly-labeled (according to PL1) with the competing operations shown in bold. Using these examples, we show that without an extra constraint such as the reach condition, conditions (a)-(d) in Figure 4.9 fail to provide sequentially consistent executions for these programs. We then proceed to describe the reach condition and how it disallows the non-SC executions.

Consider the program segment in Figure 4.10(a). The program shows P1 reading location A, testing its value, and conditionally writing location B, while P2 does the symmetric computation with accesses to A and B interchanged. As usual, assume all memory locations are initialized to zero. On each processor, whether the write occurs depends on the value returned by the read. With the sequential consistency model, neither write occurs in any execution since both conditionals would fail in every possible case. Therefore, the two read operations are non-competing, and since the write operations do not appear in any SC execution, we can safely label both of them as non-competing as well. Given that the program is a PL1 program, any implementation that satisfies the conditions provided in Figure 4.9 should disallow the writes from occurring in any execution as well. Now consider an implementation that speculatively allows writes to occur before the result of the conditional is resolved (e.g., theoretically possible in a system that uses branch prediction with speculative execution past branches). In such an implementation, it is possible to get an execution in

[7] Since the specifications presented for SC in the previous sections disallow this type of optimization, there is no need for an explicit reach condition in those specifications.

```
        P1                      P2
a1: if (A == 1) {       a2: if (B == 1) {
b1:     B = 1;          b2:     A = 1;
c1: }                   c2: }
```

(a)

```
        P1                      P2
a1: u = A;              a2: v = B;
b1: C[u] = 5;           b2: D[v] = 11;
```

(b)

C[0]
C[1]
C[2]
C[3]
C[4]
D[0]
D[1]
D[2]
D[3]
D[4]
A
B

memory layout

```
        P1                                      P2
a1: while (test&set(L1)==0);            a2: while (test&set(L2)==0);
b1: u = A;                              b2: v = B;
c1: C[u] = 5;                           c2: D[v] = 11;
d1: L1 = 0;                             d2: L2 = 0;
```

(c)

```
        P1                      P2
a1: if (A == 1) {       a2: if (B == 0) {
b1:     B = 1;          b2:     while (Flag == 0);
c1: }                   c2: }
d1: Flag = 1;           d2: A = 1;
```

(d)

Figure 4.10: Examples to illustrate the need for the reach condition.

which both conditionals succeed and both writes occur in the execution. The above execution is somewhat anomalous since writes in the future affect the value of previous reads that in turn determine whether the writes should have been executed in the first place. However, without the reach condition, such an anomalous execution with its corresponding execution order is perfectly consistent with conditions (a)-(d) in Figure 4.9. The purpose of the reach requirement is to provide a precise and formal condition that would prohibit the above type of anomalous executions.

Figure 4.10 shows a number of other scenarios where such anomalous executions can occur. The example in Figure 4.10(a) illustrated that certain control dependences need to be upheld for PL1 programs to execute correctly. Figure 4.10(b) shows an example that illustrates the need for upholding certain data dependences as well. Each processor reads a shared-memory location and uses the return value to index an array in memory. Assume all locations are 32-bit words and arrays C and D are each 5 elements long (i.e., C[0..4] and D[0..4]). In addition, assume the following contrived positioning of the locations in memory (depicted in the figure): the location specified by C[11] is the same as location B (i.e., in the C language notation, &(C[11]) == &B) and the location specified by D[5] is the same as location A (i.e., &(D[5]) == &A).[8] Assuming all locations initially have the value zero, all sequentially consistent executions of this program would result in P1 reading the value of 0 for A and accessing C[0] and P2 reading the value of 0 for B and accessing D[0]. Since there are no conflicts among the memory operations, all operations are non-competing. Now consider a non-SC execution where the read of A returns the value of 11 (causing P1 to write to C[11]) and the read of B returns the value of 5 (causing P2 to write to D[5]). This execution conceptually corresponds to guessing/predicting the value of the read operations (analogous to branch prediction), performing the stores based on the guess, and then noticing that the values that were guessed are indeed the values returned by the reads. Similar to the previous example, the anomaly arises from the fact that future stores affect the return value of previous reads which in turn determine the address the stores write to. And again, conditions (a)-(d) in Figure 4.9 fail to disallow this behavior.

To illustrate that scenarios such as the one above may actually occur even when programmers follow strict synchronization disciplines, Figure 4.10(c) shows the same program segment as in Figure 4.10(b) with extra synchronization. Assume that, by convention, the programmer protects operations to array C by lock L1 and operations to array D by lock L2. Even in the presence of this locking strategy, the same anomalous behavior can occur in this example unless we further restrict the execution order with a constraint such as the reach condition. Furthermore, even though no practical system may actually behave in this anomalous way, it is still important to explicitly disallow such behaviors if we hope to provide a complete and formal specification of the system constraints.

The previous examples illustrate the importance of observing certain uniprocessor data and control dependences. The next example, shown in Figure 4.10(d), illustrates the importance of a dependence that is particular to multiprocessors. The example shows two processes synchronizing through a flag. In all sequentially consistent executions, the first processor reads the value 0 for A which causes the conditional to fail. Similarly, the second processor always returns the value 0 for the read of B and thus waits for the flag to be set and then stores to the A location. Therefore, the write to B on the first processor never occurs in any SC execution (and it is therefore safe to label it as non-competing). The operations to Flag are competing, and the operations to location A are non-competing since they are ordered by the flag synchronization in

[8] Assume there is no address range checking on array accesses, as in the C language. The example can be easily written to use pointers instead of arrays to illustrate the point even with range checking.

every execution. Now, consider a hypothetical implementation that allows the write to A on P2 to proceed before the read of B. This can lead to a non-SC execution where P1 and P2 read the value of 1 for A and B, respectively. And again, conditions (a)-(d) in Figure 4.9 fail to disallow this execution. Note that while the dependence on P1 from the read of A to write of B is a uniprocessor control dependence, there is neither a data nor a control dependence from the read of B to the write of A on P2. Therefore, simply observing uniprocessor data and control dependences is not sufficient for guaranteeing correct behavior in this example.

The ultimate goal of the reach condition is to guarantee correctness for properly-labeled programs by imposing the fewest constraints possible. For example, it is not necessary to guarantee correctness for programs that are improperly labeled. Furthermore, even when a program is properly labeled, the reach condition need not impose extra conditions if the other conditions in the specification already guarantee sequentially consistent executions for the program. Unfortunately, defining the reach condition to maintain correctness only for the necessary cases is quite difficult. Therefore, the challenge is to provide a condition that is as aggressive as possible while still maintaining correctness.

The *reach relation*, denoted by $\xrightarrow{rch}$, attempts to identify the relationships between memory operations that are relevant to disallowing the types of anomalous behavior discussed above. Due to the numerous subtleties that arise when one attempts to provide an extremely aggressive reach relation, the definition for this relation is somewhat complex. Therefore, we will only describe the intuition behind our definition of the reach relation below. The formal definition is provided in Appendix F.

Informally, a read operation reaches a write operation (i.e., R $\xrightarrow{rch}$ W) that follows it in program order (R $\xrightarrow{po}$ W) if the read determines whether the write will execute, the address accessed by the write, or the value written by it. This aspect of reach effectively captures the notions of uniprocessor data and control dependence, covering cases such as examples (a)-(c) in Figure 4.10. In addition, the reach condition relates a read followed by a write in program order if the read controls the execution, address, or value written (in case of a write) of another memory operation that is in between R and W in program order and is related to W by certain program orders (e.g., significant program order $\xrightarrow{spo}$). This latter aspect of the reach relation captures dependences that are particular to multiprocessors to cover cases such the example in Figure 4.10(d), and makes the reach relation both model-dependent and specification-dependent. The way we transform the reach relation into a constraint in our conservative specification (Figure 4.9) is to require the sub-operation of a read to occur before any sub-operations of a write that follows it in program order if R $\xrightarrow{rch}$ W. Our aggressive specification will provide a less restrictive condition based on the reach relation.

Our formulation for the reach relation is formal and aggressive, and is simple to translate into correct and efficient implementations. It is conceptually possible to provide less restrictive conditions at a higher level of abstraction, for example by requiring that all properly-labeled programs behave correctly. However, it is difficult to provide such conditions in a way that can be easily used by system designers to determine the type of optimizations they can exploit.

One key observation that makes the reach relation aggressive is that we ultimately care only about dependences from a read to a write. Therefore, even though we delay certain "speculative" writes, we never delay speculative reads. For example, a system that performs branch prediction is allowed to freely issue read operations speculatively past unresolved branches; this optimization is critical for achieving high performance in dynamically scheduled processors and is exploited by several next generation processor designs. Furthermore, writes that do not have any dependence to previous reads can be performed speculatively

```
a1:  if (A == 1) {
b1:      u = B;
c1:      C = 1;
d1:  }
e1:  D = 1;
```

(a)

```
a1:  for (i=0; i < N; i++) {
b1:      B[i] = A[i];
c1:  }
d1:  for (j=0; j < N; j++) {
e1:      D[j] = C[j];
f1:  }
```

(b)

Figure 4.11: Examples to illustrate the aggressiveness of the reach relation.

as well. Figure 4.11 shows a couple of examples to illustrate the aggressiveness of our definition. Assume all operations are labeled as non-competing in both examples. In Figure 4.11(a), the reach relation exists only between the read of A and the write of C. Therefore, the read of B within the conditional and the write to D outside are not included in reach and may be executed before the read of A or write of C. In Figure 4.11(b), the reach relation exists between the read of array A and write of array B, and between the read of array C and write of array D, within each iteration (we are assuming the arrays are non-overlapping). However, there is no reach implied among operations in different iterations or among operations across the two loops, thus allowing substantial reordering and overlap among the different iterations.

Section 4.6 compares our approach to other related work in this area. The next section discusses the relaxation of another ordering issue related to dependences arising from the presence of infinite loops.

Relaxing the Uniprocessor Correctness Condition

One of the consequences of the uniprocessor correctness condition (Condition 4.1) is that it effectively disallows issuing instruction instances that follow infinite loops in a process' program.[9] Otherwise, a requirement such as "the number of instruction instances program ordered before a given instruction instance should be finite" would be violated. A practical way to enforce this condition is to disallow any instructions or operations that follow an unbounded (or potentially non-terminating) loop by program order from being executed until it is known that the loop will terminate. Below, we consider a relaxation of the uniprocessor correctness condition that allows certain instruction instances to execute before a preceding loop actually terminates.

The motivation for the above relaxation comes from the fact that guaranteeing sequentially consistent executions for properly-labeled programs does not necessarily require delaying instructions that follow a potentially non-terminating loop. Furthermore, optimizations such as allowing speculative reads past potentially non-terminating loops to execute before the loop terminates are important in achieving higher performance especially if the loop actually terminates in the execution (analogous to speculative reads past an unresolved branch). Figure 4.12 shows a contrived example to illustrate this. Assume the operations on P1 and P2 are synchronized through the flag synchronization such that all other operations can be correctly labeled as non-competing. After waiting for the flag synchronization, the second processor follows a linked list within a while loop and then performs other operations. Assume that the linked list is acyclic such that the while loop would terminate in every SC execution. Without this knowledge, the system would have to

[9] This condition is used in Definition 4.11(a) which formalizes the notion of an execution.

P1		P2	
a1:	...	*a2:*	while **(Flag == 0);**
b1:	**Flag = 1;**	*b2:*	ptr = Head_List;
		c2:	while (ptr != nil) {
		d2:	u += ptr –> data;
		e2:	ptr = ptr –> next;
		f2:	}
		g2:	v = A;
		h2:	B = v;

Figure 4.12: Example to illustrate optimizations with potentially non-terminating loops.

treat the loop as potentially non-terminating and thus treat the instructions that follow the loop conservatively. However, if we knew that the loop terminates in every SC execution, then it would be safe to reorder or overlap the iterations of the while loop with instructions that follow it (assume locations A and B are disjoint from the locations accessed by the linked list).

Unfortunately, automatically determining that a potentially unbounded loop terminates in every SC execution is quite difficult in general. Furthermore, the optimization of speculatively issuing instructions after a loop is not necessarily safe even when the loop terminates in every SC execution. Therefore, our goal is to determine the cases where such an optimization would be safe (such as the case in the example above) and precisely identify the type of information that is required to make this decision.

Figure 4.13 shows a couple of examples to illustrate the need for some constraints on issuing instructions before a loop is known to terminate. Both examples are properly-labeled with competing operations (if any) shown in bold. Figure 4.13(a) shows an example with a loop that never terminates in any SC execution. Therefore, the write to A on P1 never occurs in any SC execution and the read of A on P2 always returns the value 0, thus causing the conditional to fail. Executing the write of A on P1 speculatively is therefore unsafe since it will lead to a non-SC execution for this properly-labeled program. Figure 4.13(b) shows a different example where the loop terminates in *every* SC execution. Therefore, it may seem safe to speculatively issue the write to B before knowing that the loop will terminate. However, this optimization can also lead to a non-SC execution since P2 may observe the new value for B and thus not issue the write to A which ultimately causes the while loop to not terminate. Even though the second example looks similar to the examples that were disallowed by the reach condition discussed in the previous section, it turns out that the reach condition is ineffective in this case; in the anomalous case, the write on P2 does not appear in the execution and therefore there is no reach relation on P2. The conservative conditions in Figure 4.9 do not allow the anomalous behavior because condition (d) requires all sub-operations of a competing operation to occur in the execution order before any operations that follow it in program order, thus disallowing the write on P1 from being executed early. However, as we will see, the more aggressive specification we present in the next section does not enforce such an order and depends on extra constraints to disallow such anomalous executions.

Appendix G provides the aggressive uniprocessor correctness condition that allows speculative execution of certain instructions while previous loops have not terminated. With these aggressive conditions, a system

	P1		P2
a1:	while (1);	*a2:*	if (A == 1) {
b1:	A = 1;	*b2:*	B = 1;
		c2:	}

(a)

	P1		P2
a1:	while **(A == 0);**	*a2:*	if (B == 0) {
b1:	B = 1;	*b2:*	**A = 1;**
		c2:	}

(b)

Figure 4.13: Unsafe optimizations with potentially non-terminating loops.

can execute an instruction even if it is not known whether the previous loops will terminate, as long as these previous loops are known to terminate in every SC execution. A read memory operation is always allowed to execute before it is known whether the previous loops in program order will terminate. Therefore, as with the reach condition, we always allow reads to execute speculatively. In contrast, a write memory operation is not allowed to execute if the previous loop is not known to terminate in every SC execution; this restriction covers examples such as that in Figure 4.13(a). For a write memory operation to occur in the execution order before operations from a previous loop, there should also be no read operations from the previous loops that are ordered before the write by the $\xrightarrow{rch'}$ relation (the slight difference between $\xrightarrow{rch}$ and $\xrightarrow{rch'}$ is described in Appendix F); this latter restriction covers examples such as that in Figure 4.13(b). The information of whether a loop will always terminate in an SC execution is often known to the programmer and can be obtained from the programmer. In fact, most programs are written such that all their SC executions are finite (thus implying that all loops terminate). If the information is not known, then the conservative option is to assume the loop may not terminate in some SC executions (unless compiler analysis can determine otherwise).[10]

The relaxation of the uniprocessor correctness condition (Condition 4.1) as described above changes some of the assumptions described in Section 4.1.1. The basic effect of the relaxation is that given a potentially non-terminating loop, the processor can conceptually follow two paths of execution, one within and one after the loop. The difference between the relaxed version and the original version arises when an execution involves loops that actually do not terminate. First, we still require every instruction instance within the loop to occur in program order before any instruction instant that proceeds the loop. For non-terminating loops however, this allows program order to potentially order an infinite number of instruction instances (from the loop) before an instruction instance after the loop. Second, the notion of correct choice for the next instruction must be changed when there is a non-terminating loop, since the processor is now allowed to follow two paths. Finally, the values returned by read operations no longer uniquely determine the dynamic instruction instances. Of course, the original assumptions still hold for properly-labeled programs if we maintain sufficient conditions similar to those specified in Figure 4.9 (or the more aggressive conditions that are specified next), since for these programs our aggressive uniprocessor correctness condition makes everything appear the same as if the original uniprocessor correctness condition was upheld.

We will be assuming the aggressive form of the uniprocessor dependence condition described above for the system requirement specifications we present for the PL models.

[10]The use of potentially non-terminating loops is not necessarily a program error; consider the scheduling loop in an operating system, for example.

define $\xrightarrow{spo}$,$\xrightarrow{spo'}$**:**

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ Rc

Rc $\xrightarrow{po}$ Wc

Wc $\xrightarrow{po}$ Rc, to *different* locations

Wc $\xrightarrow{po}$ Wc

X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ RW

RW $\xrightarrow{po}$ Wc

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

Wc $\xrightarrow{co'}$ Rc

Rc $\xrightarrow{co'}$ Wc

Wc $\xrightarrow{co'}$ Wc

Rc $\xrightarrow{co'}$ Wc $\xrightarrow{co'}$ Rc

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to write sub-operations for *all competing writes*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: Wc $\xrightarrow{co'}$ Wc

multiprocessor dependence chain: one of

Wc $\xrightarrow{co'}$ Rc $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ {Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ }* {Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo}$ } RW

Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ {Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ }* {Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo}$ } RW

RWc $\xrightarrow{spo'}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo'}$ }+ RWc

Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo'}$ }+ Rc

reach: R $\xrightarrow{rch}$ {W $\xrightarrow{co'}$ R $\xrightarrow{rch}$ }+ W

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure 4.14: Sufficient conditions for PL1.

A More Aggressive Specification for PL1

Figure 4.14 presents the aggressive set of sufficient conditions for supporting the PL1 model. In contrast to the conservative conditions in Figure 4.9, these conditions impose execution order restrictions among conflicting sub-operations only. The format of the specification is quite similar to the aggressive specification for SC in Figure 4.7. As before, we use the $\xrightarrow{spo}$ (and $\xrightarrow{spo'}$) and $\xrightarrow{sco}$ relations for notational convenience in order to capture the relevant $\xrightarrow{po}$ and $\xrightarrow{co}$ orders that are used in constraining the execution order.

The constraints on the execution order consist of two parts. The first part imposes the four general constraints on the execution order. The second part imposes further constraints on the order of conflicting sub-operations whose operations are ordered through certain $\xrightarrow{po} \cup \xrightarrow{co}$ chains. Below, we briefly describe the different components in the second part. As before, the uniprocessor dependence condition captures the order among conflicting operations from same processor. Similar to the aggressive specification of SC, uniprocessor dependence does not impose an order from a write to a conflicting read that follows the write in program order. The coherence requirement captures the order among competing writes, requiring the sub-operations for conflicting writes to execute in the same order in all memory copies. The multiprocessor dependence chains capture the order among certain conflicting operations related by $\xrightarrow{po} \cup \xrightarrow{co}$.

Intuitively, the first three multiprocessor dependence chains primarily impose orders between two non-competing operations or between a non-competing and a competing operation, while the last two chains impose an order among competing operations only. The effect of the first two chains is to order conflicting operations if there is an ordering chain (Definition 3.1 in Chapter 3) between them. For competing operations, conflicting pairs ordered by $\xrightarrow{po} \cup \xrightarrow{co'}$ are captured by either the last two chains or the first chain. Finally, the reach condition captures the order among conflicting operations that are ordered through $\xrightarrow{rch} \cup \xrightarrow{co'}$ chains. The third multiprocessor dependence chain turns out not to be necessary for correctly supporting the PL1 model as defined in the previous chapter. Nevertheless, this chain must be maintained if we adopt the alternative definition for ordering chains presented in Appendix A. Furthermore, as we will discuss in Section 5.7 in Chapter 5, this extra constraint greatly simplifies the support required for transparently migrating processes or threads across physical processors. Finally, adding this extra restriction is not expected to affect the performance of any practical implementations of the PL1 model.

Compared to the conservative conditions for PL1 in Figure 4.9, the aggressive specification exposes more optimizations that can be safely exploited while still guaranteeing SC executions for PL1 programs. While the various specifications provided for SC (Figures 4.4, 4.6, and 4.7) are equivalent based on the result of executions (see Definition 4.5), the aggressive specification for PL1 actually allows a larger set of results compared to the conservative specification. Both specifications guarantee sequentially consistent results for PL1 programs. However, the aggressive specification takes fuller advantage of the fact that the PL1 model does not place any constraints on the memory behavior for non-PL1 programs by allowing a larger set of results for such programs.[11]

4.2.2 Sufficient Requirements for PL2

Compared to the PL1 model, the PL2 model requires a further distinction of competing memory operations conveyed through the *sync* and *non-sync* labels. By convention, we will use acquire and release to refer to sync read and write operations, respectively. Figure 4.15 provides the sufficient conditions for supporting PL2 programs. As before, Rc and Wc represent competing operations. Rc_acq and Wc_rel represent competing sync operations. Finally, R and W represent any read or write, whether non-competing, sync, or non-sync.

The conditions for PL2 are quite similar to those for PL1. The main difference arises in the multiprocessor dependence chains (and the $\xrightarrow{spo}$ relation that is used in these chains). The first three chains order conflicting pairs of operations where at least one operation in the pair is potentially non-competing. The conditions for PL2 are more relaxed than those for PL1. Compared to PL1, the competing operations that compose the first three chains are required to be sync operations, with the writes denoted as releases and the reads denoted as acquires. The reason is ordering chains for non-competing operations in PL2 are required to consist of sync operations (Definition 3.6 from the previous chapter). Therefore, unlike PL1, the program order between

[11] In general, given a conservative specification that imposes constraints among both conflicting and non-conflicting sub-operations, it is possible to come up with an equivalent (in terms of results) aggressive specification that imposes constraints only on conflicting sub-operations. However, the reverse path from an aggressive specification to an equivalent conservative one is not always possible.

define $\xrightarrow{spo}$**,**$\xrightarrow{spo'}$**:**

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ Rc

Rc $\xrightarrow{po}$ Wc

Wc $\xrightarrow{po}$ Rc, to *different* locations

Wc $\xrightarrow{po}$ Wc

X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

Rc_acq $\xrightarrow{po}$ RW

RW $\xrightarrow{po}$ Wc_rel

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

Wc $\xrightarrow{co'}$ Rc

Rc $\xrightarrow{co'}$ Wc

Wc $\xrightarrow{co'}$ Wc

Rc $\xrightarrow{co'}$ Wc $\xrightarrow{co'}$ Rc

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to write sub-operations for *all competing writes*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: Wc $\xrightarrow{co'}$ Wc

multiprocessor dependence chain: one of

Wc_rel $\xrightarrow{co'}$ Rc_acq $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ {Wc_rel $\xrightarrow{sco}$ Rc_acq $\xrightarrow{spo'}$ }* {Wc_rel $\xrightarrow{sco}$ Rc_acq $\xrightarrow{spo}$ } RW

Wc_rel $\xrightarrow{sco}$ Rc_acq $\xrightarrow{spo'}$ {Wc_rel $\xrightarrow{sco}$ Rc_acq $\xrightarrow{spo'}$ }* {Wc_rel $\xrightarrow{sco}$ Rc_acq $\xrightarrow{spo}$ } RW

Wc $\xrightarrow{co'}$ Rc $\xrightarrow{spo'}$ RWc

RWc $\xrightarrow{spo'}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo'}$ }+ RWc

Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo'}$ }+ Rc

reach: R $\xrightarrow{rch}$ {W $\xrightarrow{co'}$ R $\xrightarrow{rch}$ }+ W

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure 4.15: Sufficient conditions for PL2.

non-competing operations and non-sync operations is unimportant. Similar to the PL1 conditions, the third multiprocessor dependence chain is not necessary for supporting the PL2 model, but its presence simplifies process migration and would be required if we adopt the alternative definition for ordering chains presented in Appendix A. The last three chains maintain order between conflicting pairs that are labeled competing. As with PL1, the effect is to order any pair that is ordered by $\xrightarrow{po} \cup \xrightarrow{co'}$ when there is at least one $\xrightarrow{po}$ and one $\xrightarrow{co}$ in the chain.

4.2.3 Sufficient Requirements for PL3

Compared to the PL2 model, the PL3 model requires a further distinction of sync operations that is conveyed through the *loop* and *non-loop* labels. Figure 4.16 provides the sufficient conditions for supporting PL3 programs. As before, Rc and Wc represent competing operations. Rc_acq and Wc_rel represent competing sync operations, whether they are labeled loop or non-loop. Rc_nl_ns and Wc_nl_ns represent a competing read or write that is either non-loop (which is a sub-category of sync) or non-sync. Finally, R and W represent any read or write, whether non-competing, loop, non-loop, or non-sync.

Again, the main difference between this specification and the previous ones for PL1 and PL2 is in the multiprocessor dependence chains (and the $\xrightarrow{spo}$ relation that is used in these chains). The first two chains order conflicting operations where at least one is potentially non-competing. The PL3 conditions do not maintain a chain similar to the third chain in the PL1 or PL2 conditions. The reason is supporting such a chain effectively disallows the optimization of allowing loop writes to be non-atomic. Section 5.7 in Chapter 5 describes how transparent process migration may be supported in light of this relaxation. The third and fourth chains maintain the order between conflicting pairs that are labeled competing if there is an ordering chain between them. Finally, the last two chains impose further constraints on the order of conflicting pairs of non-loop sync or non-sync operations. In general, the order maintained among operations labeled as loop sync is much less constraining than for the non-loop sync and non-sync operations.

4.3 Expressing System-Centric Models

The specification framework presented in Section 4.1 can also be used to express various system-centric models, including those discussed in Chapter 2. Using this framework leads to formal and precise specifications of such models. Furthermore, by constraining the execution order among conflicting sub-operations only, we can provide aggressive conditions that are semantically equivalent to the original specifications and yet expose a larger set of optimizations that may be exploited by system designers.

This section presents the conditions for the two flavors of the release consistency model, RCsc and RCpc [GLL+90, GGH93b]. Appendix I provides the conditions for the other system-centric models described in Chapter 2 (i.e, IBM-370, TSO, PC, PSO, WO, Alpha, RMO, PowerPC). A few of these conditions have been presented in an earlier paper that described our specification framework [GAG+93], using a slightly different format and notation.

Figure 4.17 presents the original conditions for RCsc [GLL+90, GGH93b] translated directly to our specification framework. The RCsc model was originally proposed as a set of implementation conditions for supporting PL programs [GLL+90]. RCsc requires the same categories of labels as PL2 programs

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$, $\xrightarrow{spo''}$**:**

X $\xrightarrow{spo''}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ Rc_nl_ns

Rc $\xrightarrow{po}$ Wc

Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns, to *different* locations

Wc_nl_ns $\xrightarrow{po}$ Wc

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ Rc

Rc $\xrightarrow{po}$ Wc

Wc $\xrightarrow{po}$ Wc

X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

Rc_acq $\xrightarrow{po}$ RW

RW $\xrightarrow{po}$ Wc_rel

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

Wc $\xrightarrow{co'}$ Rc

Rc_nl_ns $\xrightarrow{co'}$ Wc_nl_ns

Wc_nl_ns $\xrightarrow{co'}$ Wc_nl_ns

Rc_nl_ns $\xrightarrow{co'}$ Wc_nl_ns $\xrightarrow{co'}$ Rc

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to write sub-operations for *all competing writes*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: Wc_nl_ns $\xrightarrow{co'}$ Wc_nl_ns

multiprocessor dependence chain: one of

Wc_rel $\xrightarrow{co'}$ Rc_acq $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ {Wc_rel $\xrightarrow{sco}$ Rc_acq $\xrightarrow{spo'}$ }* {Wc_rel $\xrightarrow{sco}$ Rc_acq $\xrightarrow{spo}$ } RW

Wc $\xrightarrow{co'}$ Rc $\xrightarrow{spo'}$ RWc

RWc $\xrightarrow{spo'}$ {Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ }+ RWc

RWc_nl_ns $\xrightarrow{spo''}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo''}$ }+ RWc_nl_ns

Wc_nl_ns $\xrightarrow{sco}$ Rc $\xrightarrow{spo''}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo''}$ }+ Rc_nl_ns

reach: R $\xrightarrow{rch}$ {W $\xrightarrow{co'}$ R $\xrightarrow{rch}$ }+ W

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure 4.16: Sufficient conditions for PL3.

define $\xrightarrow{spo}$,$\xrightarrow{spo'}$:

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

RWc $\xrightarrow{po}$ RWc

X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

Rc_acq $\xrightarrow{po}$ RW

RW $\xrightarrow{po}$ Wc_rel

Conditions on $\xrightarrow{xo}$:

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to write sub-operations for *all competing writes*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) uniprocessor dependence: if X and Y conflict and are the first and last operations in RW $\xrightarrow{po}$ W from P_k, then X(k) $\xrightarrow{xo}$ Y(k).

(c) coherence: if W1 $\xrightarrow{co}$ W2, then W1(i) $\xrightarrow{xo}$ W2(i) for all i.

(d) multiprocessor dependence: given X and Y are the first and last operations in one of following and Y is from P_k

X $\xrightarrow{spo'}$ Y

X $\xrightarrow{spo}$ Y

W $\xrightarrow{co}$ R $\xrightarrow{spo'}$ RW

if Y=R then X(i) $\xrightarrow{xo}$ Y(j) for all i,j and if Y=W then X(i) $\xrightarrow{xo}$ Y(j) for all i,j except j=k.

(e) reach: if R $\xrightarrow{rch}$ W, where R and W are from P_k, then R(i) $\xrightarrow{xo}$ W(j) for all i,j except j=k.

Figure 4.17: Original conditions for RCsc.

(the distinction between the PL and PL2 models was described in the previous chapter). As with the PL2 specification, we use the following notation. Rc and Wc represent competing operations. Rc_acq and Wc_rel represent competing sync operations. Finally, R and W represent any read or write, whether non-competing, sync, or non-sync. The conditions for RCsc exploit the read forwarding abstraction in our framework to capture the behavior of this model. This abstraction is also important for capturing the behavior of other models such as PC, RCpc, TSO, PSO, and RMO. The original conditions for RCsc and RCpc do not formally define the behavior with respect to the termination of writes, the reach condition, and the uniprocessor correctness condition (the same is true for most other system-centric models). We assume that termination holds for all competing write sub-operations. We formalize the notion of "uniprocessor data and control dependences" [GLL+90] by using the reach relation ($\xrightarrow{rch}$). The reach relation is only necessary for models such as WO and RCsc that allow the reordering of reads with respect to writes that follow them in program order. Appendix I describes how the reach relation and the aggressive uniprocessor correctness conditions can be adapted for the relevant system-centric models.

Figure 4.18 presents the equivalent conditions for RCsc in their aggressive form where the constraints on the execution order are present among conflicting sub-operations only. These conditions produce the same set of valid executions as the conditions shown in Figure 4.17; the proof for this is presented in an earlier paper [GAG+93]. The "|" symbol used in the regular expressions represents an "or" that signifies a choice among the different options. Since the original specification constrains execution order among non-conflicting operations and at every intermediate point in relevant $\xrightarrow{po} \cup \xrightarrow{co}$ chains, there are a lot of extraneous constraints that are imposed. These extra constraints are clearly exposed when we express the conditions in their aggressive form, especially when we consider the multiprocessor dependence chains.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$, $\xrightarrow{spo''}$, $\xrightarrow{spo'''}$**:**

X $\xrightarrow{spo'''}$ Y if X and Y are the first and last operations in one of

RWc $\xrightarrow{po}$ RWc

X $\xrightarrow{spo''}$ Y if X and Y are the first and last operations in one of

Rc_acq $\xrightarrow{po}$ RW

RW $\xrightarrow{po}$ Wc_rel

X $\xrightarrow{spo'}$ Y if X $\{\xrightarrow{spo'''}\}$ $\{\xrightarrow{rch} \mid \xrightarrow{spo''} \mid \xrightarrow{spo'''}\}*$ Y

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} \mid \xrightarrow{spo''} \mid \xrightarrow{spo'''}\}+$ Y

define $\xrightarrow{sco}$, $\xrightarrow{sco'}$**:**

X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on the same processor

X $\xrightarrow{sco'}$ Y if X and Y are the first and last operations in R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on different processors

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to write sub-operations for *all competing writes*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ $\{$(A $\xrightarrow{sco}$ B $\xrightarrow{spo}$) | (A $\xrightarrow{sco'}$ B $\xrightarrow{spo'}$)$\}+$ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo'}$ $\{$(A $\xrightarrow{sco}$ B $\xrightarrow{spo}$) | (A $\xrightarrow{sco'}$ B $\xrightarrow{spo'}$)$\}+$ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure 4.18: Equivalent aggressive conditions for RCsc.

To better illustrate this, we can compare the sufficient conditions for PL2 shown in Figure 4.15 with the RCsc conditions in Figure 4.18. As we will show below, the conditions for RCsc are a superset of those for PL2. The general conditions (Conditions 4.4-4.7) in part (a) of the specification are identical in the two specifications. We next compare the different components in part (b). The uniprocessor dependence condition is identical. Coherence is stricter in the RCsc specification, since PL2 only requires it among competing operations. The most important difference is in the multiprocessor dependence chains. The chains in RCsc clearly cover every chain specified for the PL2 specification. While chains in the PL2 specification only contain competing operations in the middle of the chain, the RCsc specification maintains a chain even when the intermediate operations in the chain are non-competing. Finally, the reach condition of PL2 is upheld in the RCsc specification in a more conservative way through the second multiprocessor dependence chain (note that the reach relation is used in the definition of $\xrightarrow{spo}$ and $\xrightarrow{spo'}$).

We illustrate the extra constraints imposed by the RCsc multiprocessor dependence chains as compared to the PL2 specification using the example program segments in Figure 4.19. The program segment on the right is identical to the program segment on the left except for the couple of competing operations added to each process' program. Competing operations are shown in bold and the label for such operations is shown beside the operation. Neither program is properly labeled since the operations to locations A and B are competing,

	P1		P2
a1:	A = 1;	*a2:*	u = B;
b1:	B = 1;	*b2:*	v = A;

(a)

	P1			P2	
a1:	A = 1;		*a2:*	u = B;	
b1:	**L1 = 1;**	[sync (rel)]	*b2:*	**L3 = 1;**	[sync (rel)]
c1:	**x = L2;**	[sync (acq)]	*c2:*	**y = L4;**	[sync (acq)]
d1:	B = 1;		*d2:*	v = A;	

(b)

Figure 4.19: Example to illustrate the behavior of the RCsc specifications.

but are not labeled as such. Under sequential consistency, the outcome (u,v)=(1,0) is disallowed in both examples. Meanwhile, the sufficient conditions for PL2 allow this non-SC outcome in both example; this does not violate the PL2 model since neither example is a PL2 program. Similar to the PL2 specification, both the original specification and the equivalent aggressive specification for RCsc allow the non-SC outcome of (u,v)=(1,0) for the example in Figure 4.19(a). However, unlike the PL2 specification, the non-SC outcome is surprisingly disallowed for the example on the right. This arises from the fact that the original conditions for RCsc constrain execution order at every intermediate point in relevant $\xrightarrow{po} \cup \xrightarrow{co}$ chains.

Figure 4.20 shows the aggressive conditions for RCpc, the second flavor of release consistency. In comparison to the RCsc specification in Figure 4.18, the main difference is that the multiprocessor dependence chains enforce fewer ordering constraints. The reason is (a) RCpc allows reads to be reordered with respect to preceding writes in program order even when both are competing, and (b) all writes are allowed to be non-atomic. Since the PL3 model was originally inspired by RCpc, it is interesting to compare the sufficient conditions for supporting PL3 with the conditions for RCpc. As with the PL2 comparison with RCsc, the three main differences are in the coherence requirement, the reach condition, and the multiprocessor dependence chains. Similar to RCsc, RCpc maintains coherence for all write operations and supports the reach condition by incorporating the reach relation into $\xrightarrow{spo}$. RCpc also suffers from extraneous orders that arise because the original conditions impose orders at intermediate points within a chain. The multiprocessor dependence chains in RCpc strictly cover the first four chains in the PL3 specification. However, the last two chains in PL3, which mainly order non-loop or non-sync operations, are not fully covered by RCpc. This makes the two specifications incomparable. Nevertheless, as we will show in the next section, it is still possible to port PL3 programs to the RCpc model relatively efficiently by mapping the operations in a way that would enforce the last two multiprocessor dependence chains in the PL3 specification.

Compared to the original conditions for RCsc and RCpc [GLL$^+$90, GGH93b], the conditions presented here are more precise (e.g., with respect to formally specifying data and control dependences) and expose more aggressive optimizations while maintaining the same semantics. Appendix I provides aggressive specifications for a number of other system-centric models; these specifications are also often more precise and expose more aggressive optimizations compared to the original specifications for the models. The use of a formal framework such as the one we have presented is extremely important for precisely specifying, and for helping designers correctly implement, such models.

The next section describes how we can efficiently port properly-labeled programs to implementations that support various system-centric models.

define $\xrightarrow{spo}$,$\xrightarrow{spo'}$:

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ RWc

Wc $\xrightarrow{po}$ Wc

Rc_acq $\xrightarrow{po}$ RW

RW $\xrightarrow{po}$ Wc_rel

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} \mid \xrightarrow{spo'}\}+$ Y

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on the same processor

Conditions on $\xrightarrow{xo}$:

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to write sub-operations for *all competing writes*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ RW

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure 4.20: Aggressive conditions for RCpc.

4.4 Porting Programs Among Various Specifications

This section describes the process of correctly and efficiently porting programs written for one model or specification to another. We specify the ports by providing a mapping from operations in the source model to operations in the target model. The process of choosing the appropriate mapping of operations involves comparing the source and target system specifications and ensuring that all orders required by the source specification are maintained by the target specification. This task is greatly simplified if both systems are specified using the same framework, which emphasizes the importance of a general specification framework such as the one proposed in this chapter.

We begin by describing the required mappings for porting programs written for the sequential consistency model to other system-centric models. Since we assume no extra information is known about the program, these ports fail to exploit the various optimizations allowed by the more relaxed system-centric models. These mappings build intuition as to how various orders may be enforced in system-centric models. We next discuss how the subtle differences among the system-centric models make it difficult to efficiently port programs among them. Finally, we describe the mappings for porting properly-labeled programs to the various system-centric models. In contrast to the other mappings, the information provided by properly-labeled programs in the form of operation labels enable extremely efficient mapping of such programs to the various system-centric models. Furthermore, the mappings are mechanical, allowing for automatic and efficient portability. A few of the mappings we describe here are similar to mappings previously presented in the context of porting PLpc programs to system-centric model [GAG+92].[12]

[12] The correctness proofs for the PLpc ports are in a different paper [AGG+93]; the proof of correctness for the ports presented in this

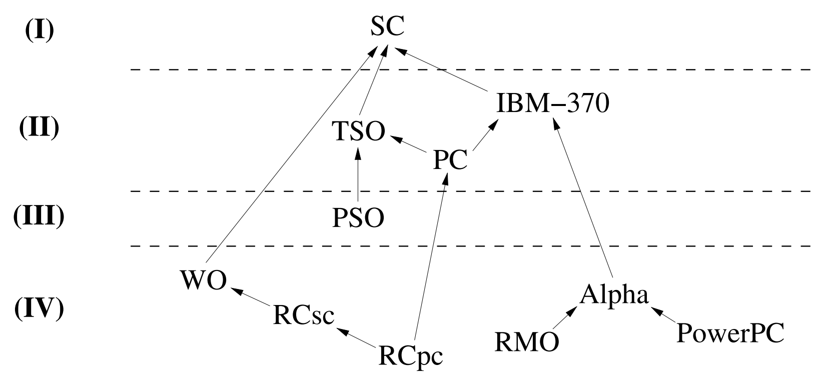


Figure 4.21: Relationship among models (arrow points to stricter model).

4.4.1 Porting Sequentially Consistent Programs to System-Centric Models

This section describes how a program that is written for a sequentially consistent system may be ported to the more relaxed system-centric models. Since no extra information is assumed about the program, we need to conservatively enforce the same ordering constraints as a sequentially consistent system and cannot exploit the reorderings allowed by the target models. Furthermore, enforcing the required orderings can incur extra overhead (e.g., additional instructions) in some of the models. Therefore, depending on the target model, the port may achieve either the same performance or lower performance relative to a comparable system that inherently supports sequential consistency. As we will show later, the extra information conveyed by properly-labeled programs is instrumental in allowing us to achieve higher performance by exploiting the reordering optimizations offered by the relaxed models without violating sequential consistency.

Figure 4.21 depicts the various system-centric models discussed in Chapter 2; this figure appeared in Chapter 2 and is reproduced here for easier reference. The arrows depict the stricter relation, going from the more relaxed to the stricter model (e.g., SC is stricter than WO). This relation is transitive (e.g., SC is stricter than RCpc); models not related by the stricter relation are incomparable. The models are partitioned into four categories depending on the types of program orders that are enforced among operations. SC enforces all program orders. The models in the second category allow write-read reordering. The models in the third category allow write-write reordering as well. Finally, models in the fourth category also allow read-read and read-write reordering. Another main difference among the models (that is not depicted in the figure) is the level of multiple-copy atomicity that is supported for writes.

Table 4.1 shows the sufficient mappings for a correct port from the sequential consistency model to various target system-centric models. The mappings in the table specify how read and write operations in SC should be mapped to operations in the target model, sometimes requiring extra instructions such as fences to be added to the program text (RMW in the table denotes a read-modify-write instruction). To determine the sufficient mappings, we compare the specification for SC (e.g., from Figure 4.7) with the specification of the target model (e.g., specification for RCsc in Figure 4.18). A mapping is sufficient if every order enforced by the SC specification is also enforced the target model after the mapping. In our experience, it is simpler to figure out the appropriate mappings by comparing the aggressive specifications for the two models since conservative specifications often mask the extraneous orders that are maintained by a model. The mappings specified are

chapter are similar to these proofs.

Table 4.1: Sufficient mappings for achieving sequential consistency.

Model	*Mapping to Achieve Sequential Consistency*
IBM-370	(a) for every W $\xrightarrow{po}$ R, at least one of R or W is a synchronization operation, or W $\xrightarrow{po}$ X $\xrightarrow{po}$ R where X is a fence, a synchronization, or a read to same location as W.
TSO	(a) for every W $\xrightarrow{po}$ R, at least one of R or W is part of a RMW, or there is a RMW such that W $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R.
PC	(a) every R is part of a RMW.
PSO	(a) for every W $\xrightarrow{po}$ R, at least one of R or W is part of a RMW, or there is a RMW such that W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R. (b) a STBAR exists between every W $\xrightarrow{po}$ W.
WO	(a) every R and W is mapped to a synchronization operation.
RCsc	(a) every R is mapped to an acquire or non-sync. (b) every W is mapped to a release or non-sync.
RCpc	(a) every R is mapped to an acquire or non-sync. (b) every W is mapped to a release or non-sync. (c) every R is part of a RMW, with W mapped to a release or non-sync.
Alpha	(a) an MB exists between every X $\xrightarrow{po}$ Y.
RMO	(a) a MEMBAR(XY) exists between every X $\xrightarrow{po}$ Y.
PowerPC	(a) a SYNC exists between every X $\xrightarrow{po}$ Y. (b) every R is part of a RMW.

not necessarily unique. In addition, the port will be correct as long as these mappings or a more conservative set of mappings are used. For example, even though we require a fence between every W $\xrightarrow{po}$ W pair for PSO, the port remains correct if there are extra fences also present among other operation pairs.

The mappings for models that require fences to enforce program orders (e.g., IBM-370, PSO, Alpha, RMO, and PowerPC) raise an important issue regarding the static placement of the fence instructions within the program text. For example, the mapping for Alpha requires a memory barrier instruction (MB) between every pair of memory operations X $\xrightarrow{po}$ Y. For correctness, we assume that the fence instruction is placed in such a way that every execution path between X and Y results in the execution of the fence. A simple way to ensure this is to place the fence instruction within the same basic block as the instruction that generates memory operation X or memory operation Y.

Porting SC Programs to IBM-370, WO, RCsc, Alpha, and RMO

We first concentrate on the system-centric models that provide direct mechanisms for enforcing the orders required for supporting sequential consistency: IBM-370, WO, RCsc, Alpha, and RMO. IBM-370, Alpha, and RMO are fence-based models and degenerate to SC if sufficient fences are introduced in between memory operations. For example, the Alpha model guarantees SC if every operation pair in program order is separated by a memory barrier (MB). WO and RCsc enforce orders based on operation labels and degenerate to SC if operations are labeled conservatively. For example, RCsc guarantees SC executions if all reads are labeled

P1	P2
a1: A = 1;	*a2:* B = 1;
b1: u = B;	*b2:* v = A;

(a)

P1	P2	P3
a1: A = 1;	*a2:* u = A;	*a3:* v = B;
	b2: B = 1;	*b3:* w = A;

(b)

P1	P2	P3
a1: A = 1;	*a2:* u = A;	*a3:* v = A;
	b2: A = 2;	*b3:* w = A;

(c)

P1	P2
a1: A = 1;	*a2:* u = A;
	b2: v = A;

(d)

Figure 4.22: Examples to illustrate porting SC programs to system-centric models.

as acquires or non-syncs and all writes are labeled as releases or non-syncs.

Porting SC Programs to TSO, PSO, PC, RCpc, and PowerPC

Ports to the remaining system-centric models are more complicated since these models do not provide direct mechanisms for enforcing some relevant program orders or multiple-copy atomicity for writes. For example, the TSO and PSO models do not provide any direct mechanism for enforcing the W $\xrightarrow{po}$ R order. Similarly, the PowerPC model does not directly provide multiple-copy atomicity for any writes; therefore, even if every pair of operations is separated by a fence, it is still possible that SC may be violated. Finally, PC and RCpc do not provide any mechanisms to directly enforce either of the above constraints.

Figure 4.22 shows a number of examples to illustrate these issues. For the program segment in Figure 4.22(a), TSO, PSO, PC, and RCpc do not provide a direct mechanism to disallow the non-SC outcome of (u,v) = (0,0). Similarly, PC, RCpc, and PowerPC do not provide a direct mechanism for enforcing a multiprocessor dependence chain between the write of A on P1 and the read of A on P3 in Figure 4.22(b), and thus allow the non-SC outcome of (u,v,w) = (1,1,0). Figure 4.22(c) shows a similar example to Figure 4.22(b), except with all the operations to the same location. This example distinguishes the behavior of PowerPC from the other models since PowerPC is the only model that does not provide a direct mechanism for disallowing the non-SC outcome of (u,v,w) = (1,1,0). Finally, Figure 4.22(d) shows a degenerate form of the previous example using only two processors. Even for this simple example, the PowerPC model does not provide a direct way to disallow the non-SC outcome of (u,v) = (1,0).

Even though these models fail to provide direct mechanisms for enforcing all the required orders for guaranteeing SC, there are indirect ways to enforce such orders. Consider TSO first. Table 4.1 enumerates several indirect ways to enforce the W $\xrightarrow{po}$ R order in TSO, some of which are directly derived from looking at the definition of $\xrightarrow{spo}$ for this model in Appendix I. Given W $\xrightarrow{po}$ R, if W is part of an atomic read-modify-write operation or if there is a read-modify-write (to any location) between the two operations such that W $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R, then the specification implies W $\xrightarrow{spo}$ R which ends up maintaining the required order through the multiprocessor dependence chains. The above orders are maintained due to the strong restrictions that the original specifications for TSO and PSO place on atomic read-modify-write operations which disallow writes to *any* location from occurring between the read and write of the read-modify-write (other models, except RMO, only disallow writes to the *same* location). As an alternative mapping to the above, the write to read program order is also maintained if the read is part of a read-modify-write. This mapping does not depend on the stricter definition of read-modify-write, though proving the correctness of the latter mapping is slightly more subtle compared to proving the other two mappings [GAG$^+$92, AGG$^+$93]. Referring back to Figure 4.22(a), use of any of the above mappings will disallow the non-SC outcome. The mapping for PSO is similar to TSO, except we also need store barriers (STBAR) between writes to disallow write reordering.

In contrast to TSO and PSO, the PC and RCpc models do not provide as many options for enforcing the write to read program order simply because their constraints on atomic read-modify-writes are less restrictive than those of TSO and PSO. However, the option of requiring the read to be part of a read-modify-write for every W $\xrightarrow{po}$ R still provides the correct behavior. Furthermore, requiring reads to be part of a read-modify-write also solves the lack of atomicity for writes. For example, referring back to Figure 4.22(b), if the read of A on P3 is made to be part of a read-modify-write, then the outcome (u,v,w) = (1,1,0) is disallowed by PC. Since we cannot automatically select the reads in a program that depend on this extra order, the mapping for PC presented in Table 4.1 requires every read to be part of a read-modify-write. The mapping for RCpc is similar, but also requires conservative labels for all reads and writes (similar to RCsc).

Similar to PC and RCpc, the PowerPC model lacks a direct way of providing multiple-copy atomicity. To solve this, we can use the same mapping used for PC, i.e., require every read to be part of a read-modify-write. In conjunction with the use of fences (called SYNC in PowerPC), this mapping enforces sufficient order to guarantee sequential consistency, even for the examples in Figure 4.22(c) and (d) which were particular to PowerPC.

The above mappings for TSO, PSO, PC, RCpc, and PowerPC use atomic read-modify-write operations to indirectly enforce some of the orders required for guaranteeing sequential consistency. In the cases where an operation is not naturally part of an atomic read-modify-write, these mappings require the operation to be converted into a dummy read-modify-write. When a read is converted to a dummy read-modify-write, the write is required to store the same value as the read. Similarly, when a write is converted to a dummy read-modify-write, as for TSO and PSO, the write is required to store the originally specified value regardless of the value returned by the read. Conversion of a read or write to a dummy read-modify-write may be impossible if the implementation does not provide sufficiently general read-modify-write operations. For example, a test&set instruction is not general enough for this purpose. In practice, however, the wide acceptance of instructions similar to Alpha's load-locked and store-conditional [Sit92] effectively alleviates this problem by providing sufficient generality.

The more important concern about converting normal operations into dummy read-modify-write operations is the extra overhead involved with the read-modify-write. One possible solution is to extend the above models to provide direct (and therefore, more efficient) mechanisms for enforcing the required orders. For example, TSO and PSO can be extended with an extra fence instruction that orders W $\xrightarrow{po}$ R. In fact, the Sparc V9 architecture allows the use of memory barriers defined for RMO under TSO and PSO (TSO and PSO were originally defined under Sparc V8), with the MEMBAR(WR) providing the desired functionality. For PC, we need a similar fence mechanism. In addition, we need a mechanism to force certain writes to

Table 4.2: Sufficient mappings for extended versions of models.

Model	*Mapping to Achieve Sequential Consistency*
TSO+	(a) a MEMBAR(WR) exists between every W $\xrightarrow{po}$ R.
PC+	(a) a fence exists between every W $\xrightarrow{po}$ R. (b) every W is mapped to an atomic write.
PSO+	(a) a MEMBAR(WR) exists between every W $\xrightarrow{po}$ R. (b) a STBAR exists between every W $\xrightarrow{po}$ W.
RCpc+	(a) every R is mapped to an non-sync or acquire. (b) every W is mapped to a non-sync or release. (c) a fence exists between every W $\xrightarrow{po}$ R. (d) every W is mapped to an atomic write.
PowerPC+	(a) a SYNC exists between every X $\xrightarrow{po}$ Y. (b) every W is mapped to an atomic write.

appear atomic, for example by labeling them as atomic writes. RCpc requires similar mechanisms to PC, except we can limit the scope of the fence and the atomic write label to competing operations only. Finally, PowerPC can be extended with the atomic write option. We have included the formal specifications for the extended models in Appendix I along with the other specifications. Table 4.2 shows the alternative mappings for the extended versions of the models, shown with a "+" following the name of the model (e.g., TSO+).

Efficiency of Porting SC Programs to System-Centric Models

The ports presented in Tables 4.1 and 4.2 do not exploit the reordering optimizations that are enabled by the system-centric models since they impose program order and atomicity constraints on all operations. Furthermore imposing these constraints often incurs extra overhead compared to the original program executing on a sequentially consistent system. For some of the ports, the overhead is due to the extra fence instructions that are added to the program. For other ports, transforming a read or write operation into a dummy atomic read-modify-write incurs overhead. The effect of these overheads on performance can be quite substantial due to their high frequency. For example, if we assume 50% of the instructions lead to shared-memory operations, adding a fence instruction between every pair of memory instructions (as is required for models such as Alpha, RMO, or PowerPC) leads to 1.5 times more instructions. In fact, WO and RCsc are the only two target models that have the potential of incurring no extra overhead since labeling all memory operations conservatively enforces the required ordering constraints to satisfy SC without requiring extra instructions. However, this requires the underlying implementation to support orderings through labels.

Many of the ordering constraints that are enforced by the above ports are unnecessary for guaranteeing sequential consistency in a typical program. As we will see shortly, the type of information provided by properly-labeled programs allows us to identify the unnecessary constraints and achieve substantially more efficient ports to system-centric models.

4.4.2 Porting Programs Among System-Centric Models

This section considers the issues in porting programs across various system-centric models. A naive way to provide a correct port from any source to any target system-centric model is to pretend the source model is sequential consistency and use the same mappings as those specified in the previous section. Since sequential consistency is stricter than any system-centric model, guaranteeing sequential consistency on the target trivially disallows any executions that are disallowed by the source model. A more efficient port would attempt to only enforce the orderings imposed by the source model.

We consider a port to be efficient if the ported program on the target model does not perform much worse relative to the minimum of the two performance levels described below: (a) the performance of the original program on the source model, and (b) the performance of the program on the target model assuming it is optimized for the target model. According to the above criteria, porting a program to a stricter target model is simple and efficient. For example, referring back to Figure 4.21, porting a program from WO to SC or from RMO to Alpha fall in this category. There is typically no change required in such a port. As an optimization, it is sometimes possible to remove fence instructions if the orders enforced by the fence are implicitly upheld on the target model (e.g., consider a port from Alpha to SC).

In contrast, porting programs from a stricter model to a more relaxed model, or porting programs among incomparable models (i.e., neither model is stricter than the other), may lead to inefficiencies especially when there are subtle differences between the source and target models. Such subtle differences can arise in several areas, e.g., definition of significant program orders, whether read forwarding is allowed, level of write atomicity, requirements on atomic read-modify-writes, and how the reach and termination conditions are enforced. Problems often arise because the constraints imposed by the source model are difficult and inefficient to impose in the destination model. There can be two reasons for this: (i) the constraints in the source model may be extraneous constraints that are not necessary for the correctness of most programs, but need to be enforced in the port since we do not have more information about the program, or (ii) the destination model is deficient since it cannot efficiently enforce certain constraints that are actually required for the correctness of most programs. As an example of the first reason, consider the Alpha model which enforces program order among reads to the same location. This makes the Alpha model a difficult source if we want to port programs to other models, such as WO, RCsc, or RMO, that also relax the program order among reads but do not implicitly maintain program order among reads to the same location. As an example of the second reason, consider porting programs to models such as PC, RCpc, or PowerPC. The fact that these models do not support write atomicity, along with the lack of extra information about the program, can make these models inefficient targets. Of course, models that are related do not exhibit such subtle differences, making ports between them relatively efficient. For example, porting programs between PC and RCpc, or among TSO, PSO, and RMO, is simple. Overall, the closest to an ideal source model is RCpc, since it is virtually the most relaxed system-centric model. Similarly, the closest to an ideal target is SC since it implicitly enforces every constraint, whether it is extraneous or not.

In summary, porting programs among system-centric models can be inefficient due to subtle differences among the models. The fact that we do not have any extra information about the program exacerbates this problem because the port must guarantee that all orders enforced by the source model are also enforced on the target. As we will see in the next section, the information provided by properly-labeled programs provides much more efficient portability across the whole range of system-centric models.

4.4.3 Porting Properly-Labeled Programs to System-Centric Models

While porting a properly-labeled program to a system-centric model requires guaranteeing sequentially consistent executions for the program, the information provided by properly-labeled programs in the form of operation labels allows us to achieve ports that are substantially more efficient than the ports of SC programs specified in Table 4.1.

Tables 4.3, 4.4, and 4.5 show the sufficient mappings for porting PL1, PL2, and PL3 programs.[13] These mappings are not unique and other mappings may be possible. To determine the appropriate mappings, we ensure that any orders imposed by the sufficient conditions for a properly-labeled program (Figures 4.14, 4.15, and 4.16) are also imposed by the specification of the destination model with the specified mapping. Although these mappings are similar in nature to the mappings for SC programs in Table 4.1, they are significantly more selective. For example, while porting an SC program to WO requires all memory operation to be labeled as synchronization, porting a PL1 program requires this for only the competing memory operations. Since competing operations are typically far less frequent that non-competing operations, selective mapping of operations can provide a substantial performance advantage by exploiting the reordering optimizations of the target model.

Properly-labeled programs port efficiently to most of the system-centric models. Furthermore, overheads due to extra fence instructions for models such as IBM-370, PSO, Alpha, RMO, and PowerPC, or dummy read-modify-writes for models such as TSO, PSO, PC, RCpc, and PowerPC, are significantly reduced compared to porting SC programs simply because these additional instructions are used selectively and infrequently. For example, consider the need for dummy read-modify-writes when porting a PL1 program to PC. Without any information about the program, the mapping in Table 4.1 requires every read to be part of a read-modify-write. However, with the PL1 labels, we can limit this requirement to competing reads only. Therefore, the number of dummy read-modify-writes that are added by the port can be quite low because (a) competing reads are typically infrequent, and (b) some of these reads may already be part of a read-modify-write.[14] Table 4.6 shows the ports of properly-labeled programs to the extended versions of some of the system-centric models introduced in Section 4.4.1. While the extended versions were important for porting SC programs more efficiently, the ordinary versions of the models are sufficiently efficient for porting PL programs.

Most of the system-centric models can beneficially exploit the extra information about memory operations as we move from PL1 to PL2 and PL3. The IBM-370, TSO, and PC models have the same mapping for PL1 and PL2 programs since the distinction between competing sync and non-sync operations cannot be exploited by these models. However, PL3 programs lead to potentially more efficient mappings in the above models. Table 4.4 also shows the RCsc model as having the same mapping for PL2 and PL3 programs. While RCsc can actually benefit from the extra information provided by PL3 programs, the more aggressive mapping that arises is complicated and difficult to achieve in practice; therefore, we use the same mappings as for PL2.

Among the system-centric models shown, RCsc is the most efficient model for executing PL1 and PL2 programs and RCpc is the most efficient model for executing PL3 programs. Compared to the other system-centric models, RCsc and RCpc best exploit the reordering optimizations allowed by PL programs and enforce

[13]For the PowerPC mappings shown in Table 4.5, we assume that if we depend on an existing write to the same address (clause (c) for PL1 and PL2, or clause (d) for PL3), any SYNC that needs to be placed after the Rc due to the other clauses is placed after the existing write as well (except the SYNC that may be required between Rc and the existing write).

[14]The fact that PL1 and PL2 programs can be ported reasonably efficiently to the PC, RCpc, and PowerPC models is quite surprising since the latter models do not provide direct mechanisms for providing atomic writes. As explained above, however, it turns out the need for (potentially dummy) read-modify-writes can be limited to competing reads.

the required orders with virtually no extra overhead (such as additional fence instructions). This is somewhat expected since the RC models were originally developed in conjunction with the proper labeling framework. Of course, the sufficient conditions for properly-labeled models allow yet further optimizations not exploited by RCsc and RCpc.

Table 4.3: Sufficient mappings for porting PL programs to system-centric models.

Model	**PL1**	**PL2**	**PL3**
IBM-370	(a) for every Wc $\xrightarrow{po}$ Rc, at least one of Rc or Wc is a synchronization operation, or Wc $\xrightarrow{po}$ X $\xrightarrow{po}$ Rc where X is a fence, a synchronization, or a read to same location as Wc.	(a) same as PL1.	(a) for every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns, at least one of Rc_nl_ns or Wc_nl_ns is a synchronization operation, or Wc_nl_ns $\xrightarrow{po}$ X $\xrightarrow{po}$ Rc_nl_ns where X is a fence, a synchronization, or a read to same location as Wc_nl_ns.
TSO	(a) for every Wc $\xrightarrow{po}$ Rc, at least one of Rc or Wc is part of a RMW, or there is a RMW such that Wc $\xrightarrow{po}$ RMW $\xrightarrow{po}$ Rc.	(a) same as PL1.	(a) for every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns, at least one of R_nl_ns or W_nl_ns is part of a RMW, or there is a RMW such that Wc_nl_ns $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R_nl_ns.
PC	(a) every Rc is part of a RMW.	(a) same as PL1.	(a) every Rc_nl_ns is part of a RMW.
PSO	(a) for every Wc $\xrightarrow{po}$ Rc, at least one of Rc or Wc is part of a RMW, or there is a RMW such that Wc $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ RMW $\xrightarrow{po}$ Rc. (b) a STBAR exists between every W $\xrightarrow{po}$ Wc.	(a) for every Wc $\xrightarrow{po}$ Rc, at least one of Rc or Wc is part of a RMW, or there is a RMW such that Wc $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ RMW $\xrightarrow{po}$ Rc. (b) a STBAR exists between every Wc $\xrightarrow{po}$ Wc and every W $\xrightarrow{po}$ Wc_rel.	(a) for every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns, at least one of Rc_nl_ns or Wc_nl_ns is part of a RMW, or there is a RMW such that Wc_nl_ns $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ RMW $\xrightarrow{po}$ Rc_nl_ns. (b) a STBAR exists between every Wc $\xrightarrow{po}$ Wc and every W $\xrightarrow{po}$ Wc_rel.

Table 4.4: Sufficient mappings for porting PL programs to system-centric models.

Model	**PL1**	**PL2**	**PL3**
WO	(a) every Rc and Wc is mapped to a synchronization operation.	(a) every Rc_acq and Wc_rel is mapped to a synchronization operation. (b) for every Xc $\xrightarrow{po}$ Yc, at least one of X or Y is mapped to a synchronization operation.	(a) every Rc_acq and Wc_rel is mapped to a synchronization operation. (b) for every Rc $\xrightarrow{po}$ Yc or Wc $\xrightarrow{po}$ Wc, at least one operation is mapped to a synchronization operation. (c) for every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns, at least one operation is mapped to a synchronization operation.
RCsc	(a) every Rc is mapped to an acquire. (b) every Wc is mapped to a release.	(a) every Rc_acq is mapped to an acquire; other Rc mapped to non-sync or acquire. (b) every Wc_rel is mapped to a release; other Wc mapped to non-sync or release.	(a) same as PL2.
RCpc	(a) every Rc is mapped to an acquire. (b) every Wc is mapped to a release. (c) every Rc is part of a RMW, with W mapped to non-sync or release.	(a) every Rc_acq is mapped to an acquire; other Rc mapped to non-sync or acquire. (b) every Wc_rel is mapped to a release; other Wc mapped to non-sync or release. (c) every Rc is part of a RMW, with W mapped to non-sync or release.	(a) every Rc_acq is mapped to an acquire; other Rc mapped to non-sync or acquire. (b) every Wc_rel is mapped to a release; other Wc mapped to non-sync or release. (c) every Rc_nl_ns is part of a RMW, with W mapped to non-sync or release.

Table 4.5: Sufficient mappings for porting PL programs to system-centric models.

Model	**PL1**	**PL2**	**PL3**
Alpha	(a) an MB exists between every Rc $\xrightarrow{po}$ RW and RW $\xrightarrow{po}$ Wc. (b) an MB exists between every Xc $\xrightarrow{po}$ Yc.	(a) an MB exists between every Rc_acq $\xrightarrow{po}$ RW and RW $\xrightarrow{po}$ Wc_rel. (b) an MB exists between every Xc $\xrightarrow{po}$ Yc.	(a) an MB exists between every Rc_acq $\xrightarrow{po}$ RW and RW $\xrightarrow{po}$ Wc_rel. (b) an MB exists between every Rc $\xrightarrow{po}$ Rc, Rc $\xrightarrow{po}$ Wc, and Wc $\xrightarrow{po}$ Wc. (c) an MB exists between every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns.
RMO	(a) a MEMBAR(RY) exists between every Rc $\xrightarrow{po}$ Y. (b) a MEMBAR(XW) exists between every X $\xrightarrow{po}$ Wc. (c) a MEMBAR(XY) exists between every Xc $\xrightarrow{po}$ Yc.	(a) a MEMBAR(RY) exists between every Rc_acq $\xrightarrow{po}$ Y. (b) a MEMBAR(XW) exists between every X $\xrightarrow{po}$ Wc_rel. (c) a MEMBAR(XY) exists between every Xc $\xrightarrow{po}$ Yc.	(a) a MEMBAR(RY) exists between every Rc_acq $\xrightarrow{po}$ Y. (b) a MEMBAR(XW) exists between every X $\xrightarrow{po}$ Wc_rel. (c) a MEMBAR(RY) exists between every Rc $\xrightarrow{po}$ Yc. (d) a MEMBAR(WW) exists between every Wc $\xrightarrow{po}$ Wc. (e) a MEMBAR(WR) exists between every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns.
PowerPC	(a) a SYNC exists between every Rc $\xrightarrow{po}$ RW and RW $\xrightarrow{po}$ Wc. (b) a SYNC exists between every Xc $\xrightarrow{po}$ Yc. (c) every Rc is either part of a RMW or is immediately followed in program order by an existing write to the same address.	(a) a SYNC exists between every Rc_acq $\xrightarrow{po}$ RW and RW $\xrightarrow{po}$ Wc_rel. (b) a SYNC exists between every Xc $\xrightarrow{po}$ Yc. (c) every Rc is either part of a RMW or is immediately followed in program order by an existing write to the same address.	(a) a SYNC exists between every Rc_acq $\xrightarrow{po}$ RW and RW $\xrightarrow{po}$ Wc_rel. (b) a SYNC exists between every Rc $\xrightarrow{po}$ Rc, Rc $\xrightarrow{po}$ Wc, and Wc $\xrightarrow{po}$ Wc. (c) a SYNC exists between every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns. (d) every Rc is either part of a RMW or is immediately followed in program order by an existing write to the same address.

Table 4.6: Porting PL programs to extended versions of some system-centric models.

Model	**PL1**	**PL2**	**PL3**
TSO+	(a) a MEMBAR(WR) exists between every Wc $\xrightarrow{po}$ Rc.	(a) same as PL1.	(a) a MEMBAR(WR) exists between every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns.
PC+	(a) a fence exists between every Wc $\xrightarrow{po}$ Rc. (b) every Wc is mapped to an atomic write.	(a) same as PL1.	(a) a fence exists between every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns. (b) every Wc_nl_ns is mapped to an atomic write.
PSO+	(a) a MEMBAR(WR) exists between every Wc $\xrightarrow{po}$ Rc. (b) a STBAR exists between every W $\xrightarrow{po}$ Wc.	(a) a MEMBAR(WR) exists between every Wc $\xrightarrow{po}$ Rc. (b) a STBAR exists between every Wc $\xrightarrow{po}$ Wc and every W $\xrightarrow{po}$ Wc_rel.	(a) a MEMBAR(WR) exists between every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns. (b) a STBAR exists between every Wc $\xrightarrow{po}$ Wc and every W $\xrightarrow{po}$ Wc_rel.
RCpc+	(a) every Rc is mapped to an acquire. (b) every Wc is mapped to a release. (c) a fence exists between every Wc $\xrightarrow{po}$ Rc. (d) every Wc is mapped to an atomic write.	(a) every Rc_acq is mapped to an acquire; other Rc mapped to non-sync or acquire. (b) every Wc_rel is mapped to a release; other Wc mapped to non-sync or release. (c) a fence exists between every Wc $\xrightarrow{po}$ Rc. (d) every Wc is mapped to an atomic write.	(a) every Rc_acq is mapped to an acquire; other Rc mapped to non-sync or acquire. (b) every Wc_rel is mapped to a release; other Wc mapped to non-sync or release. (c) a fence exists between every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns. (d) every Wc_nl_ns is mapped to an atomic write.
PowerPC+	(a) a SYNC exists between every Rc $\xrightarrow{po}$ RW and RW $\xrightarrow{po}$ Wc. (b) a SYNC exists between every Xc $\xrightarrow{po}$ Yc. (c) every Wc is mapped to an atomic write.	(a) a SYNC exists between every Rc_acq $\xrightarrow{po}$ RW and RW $\xrightarrow{po}$ Wc_rel. (b) a SYNC exists between every Xc $\xrightarrow{po}$ Yc. (c) every Wc is mapped to an atomic write.	(a) a SYNC exists between every Rc_acq $\xrightarrow{po}$ RW and RW $\xrightarrow{po}$ Wc_rel. (b) a SYNC exists between every Rc $\xrightarrow{po}$ Rc, Rc $\xrightarrow{po}$ Wc, and Wc $\xrightarrow{po}$ Wc. (c) a SYNC exists between every Wc_nl_ns $\xrightarrow{po}$ Rc_nl_ns. (c) every Wc is mapped to an atomic write.

4.5 Extensions to Our Abstraction and Specification Framework

While our framework is general and extensible, the methodology we have presented so far deals only with data read and data write memory operations. Furthermore, parts of our specification methodology, such as imposing execution order constraints among conflicting operations only, depend heavily upon our simplified definition for the result of an execution. This simplified abstraction is extremely useful for isolating and specifying the behavior of shared memory. Furthermore, for most programmers, the simple abstraction and specifications are sufficient for understanding the behavior of a system. Nevertheless, a small number of programmers, such as system programmers, may require a more general framework that also encompasses other types of operations such as those issued to and from I/O devices. Similarly, system designers must typically deal with ordering semantics for a more general set of events.

To characterize the behavior of realistic shared-memory systems, our framework must be generalized in two ways: (a) include more events, such as events generated by I/O devices, and (b) extend the notion of result to include the effect of some of these events. Appendix J identifies some of the issues that arise in modeling a realistic system and describes possible extensions to our framework to deal with these issues. Many of the issues we discuss are not particular to multiprocessors and occur in uniprocessor systems as well. Furthermore, even uniprocessor designs may sacrifice serial semantics for events such as I/O operations or instruction fetches in order to achieve higher performance. Chapter 5 further describes implementation issues with respect to I/O operations, instruction fetches, and multiple granularity data operations.

4.6 Related Work

This section compares our abstraction and specification framework with other approaches. We also describe related work in specifying sufficient conditions for supporting properly-labeled programs.

4.6.1 Relationship to other Shared-Memory Abstractions

Section 4.1.3 presented our general abstraction for shared memory and enumerated the significance of the three main features modeled by this abstraction: a complete copy of memory for each processor, several atomic sub-operations for a write, and buffering operations before issue to memory. This abstraction is an extension of an earlier abstraction we developed jointly with Adve and Hill of Wisconsin [GAG+93]. Below we discuss some other abstractions for shared-memory that have been proposed as a basis for specifying memory models. We compare these abstractions mainly on the basis of their flexibility for capturing the behavior of various memory models and ordering optimizations. The next section considers the various specification methodologies, most of which are based on the abstractions discussed below.

Dubois et al. [DSB86, SD87] present an abstraction that models the various stages of completion for a memory operation. They use this abstraction to present specifications for both sequential consistency and weak ordering. The notion of "perform with respect to a processor" in this abstraction models the effects of replication and the non-atomicity of writes, essentially capturing the first two features of our abstraction. One of the problems with this abstraction is that the definition of "perform" is based on real time. Another shortcoming of the abstraction is that it does not seem to be powerful enough to capture the read forwarding optimization where the processor is allowed to read the value of its own write before the write takes effect in any memory copy. In fact, it seems difficult to capture the behavior of commercial models such as TSO, PSO, and RMO and other models such as PC, RCsc, and RCpc using Dubois' abstraction.[15] Furthermore, even though models such as SC or WO can be easily modeled without resorting to the notion of read forwarding, more aggressive specifications of such models (e.g., conditions shown for SC in Figure 4.7) benefit from a more general abstraction.

The abstraction proposed by Collier [Col92] is formal and captures replication of data and the non-atomicity of writes. In fact, the first two features in our abstraction, that of a complete memory copy per processor and several atomic sub-operations for writes, are based directly on this abstraction. Collier's abstraction has also been used by other researchers to specify system requirements for memory models (e.g., DRF1 [AH92a]). Yet it has the same shortcoming as Dubois et al.'s abstraction in its inability to capture the read forwarding optimization. Our abstraction subsumes Collier's abstraction. In particular, our abstraction degenerates to Collier's abstraction if we remove the R_{init} and W_{init} sub-operations and require W $\xrightarrow{po}$ R to imply W(i) $\xrightarrow{xo}$ R(i) in the specification when both operations are to the same location. These notions are important, however, for properly capturing the read forwarding optimization.

Sindhu et al. [SFC91] also propose an abstraction which is used to specify the TSO and PSO models. This abstraction is flexible enough to handle the read forwarding optimization, modeling it through a conceptual write buffer that allows a read to return the value of a write before it is retired from this buffer. However, the abstraction fails to capture the non-atomicity of writes which is an important feature for modeling the behavior of several models and systems. More recently, Corella et al. [CSB93] have also proposed an abstraction for specifying the PowerPC model. This abstraction fails to deal with the lack of multiple-copy atomicity when the coherence requirement is not imposed on all writes, and also fails to model the read forwarding optimization.

Yet another way of abstracting the system is to represent it in terms of execution histories [HW90]; Hagit et al. have also used this type of abstraction to specify the hybrid consistency model [AF92]. Effectively, a history represents one processor's view of all memory operations or a combined view of different processors. This type of abstraction is in essence similar to Collier's abstraction and shares the same advantages and disadvantages.

The abstraction used by Gibbons et al. [GMG91, GM92] to formalize the system requirements for properly-labeled (PL) programs and release consistency is the only abstraction we are aware of that captures the same set of features as our general abstraction. That is, they model the existence of multiple copies, the non-atomicity of writes, the out-of-order execution of memory operations, and allowing the processor to read its own write before the write is issued to the memory system.[16] The one shortcoming of this abstraction is that specifications based on it typically model the system at too detailed a level. For example, Gibbons et al.'s specifications [GMG91, GM92] involve more events than we use in our specification and inherently depend on states and state transitions, making it complex to reason with and difficult to apply to system designs with substantially different assumptions.

The abstraction presented in this chapter extends our previous abstraction [GAG+93] in a few ways. First, we added the R_{init}(i) sub-operation. Our original abstraction had a subtle limitation: given R1 $\xrightarrow{po}$ W $\xrightarrow{po}$ R2

[15] See an earlier technical report [GGH93b] for a discussion of this limitation and a possible extension to Dubois' abstraction that remedies it.

[16] While the original M_{base} abstraction [GMG91] did not model out-of-order read operations from the same processor, the non-blocking M_{base} abstraction [GM92] later removed this restriction.

to the same location on P_i, our original definition of the initiation condition would require R1(i) $\xrightarrow{xo}$ W_{init}(i) and W_{init}(i) $\xrightarrow{xo}$ R2(i). This implicitly orders R1 and R2 (i.e., R1(i) $\xrightarrow{xo}$ R2(i)) which turns out to be overconstraining in some specifications. Introducing the R_{init}(i) sub-operation removes this problem. Second, we made the atomic read-modify-write condition (Condition 4.7) more aggressive to allow the read forwarding optimization from a previous write to the read of the read-modify-write. Finally, we simplified the format of the specifications by removing some of the intermediate ordering relations (such as $\xrightarrow{sxo}$ [GAG+93]).

4.6.2 Related Work on Memory Model Specification

This section describes the various approaches that have been proposed for specifying system requirements for memory models. We compare the various specification methodologies primarily based on the level of aggressive optimizations that can be captured and exposed by each technique.

One of the key observations in our specification methodology is that the behavior of most memory models can be captured without constraining the execution order among non-conflicting operations. For example, we showed equivalent conservative and aggressive specifications for sequential consistency (Figures 4.4 and 4.7), where the aggressive specification imposes execution orders among conflicting operations only and yet maintains the same semantics as the conservative specification. Such aggressive specifications expose a much wider range of optimizations and allow the specification to be used for a wider range of system designs.

We originally made the observation that memory models can be specified aggressively by only imposing constraints on conflicting operations as part of our joint work with Adve and Hill [GAG+93]. The above observation has been previously made by others as well. For example, Shasha and Snir [SS88] exploit a similar observation in identifying a minimal set of orders (derived from the program order) that are sufficient for achieving sequential consistency for a given program. Collier [Col92] also uses this observation for proving equivalences between different sets of ordering constraints. However, previous specifications of memory models do not exploit this observation to its full potential. Specifically, many of the specifications impose unnecessary ordering constraints on non-conflicting pairs of memory operations; even Shasha and Snir's implementation involves imposing delays among non-conflicting memory operations that occur in program order. In contrast, our framework presents a unified methodology for specifying ordering constraints that apply to pairs of conflicting memory operations only.

There have been numerous specification techniques that lead to conservative constraints. Dubois et al.'s specification style [DSB86, SD87] places unnecessary constraints on memory ordering since it constrains the execution order among accesses to different locations in a similar way to the conservative conditions for SC in Figure 4.6. This same limitation exists with the specifications for TSO and PSO provided by Sindhu et al. [SFC91] and the specification of release consistency provided by Gibbons et al. [GMG91, GM92]. As discussed above, Collier [Col92] does observe that two sets of conditions are indistinguishable if they maintain the same order among conflicting accesses, yet his methodology for specifying conditions constrains order among non-conflicting operations just like the other schemes. Therefore, none of the above methodologies expose the optimizations that become possible when only the order among conflicting operations is constrained.

Adve and Hill's specification of sufficient conditions for satisfying DRF1 [AH92a] is one of the few specifications that presents ordering restrictions among conflicting memory operations only. However, parts of these conditions are too general to be easily convertible to an implementation. While Adve and Hill provide a second set of conditions that translates more easily into an implementation, this latter set of conditions are not as aggressive and restrict orders among operations to different locations. Finally, because their specification is based on Collier's abstraction, their approach does not easily lend itself to specifying models that exploit the read forwarding optimization.

In designing our specification technique, our primary goals have been to provide a framework that covers both the architecture and compiler requirements, is applicable to a wide range of designs, and exposes as many optimizations as possible without violating the semantics of a memory model. Our specification framework could conceivably be different if we chose a different set of goals. For example, with the general nature of our framework, the designer may have to do some extra work to relate our conditions to a specific implementation. Had we focused on a specific class of implementations, it may have been possible to come up with an abstraction and a set of conditions that more closely match specific designs. Similarly, our methodology of only restricting the order among conflicting operations is beneficial mainly at the architectural level. This complexity would not be very useful if we wanted to only specify requirements for the compiler. And in fact, such complexity is undesirable if the specification is to only be used by programmers to determine the set of possible outcomes under a model (however, we strongly believe programmers should reason with the high-level abstraction presented by programmer-centric models). Nevertheless, we feel the benefit of providing a uniform framework that applies to a wide range of implementations outweighs any of its shortcomings.

In summary, our specification methodology exposes more optimizations and is easier to translate into aggressive implementations than previous methods. Given the generality of our framework, it would be interesting to also use it to specify the system requirements for other models that we have not discussed in this thesis. The fact that a uniform framework may be used for specifying different models can greatly simplify the task of comparing the system implications across the various models.

4.6.3 Related Work on Sufficient Conditions for Programmer-Centric Models

There have been a number of attempts at specifying and proving the correctness of sufficient conditions for supporting various programmer-centric models. The seminal work in this area has been done by our group at Stanford and Adve and Hill at Wisconsin, with some of the work done jointly.

The original papers on the properly-labeled (PL) [GLL+90] and the data-race-free-0 (DRF0) [AH90b] frameworks each provide sufficient conditions for satisfying the relevant programmer-centric model, along with proofs of correctness for these conditions. For the PL work, the sufficient conditions were in the form of the RCsc model. Adve and Hill later extended their data-race-free model to distinguish between acquire and release operations similar to the PL framework, and provided a new set of conditions for satisfying DRF1 [AH93]. Gibbons *et al.* [GMG91, GM92] have also provided sufficient conditions, along with proofs, for supporting properly-labeled programs. The sufficient conditions presented in the first paper [GMG91] were limited to processors with blocking reads, but this restriction was alleviated in a later paper [GM92].

As part of our joint work with Adve and Hill on the PLpc model, we identified the optimizations allowed by this model along with specifying ports of PLpc programs to a few system-centric models [GAG+92]. In a later paper [AGG+93], we formally specified the sufficient conditions for supporting PLpc programs and provided correctness proofs for both these conditions and the conditions specified for porting PLpc programs to other models. The conditions for supporting PLpc programs were specified using our aggressive specification methodology [GAG+93], and therefore imposed execution orders among conflicting operations only, thus exposing a large set of ordering optimizations. The sufficient conditions presented in this chapter for supporting the three properly-labeled models, along with the conditions for porting properly-labeled programs, are an extension of the above work on PLpc. Furthermore, the conditions for porting PL programs provided in this chapter cover a wider range of system-centric model compared to our previous work in specifying such ports for the PLpc model [GAG+92].

Hagit et al. [AF92, ACFW93] have also proposed hybrid consistency as a set of sufficient conditions for supporting a few programmer-centric models that they have defined. However, as we mentioned in Chapter 3, hybrid consistency places severe restrictions on the reordering of operations compared to analogous conditions for PL and DRF programs, partly because some of the programmer-centric models defined by Hagit et al. are overly restrictive.

Compared to the sufficient conditions presented in this chapter (or in our work on PLpc [AGG+93]), many of the other specifications are more conservative and often less precise. The evolution of the reach condition is indicative of the latter point. The main purpose for the reach condition is to disallow the types of anomalous executions that arise if we allow "speculative" write sub-operations to take effect in other processors' memory copies. In most previous work, such conditions were either implicitly assumed or assumed to be imposed by informal descriptions such as "intra-processor dependencies are preserved" [AH90b] or "uniprocessor control and data dependences are respected" [GLL+90]. Some proofs of correctness (e.g., proof of correctness for PL programs executing on the RCsc model [GLL+90]) formalized certain aspects of this condition, but the full condition was never presented in precise terms. Later work by Adve and Hill specified this condition more explicitly in the context of the DRF1 model [AH92a] and proved that it is sufficient for ensuring sequentially consistent results for data-race-free programs executing on models such as WO and RCsc. More recently, we jointly formalized an aggressive form of this condition as part of specifying sufficient conditions for PLpc [AGG+93]. The reach condition presented in this thesis is based upon the above work on PLpc and constitutes the most precise and aggressive set of conditions that we are aware of to eliminate anomalous executions due to speculative writes.

4.6.4 Work on Verifying Specifications

Specifying system requirements often involves subtle issues that may lead to errors in the specification or implementation based on the specification, making automatic verification of specifications and implementations an important research area.

There are several types of verification tools that are interesting. One use is to verify that two specifications are equivalent, or that one is stricter than the other. For example, such a tool may be used to check the equivalence of the aggressive and conservative specifications for various models. Another use is to verify that an implementation satisfies the constraints imposed by a specification. This is somewhat similar to verifying two specifications, except the specification of an implementation may be provided at a much lower level of abstraction compared to the specification of a memory model. Yet a third use is to automatically verify the outcome of small programs or traces or to verify the correctness of simple synchronization primitives under a given specification. A tool that verifies the outcome of small programs or traces may also be used to probabilistically check the equivalence (or stricter relation) between two specifications by comparing the behavior of each specification across a large set of examples (potentially generated in a random fashion). A side benefit of attempting to verify a given specification using any of the above methods is that it will require the description of the model to be formal and precise. This alone can expose a number of subtle issues to the designers of the model.

Park and Dill [PD95] have described a verifier for the Sparc TSO, PSO, and RMO models that is capable of producing the possible outcomes of very small programs and may also be used to verify the correctness of simple synchronization primitives. Hojati et al. [HMTLB95] have also suggested automatic tools for verifying the correctness of an implementation against a set of properties and the correctness of simple programs in the context of the Sparc TSO and PSO models. Finally, Collier [Col92] has suggested a number of simple test programs that may be executed on a target system to identify the type of optimizations that are exploited. These tests are limited in scope, however; they do not check for many of the aggressive optimizations that are used in systems today and the fact that a system passes a given test fails to conclusively determine that the system does not exploit a given optimization. Overall, there remain a lot of opportunities and challenges for research in this area.

4.7 Summary

This chapter described a formal framework for specifying sufficient system requirements that precisely capture the semantics of various memory consistency models. Our specification methodology has two major advantages compared to previous approaches. First, it exposes many aggressive implementation optimizations that do not violate the semantics of a given model. The key observation is to constrain the execution order among conflicting operations only; this turns out to be sufficient for specifying the semantics of the models described in this thesis. The second advantage of our methodology is that converting our specifications into either conservative or aggressive implementations and conversely verifying that an implementation satisfies the specification are relatively straightforward. These characteristics aid the designer in implementing a memory model correctly and efficiently across a wide range of system designs. The next chapter describes how these specifications can be translated into efficient implementations.

Chapter 5

Implementation Techniques

The choice of a memory consistency model often influences the implementation of the entire system, including the hardware, compiler, and runtime or operating system. The three most important issues from an implementation point of view are correctness, performance, and complexity or cost. The memory model, or the corresponding system requirement specification, determines the set of ordering constraints for a correct implementation. The challenge for a designer is to maintain correctness while achieving the full performance potential of the chosen memory model and limiting the complexity or cost of the design.

This chapter describes a wide range of practical techniques for efficiently supporting various memory models, focusing on both architecture and compiler issues. A major portion of the chapter is dedicated to implementation issues related to the architecture, with a focus on scalable designs based on hardware-coherent caches and general interconnects. We begin by describing the notion of cache coherence which is often subsumed by the specification of a memory model (Section 5.1). Section 5.2 enumerates several architectural techniques (e.g., lockup-free caches) that take advantage of the ordering optimizations enabled by relaxed memory models. Since unconstrained ordering optimizations can violate the semantics of a given memory model, the implementation must also provide extra mechanisms that appropriately limit the scope of various optimizations. These mechanisms are described in detail in Section 5.3. Finally, Section 5.4 presents several promising techniques that utilize a higher degree of hardware complexity to enable more aggressive optimizations. While we mainly focus on systems with scalable interconnects and hardware-coherent caches, Sections 5.5 and 5.6 describe implementation techniques that are applicable to systems that use more restrictive interconnects (e.g., bus or ring) or support cache coherence in software.

The implementation of a memory model is influenced by a number of other issues such as the dynamic mapping of processes to processors, interaction with other latency hiding techniques such as prefetching or multiple contexts, and supporting other types of events such as I/O operations. These topics are covered in Sections 5.7-5.9. Finally, to fully realize the performance benefits of relaxed memory models, it is important to also exploit the compiler optimizations that are enabled by the model. To this end, Section 5.10 describes methods for determining the set of safe compiler optimizations for a given model. Discussion of related work is spread throughout the chapter.

5.1 Cache Coherence

The caching of shared data leads to the presence of multiple copies for a given memory location. *Cache coherence* refers to keeping such copies up-to-date with respect to one another. In what follows, we briefly describe issues related to cache coherence and provide an abstraction for it. Section 5.3.2 provides a further formalization of the correctness conditions for cache coherence protocols. A given cache coherence protocol can be usually adapted to support a wide range of memory models, allowing us to leverage the large body of work on correct and efficient cache coherence implementations.

While cache coherence is a prerequisite for achieving correct memory behavior in systems that allow the caching of shared data, it is not a sufficient condition and constitutes only a small subset of the constraints imposed by a memory model. For example, cache coherence does not cover ordering constraints implied by program order relations among operations to different locations. Furthermore, many of the constraints imposed by a memory model apply regardless of whether a system supports caches. Finally, it may be difficult to distinguish the cache coherence protocol from the overall protocol used to maintain consistency in some systems. Nevertheless, in most systems, the cache coherence protocol is an important part of the overall scheme for supporting a memory model.

5.1.1 Features of Cache Coherence

There are two distinct features for any cache coherence protocol: (i) the mechanism for locating all copies of a memory location, and (ii) the mechanism for eliminating stale copies. Bus-based multiprocessors typically use a *snoopy scheme* to locate cached copies, whereby addresses transmitted on the bus are observed by all caches. A more scalable approach is to keep a *directory* per memory location that identifies the list of copies. The simplest representation for the directory information is a full bit vector, as used in DASH [LLG$^+$90], where there are as many bits as the maximum number of sharers. A number of more scalable representations have been proposed based on the observation that most of memory is either not cached or cached by only a few processors at any given time [Web93, Sim92, CKA91].

To keep the copies up-to-date, a write must eliminate the stale data at the other copies. This can be accomplished by either *invalidating* the stale data or *updating* the cached data to the newly written value. The invalidation or update of a copy is often accompanied by an acknowledgement response that signals completion. While most designs are either invalidation-based or update-based, a hybrid design may statically or dynamically choose between the two schemes on a per memory location basis [KMRS88].

Systems that support cache coherence in hardware often maintain coherence at the granularity of several words, typically corresponding to the cache line size. We refer to this as the *coherence granularity*. A larger coherence granularity helps amortize the overhead of state and tag bits in caches and other state bits associated with directory-based schemes. Cache copies are typically fetched at the granularity of a cache line. In addition, invalidations eliminate a whole line even though the write that causes the invalidation is to a single word. Large coherence granularities present an obvious performance trade-off since data with good spatial locality benefit from operations on a cache line granularity, while data with poor spatial locality may experience more cache misses due to false sharing [TLH94].

Another common and important function of the cache coherence protocol is to serialize the effect of simultaneous write operations to a given location. For example, consider two processors simultaneously issuing a write to the same location with different values. Cache coherence ensures that the two writes are observed in the same order by all processors, with the same value persisting at all copies. This is effectively the same notion as the coherence requirement introduced in Chapters 2 and 4. In terms of system requirements discussed in Chapter 4, coherence implies that for two writes, W1 and W2, to the same location, W1(i) $\xrightarrow{xo}$ W2(i) for all i. While most specifications impose this constraints on all writes, some specifications (e.g., sufficient conditions for PL1, PL2, and PL3) limit the scope of this constraint to a subset of writes. However, as we will later discuss in Section 5.3.2, most practical implementations provide this serialization for all writes.

5.1.2 Abstraction for Cache Coherence Protocols

The techniques used to support a memory model are naturally influenced by some inherent characteristics of the underlying cache coherence protocol. This section presents a general abstraction for directory-based cache coherence protocols that captures the relevant characteristics of such protocols. This abstraction provides a concrete base for the various implementation techniques that are discussed in the later sections.

For purposes of describing a cache coherence protocol, it is often useful to distinguish between the following two components: (i) the coherence protocol within a processor's cache hierarchy, and (ii) the coherence protocol across different processors' caches. In what follows, we will discuss the states and events corresponding to the above two components.

Coherence for the Processor Cache Hierarchy

We begin by considering the coherence protocol within a processor cache hierarchy. In general, the cache hierarchy consists of n levels (typically 2 to 3) of caches, often with m levels (typically zero or one) of write through caches and $(n - m)$ levels of write back caches at the lower levels of the hierarchy (farther away from the processor). The use of write back caches often leads to a substantial reduction in write traffic to the memory system. Typically, each cache line has a state and tag associated with it. For a simple write through cache, the state comprises of a single valid bit. For a simple write back cache, there are three states associated with a cache line: *invalid*, *clean (or shared)*, and *dirty (or modified exclusive)*. The clean or shared state signifies that the cached copy is valid, other processors may be caching the data, and memory is up-to-date. The dirty or modified exclusive state signifies that no other processor is caching the data and that the data in memory is stale.

To simplify the design of cache hierarchies, designers often enforce the *inclusion (or subset) property* that requires the contents of each cache to be a subset of the lower cache in the hierarchy. The state bits in each cache are also extended to keep track of the presence or state of the line in the next higher level cache. For example, the dirty state in a write back cache may be expanded to three states: dirty-invalid (no copy in higher level), dirty-clean (clean copy in higher level), and dirty-dirty (dirty copy in higher level).

More sophisticated cache designs may also associate extra states with a cache line. For example, some write through caches maintain multiple valid bits per cache line to obtain better write performance [OMB92]. Similarly, write back caches may maintain more states per line to optimize certain transitions especially in

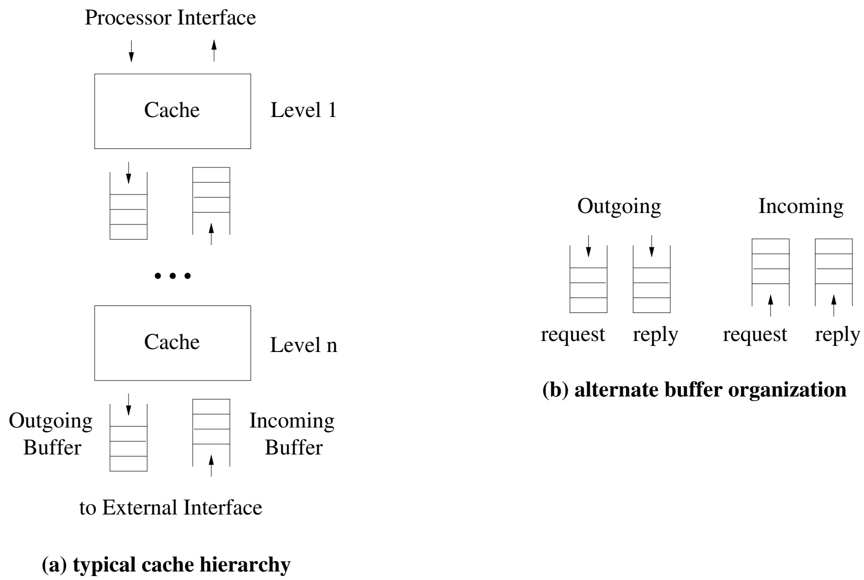


Figure 5.1: Cache hierarchy and buffer organization.

the context of snoopy (bus-based) coherence protocols. For example, a *clean exclusive* state may be used to signify that no other processor is caching the data, yet the data in memory is up-to-date, or a *dirty shared* state may be used to allow the cache to supply data to other processors without updating memory. Finally, some lockup-free cache designs may use extra states per line to signify pending requests to the memory system that are awaiting a response (see Section 5.2.3). The implementation techniques discussed in the following sections can be easily adapted to deal with these extra cache states.

The cache hierarchy is responsible for servicing processor requests and external requests that are forwarded to it. To achieve the above, various messages are exchanged between the processor and cache hierarchy, among the caches in the hierarchy, and between the cache hierarchy and the external interface (i.e., to memory and other processor caches). Figure 5.1(a) illustrates the organization of a typical cache hierarchy. Messages that travel in the direction of the external interface are called *outgoing* messages, while those that travel towards the processor are called *incoming* messages. As shown in the figure, these messages are often buffered among the different levels to enhance performance. Various buffer organizations may be used. For example, Figure 5.1(b) shows an organization with separate buffers for request and reply messages which may make it easier to avoid deadlocks in message handling and buffer management. Section 5.2.3 discusses the impact of buffer organization on the ordering among messages and the types of solutions that may be used to avoid deadlock due to finite buffer space.

The processor issues two types of memory operations to the first level cache: a *read* operation that expects a data reply, and a *write* operation with data that typically expects a completion or acknowledge reply. Each level in the cache hierarchy attempts to service the memory operations issued by the processor and otherwise passes the request to the lower level cache. If the lowest level cache fails to service the request, the request is sent to the external interface and is eventually serviced either by memory or by another processor's cache hierarchy. Similarly, requests from the external interface may be passed from the lower to the higher level caches. Table 5.1 enumerates typical messages that are exchanged within a processor cache hierarchy to service requests from either the processor (outgoing request) or the external interface (incoming request).

Table 5.1: Messages exchanged within the processor cache hierarchy. [wt] and [wb] mark messages particular to write through or write back caches, respectively. (data) and (data*) mark messages that carry data, where (data*) is a subset of a cache line.

Mode	*Outgoing Request*	*Incoming Reply*	*Incoming Request*	*Outgoing Reply*
invalidation-based	read read-exclusive [wb] exclusive [wb] write-read (data*) [wt] write (data*) [wt] write-back (data) [wb] (replacement-hint)	read (data) read-exclusive (data) [wb] exclusive [wb] invalidate-ack	read [wb] read-exclusive [wb] invalidate	read (data) [wb] read-exclusive (data) [wb] (invalidate-ack)
update-based	read update-read (data*) update (data*) write-back (data) [wb] (replacement-hint)	read (data) read-exclusive (data) [wb] update update-exclusive [wb] update-ack	read [wb] update-read (data*) [wb] update (data*)	read (data) [wb] (update-ack)

The table shows messages for both write through and write back caches using either an invalidation or update based coherence scheme. For simplicity, we do not show messages related to prefetch or uncached memory operations, and negative acknowledgement (nack) replies that force a retry of the original request (the significance of nack replies is discussed in Appendix K). The actual protocol and types of messages used within a processor's cache hierarchy are often dictated by the commercial microprocessor used in a design.

Invalidation-Based Cache Hierarchy Consider the invalidation-based scheme with write back caches. We begin by discussing the outgoing requests. A *read* operation by the processor can be serviced by the cache if the line is present in either clean or dirty state. Otherwise, the cache issues an outgoing read request, which is eventually satisfied by an incoming reply that returns a cache-line of data. A *write* operation by the processor can be serviced by the cache if the line is present in dirty state. Otherwise, the cache issues an outgoing read-exclusive request. The incoming read-exclusive reply returns a cache-line of data in addition to signaling that the write has been serialized with respect to other writes to the same location. As an optimization, the cache may issue an exclusive (or upgrade) request if the line is already in clean state (i.e., the cache already has the latest data). An exclusive (or upgrade) reply is similar to the read-exclusive reply but does not return any data. The exclusive request may still need to be satisfied by a read-exclusive reply that carries the new data (or must be reissued as a read-exclusive request) if another processor acquires the cache line in exclusive mode in the interim.

Read-exclusive or exclusive requests that reach the external interface eventually lead to the *invalidation* of copies in other processors' caches. The invalidate-ack reply signals the completion of these invalidations; Section 5.4.1 discusses the distinction between conservative and aggressive notions of completion. Depending on the design, the invalidate-ack reply may either be merged with the read-exclusive or exclusive reply or explicitly sent as a separate message (or as multiple invalidate-ack replies per processor copy). Section 5.3.5 describes designs where the read-exclusive or exclusive replies may be sent earlier than the invalidate-ack reply. The fetch and invalidation granularity are the same in most design, though this is not the case in all designs.

The *replacement* of a dirty line from a cache leads to an outgoing write-back request that carries the only valid copy of the data. Some cache designs also provide the option of signaling the replacement of a clean line through an outgoing replacement-hint request; this latter message can be safely dropped without affecting correctness in most designs (e.g., the directory information will conservatively consider the cache as still sharing a copy). There is typically no reply associated with either type of replacement request. Finally, some designs may overload the outgoing read-exclusive reply for doing a write-back, thus alleviating the need for two separate message types.

We now consider incoming requests. A cache hierarchy may receive an incoming invalidate request caused by a write from another processor. In most cache designs, there is no explicit invalidate-ack reply to signal when the clean copy is actually invalidated in the hierarchy; we further discuss this optimization in Section 5.4.1. The cache hierarchy with a dirty copy of a cache line is also responsible for servicing read or read-exclusive requests from other processors since the data in memory is stale. An incoming read request alters the state of the cache line from dirty to clean, while a read-exclusive request alters the state to invalid. In some designs, such as a snoopy bus-based scheme, the incoming read or read-exclusive request may be broadcast to all processor cache hierarchies as opposed to only the cache hierarchy with the dirty copy. A cache hierarchy that does not have the line in dirty mode simply ignores an incoming read request, and implicitly treats the incoming read-exclusive request as an invalidate request.

A processor write is handled differently in a write through cache. The write leads to an outgoing write request that carries the newly written data. As an optimization, outgoing write requests to the same or consecutive addresses may be merged into a single write request of a larger granularity. Caches that use an allocate-on-write policy may issue a write-read request if the line is not already in a valid state. The write-read request is similar to a write request except it expects a read reply that supplies the data for the portion of the cache line that is not written to. With respect to incoming requests, a write-through cache does not need to service incoming read and read-exclusive requests since memory is always kept up-to-date.

Update-Based Cache Hierarchy We first consider the update-based scheme with write through caches, referring to the message types in Table 5.1. Processor reads work exactly the same as for invalidation-based coherence. Processor writes are also similar, though other cache copies are updated with the new value instead of being invalidated. A processor write generates an outgoing update request that carries the newly written data. The update reply signals that the write has been serialized with respect to other writes to the same location. In designs with more than a single level cache, the update reply (or update-exclusive reply described below) may carry data in order to update lower level caches on its incoming path; this issue is described in more detail in Section 5.3.2. The update-ack reply is analogous to the invalidate-ack reply and signals the completion of the updates. The incoming update-ack reply may either be merged with other replies such as the update reply or may be sent as a separate message. Similar to invalidation-based caches, updates to same or consecutive addresses may be merged into a single update request. Similarly, a write may lead to an update-read request which is analogous to the write-read request; the read and update replies are often merged in this case.

Using a write back cache allows a cache copy to be maintained in a dirty state in some cases, thus alleviating the need to propagate updates on every write. For example, the external interface may respond to an update or update-read request with an update-exclusive or read-exclusive reply if no other processor caches currently have a copy, thus signaling the cache to maintain the copy in dirty state. A cache that maintains a dirty copy may receive incoming read or update-read requests, both of which lead to an outgoing read reply and alter the state of the line to clean; the update-read request also updates the stale cache copy (similar to an incoming update request).

Hybrid protocols may selectively choose between an invalidate or update policy and potentially alter the policy on a per cache line basis. Such protocols use a combination of the message types discussed above to maintain coherence within the processor cache hierarchy.

Coherence across Different Processor Cache Hierarchies

Coherence actions across different processor cache hierarchies are triggered by outgoing requests that flow to the external interface. The coherence protocol is responsible for locating the most up-to-date copy for read requests, and for locating and eliminating stale copies for write requests. Requests such as a read-exclusive request involve both the read and the write functionality. Below, we briefly describe the states and events associated with simple snoopy and directory protocols for supporting this functionality.

In a snoopy protocol, outgoing requests that reach the external interface are broadcast to all processor cache hierarchies. Each cache hierarchy determines whether it needs to take any actions. For example, a read or read-exclusive request must be serviced by the hierarchy that has the line in a dirty state. In an invalidation-based protocol, the read-exclusive request also leads to the invalidation of copies in caches with the line in clean state. The memory is responsible for supplying valid data if none of the caches have the line in dirty state; this case may either be detected by delaying the response from memory until all processor caches have had a chance to respond or by maintaining extra state (i.e, a single bit) at the memory to determine whether the memory copy is up-to-date.

In a directory protocol, directory state is logically maintained for each line of memory (typically the same granularity as the cache line). A given memory line may be in one of three possible states: *uncached*, *shared*, and *exclusive*. The uncached state corresponds to the only valid copy residing in memory. The shared state corresponds to both memory and one or more processor caches having a valid copy. Finally, the exclusive state corresponds to a single processor cache owning the valid copy. The protocol typically associates extra information with each line to identify the set of sharers or the owner processor. Requests are first sent to the home node where the given memory location resides. Read or write requests to a line in exclusive state are forwarded from the home to the processor cache with the exclusive copy. Other requests are satisfied at the home memory, with write requests potentially causing invalidate or update messages to be sent to other cache copies.

5.2 Mechanisms for Exploiting Relaxed Models

With the widening gap between computation and memory access speeds, the overlap of memory operations with other memory operations and computation within a given thread of execution has become an important factor in achieving high performance in both uniprocessor and multiprocessor systems. The effect of overlapping optimizations is to partially or fully hide the latency of memory operations. Such optimizations are especially important in multiprocessors due to the typically higher number of cache misses and larger

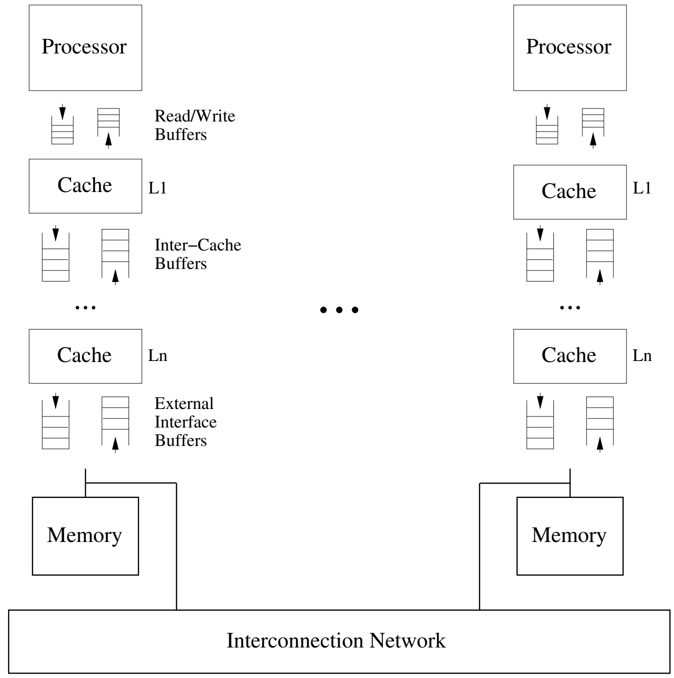


Figure 5.2: Typical architecture for distributed shared memory.

memory latencies. The performance potential for a multiprocessor implementation depends on two factors: (a) the overlap and reordering optimizations enabled by the memory model, and (b) the extent to which the implementation exploits these optimizations.

Figure 5.2 shows a typical distributed shared memory architecture. Read and write operations issued by the processor may be serviced at various levels of the memory hierarchy: by the processor's buffers or caches, by the memory, or by another processor's buffer or cache. Memory latency may be hidden by supporting multiple outstanding memory operations from each processor and servicing memory operation as early as possible. To attain the maximum benefit from these optimizations, it is important to (a) provide sufficient resources, often in the form of buffers, to support multiple outstanding memory operations, (b) reduce or eliminate serialization in issuing and servicing memory operations,[1] and (c) provide a balanced design with optimizations applied at all levels. The latter point is extremely important; for example, an aggressive processor that supports multiple outstanding operations also requires an aggressive cache and memory system.

This section describes architectural mechanisms for supporting aggressive overlapping optimizations, exposing a variety of techniques (many of which are derived from uniprocessors) along with the associated complexities of handling multiple outstanding operations. Additional mechanisms for appropriately limiting the scope of these optimizations to satisfy the constraints of specific multiprocessor memory models are discussed in the next section.

[1]Serialization in handling memory operations can limit the achievable number of outstanding operations (i.e., equal to approximately the time to service a typical operation divided by the serialization time).

5.2.1 Processor

The opportunity to overlap memory operations with other operations and computation from the same thread begins at the processor. To effectively hide the latency of memory accesses, the processor requires the ability to continue past pending operations to find other computation and memory accesses to execute. The extent of this ability is primarily determined by the technique used for scheduling instructions.

We categorize processors into two groups: in-order issue and out-of-order issue. An *in-order issue*, or statically scheduled, processor checks for data and structural hazards at decode time [HP90]. The issue of the instruction is delayed in case there is a hazard, leading to an in-order issue and execution of instructions. Typical sources of hazards are source data that is not yet computed (data hazard) or busy resources such as computation units or full buffers (structural hazards). In contrast to an in-order issue processor, an *out-of-order issue*, or dynamically scheduled, processor decouples the decoding of an instruction from its issue and execution, with the execution unit assuming the responsibility for detecting structural and data hazards [HP90]. The out-of-order issue and execution of instructions allows the processor to continue decoding and executing instructions even though some previous instructions may be delayed due to data dependences or busy functional units. While out-of-order issue designs are substantially more complex than in-order issue designs, they are becoming more common place in the new generation of commercial processors.

Consider the potential overlap between memory operations and computation with in-order issue processors. Write operations are typically issued into a write buffer and are overlapped with future computation since the processor does not inherently require a response for the write. Therefore, as long as there is sufficient space to buffer writes, the processor does not stall due to pending writes. Read operations are inherently different from write operations since future instructions may depend on the value returned by the read. Simple in-order issue processors often support *blocking reads* which stall the processor at the read operation until the return value arrives. To allow overlap of read operations with future instructions, the processor can support *non-blocking reads*, thus postponing the stall until the first instruction that uses the read return value. The possible overlap in typical code generated by current compilers is rather small since the use is often within a few instructions from the read instruction [GGH92], but may be enhanced by instruction schedulers that attempt to further separate the read instruction from its use [FJ94]. Therefore, the potential of overlap for non-blocking reads is highly sensitive to instruction scheduling.

Now consider the potential overlap among different memory operations given in-order issue processors. We divide overlap of memory operations into three categories: (i) overlap of a write followed by a read (write-read), (ii) overlap of multiple writes (write-write), and (iii) overlap of a read with a following read or write (read-read or read-write). The first two types of overlap effectively allow the processor to hide the latency of writes. The third type of overlap (read-read or read-write) allows the latency of reads to be partially hidden as well, but obviously requires a processor with non-blocking reads. Since there can be multiple outstanding memory operations, the memory system has the opportunity to overlap the service of multiple operations from the same thread. However, the number of outstanding operations at any given time may be small (e.g., zero or one) unless instructions are carefully scheduled.

Dynamic scheduling provides the opportunity for the processor to continue past an outstanding load and the instructions that are dependent on the return value. The use of dynamic scheduling to hide memory latency dates back to the IBM Stretch [Buc62] and IBM 360/91 [Tom67]. Figure 5.3 shows a simple example

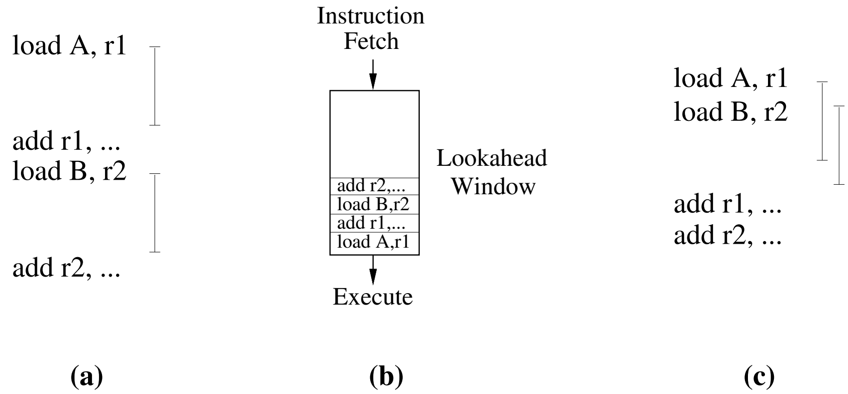


Figure 5.3: Example of out-of-order instruction issue.

of this type of overlap. The program segment in Figure 5.3(a) shows two load instructions each followed immediately by the use of the return value. Assume both load instructions are long latency cache misses. An in-order issue processor would issue the first load, stall at the use until the return value is back, issue the add, and then issue the second load. Even with non-blocking reads, no overlap is achieved between the two read operations. Figure 5.3(b) shows an abstract view of an out-of-order issue processor, modeled as a lookahead window of instructions where the processor can issue any ready instructions from the window. The processor starts by issuing the load to A. The add instruction that uses the return value of the load is not ready to be issued. However, the processor can proceed to issue the load to B while the load of A is outstanding. The add instructions are issued later when the relevant read data is returned to the processor. The effective overlap that is achieved is illustrated in Figure 5.3(c). The same type of overlap would be possible with an in-order issue processor (with non-blocking reads) if the original code is scheduled as in Figure 5.3(c), again illustrating that in-order issue processors are more sensitive to the initial instruction schedule.

To be effective in hiding large latencies, dynamic scheduling must be complemented with techniques such as *register renaming* [Kel75], *dynamic branch prediction* [LS84], and *speculative execution*. Register renaming alleviates write-after-read (WAR) and write-after-write (WAW) dependences that would otherwise delay the execution of future instructions. Dynamic branch prediction increases the lookahead capability of the processor while speculative execution allows the processor to execute instructions past unresolved branches.[2] Appendix L provides a few examples of how speculative execution can help in exploiting a relaxed model.

The following sections describe how the various components in the memory system can exploit the overlap among memory operations that is exposed by the processor.

5.2.2 Read and Write Buffers

Memory operations are typically buffered after they are issued by the processor. The main purpose for buffering is to avoid processor stalls when the memory system cannot accept operations as fast as the processor issues them. This section briefly discusses buffering optimizations such as forwarding, bypassing, merging, and out-of-order (non-FIFO) service, that are enabled by relaxed models. Section 5.3.5 describes

[2]Limited forms of speculative execution can also be supported in in-order issue processors, e.g., through boosting [SLH90].

how buffers can also play an important role in enforcing the appropriate order among memory operations as dictated by a memory model.

The processor issues two types of memory operations: a read operation along with an address, and a write operation along with an address and relevant data. In-order issue processors issue memory operations to the buffer in program order with valid address and data. Out-of-order issue processors may issue operations out of program order; however, some designs may issue place-holders for unready memory operations with unresolved addresses or values in order to allow the first level of buffering to keep track of program order among the operations. In the discussion below, we will assume two logical buffers, one for reads and another for writes. In an actual implementation, the two logical buffers may be physically merged or may be implemented as several separate buffers.

Consider the logical write buffer first. Assume write operations are issued to the write buffer in program order. A simple write buffer maintains first-in-first-out (FIFO) order in issuing the writes to the next level in the memory hierarchy. More aggressive buffers may allow non-FIFO issue to alleviate issue stalls. For example, the write at the head of the buffer may still be unresolved (i.e., given an out-of-order issue processor) or the next level in the memory hierarchy may temporarily not be able to accept operations to certain addresses (e.g., if there is already an outstanding operation to that line). In such cases, non-FIFO issue can improve the service rate for writes, thus reducing the chance of processor stalls due to buffer overflow.

Another common optimization for write buffers is *write merging*. The purpose of write merging is to coalesce writes to the same address or nearby addresses into a single write. In an aggressive implementation, each entry in the write buffer can maintain data for a set of consecutive addresses, typically corresponding to the line size for the next level in the memory hierarchy. When a write is issued to the write buffer, the write buffer first searches for an already existing entry for the given address range. In case such an entry already exists, the new data is merged in with the previous data for that entry. Otherwise, a new entry is allocated for the write. In this way, before an entry is retired, any writes issued to the buffer that match the entry are merged and coalesced into a single write with potentially more data. Of course, a side effect of merging and coalescing writes is that writes may be serviced in a different order relative the order they are issued to the buffer.

We now consider optimizations for reads. Read operations are more critical than write operations since the processor typically requires the return value before proceeding with subsequent computation. Therefore, it is beneficial to allow a read to be serviced before pending writes that have already been issued. This requires the read to *bypass* any writes in the write buffer. To ensure the correct value for the read, such bypassing is only allowed if the address of the read does not match any of the write addresses. This requires an associative match, but the match can be made conservative by comparing fewer address bits. The read is simply stalled if an address match is detected. An optimization in this case is to service the read right away by *forwarding* the value of the latest write in the buffer that matches its address; this latter match requires comparing all address bits. Both bypassing and forwarding can lead to the reordering of reads with respect to writes relative to the order in which they are issued by the processor. Finally, similar to writes, non-FIFO issue of reads from the buffer may also be beneficial.

The next section describes cache optimizations that exploit relaxed models. An aggressive cache design may reduce the benefits from optimizations such as bypassing and forwarding by increasing the service rate of buffers which increases the chances of an operation encountering an empty buffer; this effect is illustrated by our simulation results in Section 6.2.3 of the next chapter. Finally, the optimizations discussed above (i.e., non-FIFO issue, merging, bypassing, and forwarding) may also be beneficial at the lower levels in the memory hierarchy.

5.2.3 Caches and Intermediate Buffers

To exploit overlap among memory operations requires a *lockup-free cache* that is capable of servicing memory operations while previous operations are still outstanding [Kro81]. A standard *blocking* cache services processor requests one at a time. Therefore, only a single processor request is serviced during a long latency cache miss. It is reasonably straightforward to extend a blocking cache design to allow requests that hit in the cache to be serviced while there is an outstanding miss. To achieve higher performance, it is important to also allow multiple outstanding misses. Lockup-free caches are inherently more complex than standard blocking caches. However, since they play a critical role as a latency hiding mechanism, their use is becoming more widespread both in uniprocessor and multiprocessor designs. The first part of this section describes various design issues for lockup-free caches. The second part considers the design of the intermediate buffers between caches, and issues such as deadlock that arise in handling multiple messages in the incoming and outgoing paths.

Lockup-free Cache Design

We begin by identifying some of the requirements for a lockup-free cache. A general lockup-free cache must support multiple outstanding read and write requests. For writes, the data written by the write must be buffered while the write request is outstanding and must be correctly merged with the reply data. For reads, there needs to be a mechanism to forward the relevant portion of the returning data reply to the requesting unit or destination register.

A key component in the design of a lockup-free cache is the mechanism that is used for tracking outstanding requests. Most designs track outstanding requests in transaction buffers that are external to the cache [Kro81, SD91, FJ94], often referred to as MSHRs (miss information/status holding registers). An alternative approach is to track the requests directly in the cache, as in the remote access cache for the Stanford DASH [LLG+90, Len92]. Finally, some recent proposals advocate a hybrid approach [Lau94]. The main difference between these designs is in the limitations placed on the number and type of outstanding requests, and the hardware complexity of each design.

Below, we describe a straightforward implementation that keeps track of outstanding requests within the cache. For simplicity, we assume an invalidation-based coherence scheme using a simple write back cache with three states: invalid, clean, and dirty. To support multiple outstanding operations, an extra state is introduced per cache line to represent a *pending* (or transient) state. A cache line enters the pending state on a cache miss and remains in this state while the miss is outstanding. The line may not be replaced while it is in the pending state. Therefore, operations that conflict on the line must be delayed, allowing only a single outstanding request per cache line (or per set in a set-associate cache). While this restriction may cause some stalls in small caches, larger caches or set-associative caches are less likely to be affected by such conflicts.

As we will see below, there are numerous optimizations possible for accesses to a cache line that already has an outstanding request. The following is a list of such optimizations for operations to the same address or line: merge reads with outstanding requests as appropriate, read data from a line with a write outstanding and forward the value appropriately, write to a line with a read outstanding, and merge multiple writes to the same line. Some designs may also allow simultaneous outstanding operations to different addresses even when there is a mapping conflict in the cache.

A *write miss* results in the write data to be written into the cache line and the state of the line to be changed to pending. To allow for correct merging with the reply data, each cache line is augmented with fine grain *valid bits* (corresponding to the smallest size write operation that is supported). The state of the valid bits are only considered while the line is in the pending state. The appropriate valid bits are set during a write miss as the write data is written to the cache. A future write to the pending line may be merged by simply setting the corresponding valid bits and writing its data into the line. Once the write data reply arrives, it is merged appropriately with the data present in the cache line according to the valid bits. A future read to the pending line may fetch its value if the corresponding valid bits for that data segment are all set. Otherwise, the read must be delayed. As an optimization, an extra pending state may be used to signify a line that was in clean mode before a write miss. We refer to this as the *pending clean* state. A read to a line in pending clean state may fetch its value immediately regardless of the state of the valid bits. The line can exit the pending or pending clean state as soon as the exclusive or read-exclusive reply returns, even in protocols that provide an early response (i.e., before all invalidation-ack replies are back).

A *read miss* results in the data to be requested and the state of the cache line to be changed to pending. To allow appropriate routing of the return value when the read reply arrives, the destination register or unit and the actual word address to be read from the line are also recorded in a separate *read transaction buffer*. The size of this buffer places a limit on the number of outstanding reads. A future read to the pending line can be merged with the pending read if there is sufficient space in the transaction buffer to record the information for the new read. A future write to the pending line is treated in the same way as when the state is set to pending due to a previous write miss (in fact, these two cases are indistinguishable based on the state); the appropriate valid bits are set and the write data is stored in the cache line. The read data reply is placed in the cache on arrival. In addition, for each address/register pair in the transaction buffer that corresponds to this line, the specified word is selected and appropriately routed to its destination. If any of the valid bits are set, the read data reply is properly merged with the write data in the line, a write miss request is generated to fetch the line in exclusive mode, the valid bits remain set, and the line remains in the pending state (or is changed to pending clean state if this state is supported) until the write reply arrives. Even though delaying the servicing of a write miss until the read reply returns is conservative, it greatly simplifies the design.

While the line is in the pending or pending clean state, incoming read and read-exclusive requests (i.e., originating from lower level caches or the external interface) destined for the line must be either delayed or sent back with a negative acknowledgement reply.[3] Incoming invalidate requests may be readily serviced, however. An invalidate request has no effect if the line is in the pending state. An invalidate to a line in pending clean state changes the state to pending to reflect the fact that blocks with clear valid bits contain potentially stale data (and should therefore not be forwarded to future reads).

Another issue in lockup-free cache designs is the out-of-order arrival of replies. In a standard blocking

[3] Implementations that support a dirty-shared caching state may have to service certain requests in order to avoid deadlock or livelock. For example, consider P1 with a dirty-shared copy. Assume P1 and P2 both write to A, resulting in an exclusive request from P1 and a read-exclusive request from P2 to be sent to the home. If P2's request gets to the home first, it will be forwarded to P1 and may need to be serviced by P1 to avoid deadlock or livelock.

cache, there is only a single request outstanding. Therefore, an incoming reply can be easily matched with the request. A simple way to solve the problem in lockup-free caches is to require the reply to carry an address. This allows replies to be easily matched with requests, but implies that the address must be carried back by the reply even outside the cache hierarchy. To reduce the amount of extra bits carried by the reply, the external interface can assign a unique tag to each outstanding request to be carried back by the reply. For example, supporting eight outstanding requests would require only a three bit tag.

In the lockup-free cache hierarchy design described above, only the first level cache needs to support the mechanisms required for dealing with pending requests and appropriately merging requests to a pending line. The first level cache essentially acts as a filter for requests to the lower level caches, guaranteeing that a lower level cache will not receive further requests for a given line while the line has a miss pending.[4] Therefore, while the lower level caches still need to allow multiple outstanding requests, they need not support extra cache states, valid bits, or read transaction buffers. Allowing multiple outstanding requests is a simple extension over a blocking cache in the context of a multiprocessor, since even blocking caches need to simultaneously service processor requests and requests from the external interface.

Compared to a standard blocking cache, the above lockup-free cache design incurs some obvious additional costs. There is a separate transaction buffer that keeps track of address/register pairs for pending read requests. Furthermore, replies must carry either a unique tag or the full address. Finally, each cache line has one or two (in case the pending clean state is provided) extra states in addition to the fine grain valid bits. There is a way to reduce the extra cost per cache line, however. The main observation is that the valid bits are only useful for lines with outstanding requests. Based on this observation, Laudon has proposed a hybrid design that maintains the valid bits for outstanding cache lines in an external transaction buffer, called the *pending-write buffer*, while still maintaining the write data in the cache [Lau94]. The pending-write buffer can be built as a small fully-associative cache, with the tag pointing to the cache line for which the pending-write entry is allocated. While Laudon's design only maintains a single pending state (i.e., the optimization involving the pending clean state is not included), it is possible to extend the pending-write buffer to support the extra pending clean state and still maintain a single pending state per line within the cache. Of course, the size of the pending-write buffer places a limit on the number of outstanding write requests as compared to the original approach discussed above which theoretically allows a maximum of one outstanding request per cache line. However, the pending-write buffer can have more entries than a traditional MSHR since each entry is smaller given the write data is placed within the cache.

Overall, there are numerous possible techniques for supporting lockup-free caches with varying implementation and performance trade-offs. While there have been some detailed studies of these trade-offs in the context of uniprocessors [FJ94] and multiprocessors [Lau94], we expect more studies to appear as lockup-free caches become more widely used. Finally, multi-ported and interleaved caches are also becoming more prominent as a way of satisfying the bandwidth requirements of multiple instruction issue processors. Like lockup-free caches, these designs depend on the reordering of operations allowed by relaxed models to overlap simultaneous accesses to the cache.

[4] This assumes that the lower level caches do not have a larger cache line size relative to the higher caches. This simplification may also not apply to hierarchies that contain a write through cache.

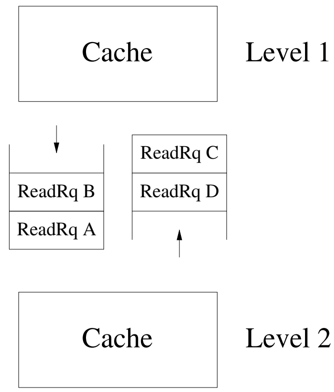


Figure 5.4: Example of buffer deadlock.

Inter-Cache Buffers

The buffers between caches allow each cache to operate more independently and are important for supporting multiple simultaneous requests within the hierarchy. However, the buffer design has to cope with the potential for deadlock in hierarchies with more than a single write back cache. Figure 5.4 shows a canonical example of how buffer deadlock may arise. Assume both levels use write back caches. There is an incoming and an outgoing FIFO buffer connecting the two caches, each with two entries. The figure shows two read requests in the outgoing queue (originating from the processor) going to the level 2 cache and two read requests in the incoming queue (originating from the external interface) going to the level 1 cache. Assume the states of the cache lines are such that the each cache can service the request destined to it. However, the level 2 cache requires an entry in the incoming buffer to generate the reply to the read of A, and the level 1 cache requires an entry in the outgoing buffer to generate the reply to the read of C. This circular dependency arises because servicing a request involves placing a reply in the buffer in the opposite direction and leads to deadlock due to a lack of buffering resources.

The buffer deadlock problems described above arise only in cache hierarchies with more than a single write back cache. Write through caches do not cause a buffer deadlock problem because incoming requests to higher level caches do not generate a reply, thus breaking the circular dependency.[5] Therefore, the simplest solution to the buffer deadlock problem is to use only a single level of write back cache in the cache hierarchy. Use of multiple levels of write back caches requires a more general solution, as described below.

Solving the buffer deadlock problem requires analyzing the flow of messages between the caches. Consider the message types shown in Table 5.1 for write back caches using an invalidation-based coherence scheme. In general, a given message may lead to the generation of zero, one, or two messages. Most incoming or outgoing requests are either serviced by a cache, leading to a reply in the opposite direction, or are passed along to the next level cache potentially as a different type of request. The incoming read-exclusive request is unique since it sometimes leads to a reply going in the opposite direction in addition to an invalidation request going to the next level cache. The generation of replies is one source of circular dependencies. Servicing an outgoing read or read-exclusive request may also cause two messages to be sent to the next level cache, one of them being an outgoing write-back request; these write-back messages are another potential source of

[5] We are assuming that incoming invalidate requests are acknowledged early (see Section 5.4.1), thus no outgoing invalidate-ack reply is generated within the cache hierarchy.

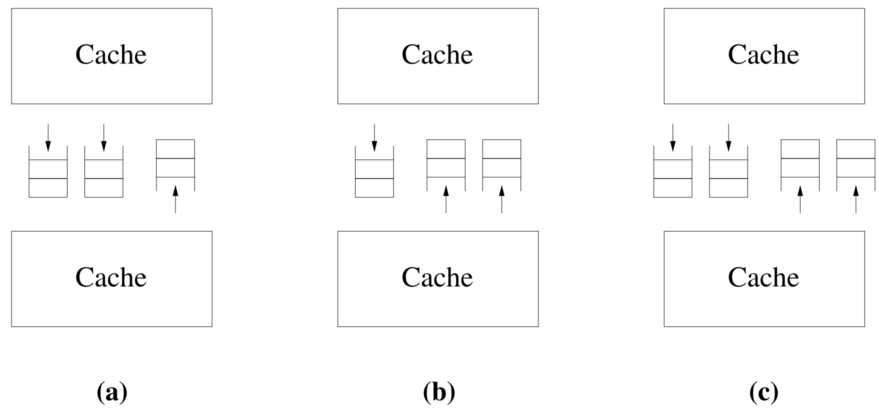


Figure 5.5: Alternative buffer organizations between caches.

buffer deadlock.[6]

There are a number of possible solutions to the buffer deadlock problem. We begin by describing a solution for designs with a single incoming and a single outgoing FIFO buffer between each cache [Lau94]. The solution is based on limiting the maximum number of requests that are injected into the hierarchy by the processor or the external interface and providing large enough buffers to absorb these messages. Given a limit of m outstanding requests from the processor and n incoming requests from the external interface, the pair of buffers between two adjacent write back caches are required to hold $2m + n$ messages plus one extra space per buffer to allow a waiting message to be processed. The reason each processor request requires two entries is because it may lead to a write-back message. Therefore, if we split the entries equally between the incoming and the outgoing buffers, each buffer must hold $\lceil(2m+n)/2\rceil+1$ messages.[7] Unfortunately, the above solution requires fairly large buffers to allow a reasonable number of outstanding requests within the hierarchy.[8]

An alternative way to solve the buffer deadlock problem is to disassociate the consumption of replies from the servicing of requests. One way to achieve this is to provide separate paths for request and reply messages. Figure 5.5 shows a number of alternative organizations. The organization in Figure 5.5(a) depicts separate outgoing buffers and a combined incoming buffer. The separate outgoing reply buffer allows incoming requests to be serviced regardless of the state of outgoing requests. This eliminates the circular dependencies that are present in single FIFO buffer designs. Figure 5.5(b) shows the dual organization where the incoming buffers are separated and the outgoing buffer is combined. Finally, Figure 5.5(c) shows the organization with separate buffers on both the incoming and the outgoing paths.

The fundamental requirement for avoiding deadlocks in the above scheme is to separate messages that potentially generate a "loop-around" message from the "loop-around" messages on both the incoming and outgoing paths. It turns out that only request messages can generate loop-around messages and the loop-around

[6]Outgoing write-back requests result from servicing an outgoing read or read-exclusive request (from a higher level outgoing queue) that misses in the cache but maps to a line that is already in the dirty state. In addition to passing the read or read-exclusive request to the lower level cache, the write-back request must also be eventually injected into the outgoing queue.

[7]The above assumes that all messages fit within a single buffer entry; the scheme can be easily extended to deal with variable size messages (e.g., messages with and without data).

[8]Laudon [Lau94] describes a second solution with more modest buffer requirements whereby an incoming (outgoing) buffer accepts messages from the downstream (upstream) buffer only when it is at least half empty. However, we uncovered scenarios where this algorithm would actually deadlock.

messages are always replies. Therefore, separation based on requests and replies satisfies the requirement above. However, this is not the only way of separating the messages. For example, not every request generates a loop-around message. Therefore, requests such as an incoming invalidate message may be placed on either queue in a design with separate incoming buffers. The same holds for outgoing requests such as a write-back in a design with separate outgoing buffers. This flexibility allows the choice of separating messages into different paths to be customized for the purpose of performance or maintaining certain orders among messages while still avoiding deadlocks.

In addition to providing separate incoming or outgoing paths, write-back requests must also be handled properly to avoid deadlock. One solution is to provide a minimum outgoing request buffer size of at least two entries. The cache does not service any outgoing requests from the higher level buffer unless there are two entries available in its outgoing request queue. In case a write-back is generated, the write-back message is placed into the queue along with the request that is passed down. Another solution is to buffer the write-back request in a separate buffer and to reinject it into the outgoing request queue when there is sufficient space available. The cache does not service requests from the higher-level outgoing queue while the write-back request is buffered. A write-back buffer of greater than one entry may be used to allow multiple miss request to be sent out before the write-backs need to be sent out. Similar issues arise in cache hierarchies that support replacement-hint requests (when the line that is replaced is clean), though a replacement-hint request does not carry any data along with it and may be dropped if necessary.

Providing separate paths for requests and replies on at least one of the incoming or outgoing directions has several advantages. First, a large number of outstanding messages can be handled with relatively small buffers. In fact, a single entry suffices for avoiding deadlock. Second, messages of different size are handled naturally, without needing to design for the worst case. Finally, as we will discuss shortly, separate paths on both the incoming and the outgoing direction provide advantages in the design of the external interface. One potential disadvantage of separate paths for requests and replies is the loss of certain ordering guarantees that would otherwise be preserved with a single incoming or outgoing path. Nevertheless, it is still possible to maintain sufficient ordering for correctly supporting a memory model by carefully choosing the messages that travel on each path.

5.2.4 External Interface

The external interface is responsible for exchanging messages between the cache hierarchy and the rest of the system. To overlap the latency of operations to memory and other processor caches, this interface must support multiple outstanding requests to and from the memory system. This section briefly discusses issues related to the buffer organization at this interface. As we will see, the design of the external interface is heavily influenced by the way messages are handled by the network and memory system, and also by the buffering scheme used between caches within the hierarchy.

First consider the issue of supporting multiple outstanding (outgoing) requests from the cache hierarchy. Almost all outgoing requests, except for write-back requests, expect an incoming reply message in response. The way such replies must be handled depends on how the network and memory system copes with potential deadlocks that may arise due to message handling (similar to issues that arise in inter-cache buffers). In designs with a large number of processors, a viable approach for avoiding deadlock within the network and memory system is to provide separate logical paths and buffers for requests and replies.[9] Both the DASH [LLG+90] and FLASH [KOH+94] systems use this technique; DASH uses physically separate request and reply networks while FLASH uses a single network with multiple virtual lanes to provide the separate paths. A key requirement for avoiding deadlock in the above scheme is that reply messages must be serviced (or consumed) unconditionally without depending on the service of any other messages (e.g., requiring a request message to be serviced first). This requirement affects the external interface since it is involved in accepting incoming replies from the network and memory system. Cache hierarchy designs with logically separate incoming request and reply paths allow the interface to unconditionally accept incoming replies. Designs with a combined request and reply path require extra support, however. A simple solution is to ensure sufficient buffer space for the reply message(s) before issuing an outgoing request that expects a reply. A static solution is to allow at most n outstanding requests and to provide sufficient buffering resources to accept n incoming replies. More dynamic solutions would reserve sufficient space on the incoming side before sending out an outgoing request.

Next consider the issues related to incoming requests. One of the key aspects with respect to incoming requests is whether the corresponding outgoing reply is expected by the outside interface potentially before any outgoing requests from the cache hierarchy are accepted. For example, a simple bus-based implementation that supports a single outstanding request on the bus requires the reply message to appear on the bus before any other messages are accepted. This is easy to support if the cache hierarchy and external interface provide logically separate paths for outgoing requests and replies. However, in designs with a single outgoing path, the external interface must provide sufficient buffering to absorb all outgoing requests buffered within the cache hierarchy (including write-back requests) in order to allow replies to bypass these requests on the way out. The above functionality may also be required in most split-transaction bus designs since there is typically a global limit on the maximum number of outstanding requests on the bus. Again, outgoing replies must be able to bypass outgoing requests within the cache hierarchy and external interface in order to avoid deadlock. Finally, to simplify the cache coherence protocol, even systems with more general interconnect networks (e.g., FLASH [KOH+94]) may require the cache hierarchy and external interface to provide the reply to an incoming request before any other outgoing messages are accepted.

In summary, cache hierarchy designs with a single incoming and single outgoing path require the external interface to provide sufficient buffering to absorb all expected incoming replies and outgoing requests. The size of both buffers is proportional to the number of outstanding requests that is allowed by each processor; the outgoing buffer must absorb both this maximum number of requests plus any write-back requests that may be induced by these requests. Designs with separate incoming paths alleviate the need for the incoming buffer in the external interface. Similarly, designs with separate outgoing paths alleviate the need for the outgoing buffer.

5.2.5 Network and Memory System

The network and memory system play an important role in overlapping the service of multiple memory operations from the same processor or different processors. To achieve such overlap, network and memory

[9] With a limited number of processors, it is possible to use large buffers to absorb the maximum number of messages that may be issued by all the processors. This approach is impractical in larger systems. An exception to this is a system such as Alewife [CKA91] that uses main memory as a large buffer to break deadlocks.

resources are often replicated and distributed across the system. Main memory is also often distributed among the nodes. This provides a level of interleaving in the memory system, and also provides each processor with higher bandwidth and lower latency to the memory local to its node. The memory within each node may also be interleaved or partitioned into multiple banks to support further overlap. High performance networks, such as two or three dimensional mesh networks, also enable the overlapping of memory operations by providing multiple paths and heavily pipelining the messages within each path. One of the consequences of multiple paths is that the network provides very few message ordering guarantees; designs that use static routing at best guarantee point-to-point order for messages within the same virtual lane.

Overall, the above optimizations lead to memory operations being serviced in an order that is potentially different from the order they are issued to the memory system and network. Furthermore, the presence of multiple paths in the network leads to inherently non-atomic coherence operations (e.g., updating a copy) across the multiple cached copies.

5.2.6 Summary on Exploiting Relaxed Models

This section described example optimizations that attempt to issue and service memory operations as quickly as possible at various levels of the memory hierarchy, exploiting operation reordering whenever necessary. Achieving high performance requires a balanced design that provides sufficient bandwidth at all levels of the memory hierarchy. Therefore, all components of the system, including the processor, cache hierarchy, network, and memory system, must be designed to support the desired level of overlap among memory operations. Such aggressive designs are naturally more complex. Furthermore, supporting multiple outstanding operations introduces various deadlock problems that must be addressed in such designs.

5.3 Maintaining the Correct Ordering Among Operations

The three most important issues from an implementation perspective are performance, correctness, and complexity. The previous section described examples of optimizations that boost performance, but lead to the reordering of memory operations that may violate the semantics of a given memory model. A key design challenge is to appropriately limit the scope of such optimizations for correctness while still achieving the full performance potential for a given model. At the same time, it is important to limit implementation complexity. Therefore, a designer must trade off the marginal performance gain from a given optimization versus the resulting increase in design complexity.

This section presents practical and efficient mechanisms for enforcing the correct ordering among memory operations. We mainly focus on issues relevant to tightly-coupled architectures based on scalable interconnection networks. Mechanisms relevant to systems with restricted networks and loosely coupled architectures are later discussed in Sections 5.5 and 5.6, respectively. As with performance optimizations, mechanisms for ensuring the correct ordering of memory operations are not just limited to the processor or the memory system, but affect the whole system. Furthermore, there is often an interdependence between the type of reordering optimizations exploited (along with deadlock-avoidance solutions) and the mechanisms used to enforce correct ordering.

We use the specifications developed in Chapter 4 to determine whether an implementation correctly implements a given memory consistency model. Because these specifications provide close to minimal constraints

define $\xrightarrow{spo}$**:** X $\xrightarrow{spo}$ Y if X and Y are to *different* locations and X $\xrightarrow{po}$ Y

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co'}$ Y

R $\xrightarrow{co'}$ W $\xrightarrow{co'}$ R

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes. **[Section 5.3.3]**

Condition 4.5: termination condition for writes; applies to *all write sub-operations*. **[Section 5.3.2]**

Condition 4.6: return value for read sub-operations. **[Section 5.3.2]**

Condition 4.7: atomicity of read-modify-write operations. **[Section 5.3.7]**

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W **[Section 5.3.3]**

coherence: W $\xrightarrow{co}$ W **[Section 5.3.2]**

multiprocessor dependence chain: one of **[Section 5.3.5]**

W $\xrightarrow{co'}$ R $\xrightarrow{po}$ R

RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure 5.6: Conditions for SC with reference to relevant implementation sections.

on memory ordering, most implementations conservatively satisfy a given specification by imposing stricter ordering constraints than required. We begin this section by relating the abstract events used in the specifications to actual events in an implementation. Sections 5.3.2 to 5.3.7 describe mechanisms and techniques for satisfying the various conditions imposed by a specification.[10] Figure 5.6 shows the sufficient system conditions for sequential consistency as presented in the previous chapter. Beside each of the main conditions, we have referenced the section that addresses the relevant implementation issues; the reach condition (which does not apply to SC) is discussed in Section 5.3.6. Section 5.4 describes a number of more sophisticated techniques that provide more efficient implementations by exploiting optimizations exposed by the aggressive specifications developed in Chapter 4.

5.3.1 Relating Abstract Events in the Specification to Actual Events in an Implementation

To determine the correctness of an implementation, we need to relate the abstract memory events and the corresponding execution order described in Chapter 4 (specifically, in Section 4.1.3) to physical events and orders in the implementation. The relationship between abstract and physical events is straightforward except for a few subtle issues related to the completion of write operations; the reason for this subtlety will become more clear in Section 5.3.4.

Figure 4.3 in the previous chapter pictorially (and informally) depicts the general abstraction for memory operations that the specifications are based upon. In this abstraction, each processing node conceptually has a complete copy of the shared memory along with a memory operation buffer that buffers that node's memory

[10]We assume the uniprocessor correctness condition (Condition 4.1 in Chapter 4) is trivially satisfied by all implementations. Appendix G describes a relaxation of this condition.

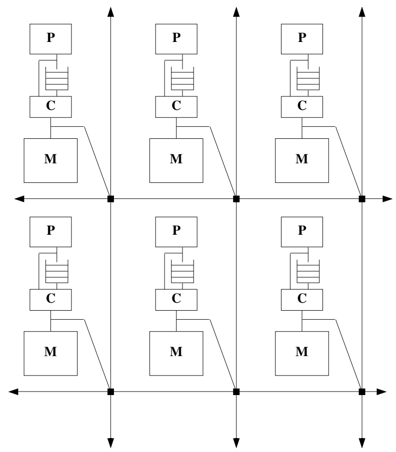


Figure 5.7: Typical scalable shared memory architecture.

operations. Reads are modeled as two sub-operation, $R_{init}(i)$ and $R(i)$ by P_i, and are serviced by either the local copy of memory or the local buffer. Writes are modeled as $n + 1$ sub-operations (where n is the number of processing nodes), consisting of a single initial sub-operation, $W_{init}(i)$ by P_i, and n sub-operations, $W(1), \ldots, W(n)$, corresponding to the write updating the values in each of the n memory copies. Since the main purpose for the R_{init}(i) and W_{init}(i) sub-operations is to capture the program order among conflicting operations, we do not attempt to relate them to physical events.

The mapping between abstract events and physical events depends heavily on the underlying implementation. Figure 5.7 shows the type of implementation we are assuming. For concreteness, we assume a simple directory-based coherence protocol similar to the protocols used in DASH [LLG+90] and FLASH [KOH+94]. The concept of the *owner* copy plays an important role in such coherence protocols. If the line is held exclusive by a cache, then it is considered to be the owner copy. Otherwise, the memory at the home node is considered to be the owner. For simplicity, we also assume the protocol maintains the coherence requirement for all writes; i.e., writes to the same location (or memory line) are serialized at the owner copy so that they complete in the same order with respect to all processors. Below we describe the path taken by a read or a write request before it is serviced.

Consider a read request. After the processor issues the read, the processor's caches and buffers attempt to service it. If this fails, the read is sent to the home node corresponding to its physical address and can be serviced there if a valid copy of the line exists in memory (i.e., the home node is the owner). Otherwise, the read is forwarded to the owner node that maintains an exclusive copy of the cache line. Due to infrequent transient cases in the protocol, a read request may need to traverse the above path more than once (e.g., due to negative-acknowledgement replies) before it is serviced. The *read request* is considered *complete* when its return value is bound. This completion event in the physical implementation corresponds to the read appearing in the execution order in our specification abstraction. Of course, in a distributed architecture such as the one shown in Figure 5.7, the processor detects the completion of the read only after it receives the reply message containing the return data.

Consider a write request now. A write request may also involve reading the cache line if the cache supports a write-allocate policy (e.g., consider a read-exclusive request); this can be effectively treated as a read request piggy-backed on top of the write. Below, we focus on the path that the write request traverses to invalidate or update stale copies of the cache line. Since the effect of a write request on multiple cache copies is inherently non-atomic, we model the write event as consisting of multiple completion events with respect to each of the n processors. A *write* is considered *complete* when its constituent events have occurred with respect to every processor.

Given a write W issued by P_i, W completes with respect to another processor P_j (i.e., $i \neq j$) at the time all older values (i.e., values from writes that are serialized before W) are eliminated from P_j's read path. Along with the coherence requirement, this implies that a write cannot complete with respect to any processor before it is serialized (at the current owner) with respect to other writes to the same location (or same line, since serialization typically takes place at the line granularity). Furthermore, a read by P_j that completes after a conflicting write W has completed with respect to P_j can no longer return an old or stale value. Finally, except for the issuing processor P_i, a read from another processor P_j cannot physically return the value of a write before the write is serialized and completes with respect to P_j. However, the issuing processor P_i is allowed to return the value of its own write before the write is serialized or completes with respect to itself.

Below, we enumerate the various cases for servicing a write. The points at which we consider a write to be complete with respect to a given processor are conservative; in some protocols, it may be possible to consider the write as having completed at an earlier point in time. A write request W by P_i can be serviced by its own processor's caches if an exclusive copy of the line exists within the hierarchy. The write is considered complete with respect to P_i at the time the exclusive copy is modified. The write is considered complete with respect to another processor P_j either when the exclusive copy is modified or when all previous writes to the same line that are serialized before W are complete with respect to P_j, whichever occurs later.[11] If the cache hierarchy cannot satisfy the write, the request is forwarded to the home node. The directory at the home maintains information on the location of other copies of the line. We enumerate the three possible cases below. The *first* case corresponds to the directory indicating that no other processor caches hold a copy of the line. In this case, the write W is considered complete with respect to a processor either when the copy at memory is modified[12] or when all previous writes to the same line that are serialized before W are complete with respect to P_j, whichever occurs later. The *second* case corresponds to the directory indicating that an exclusive copy of the line is held by P_k. In this case, the write request is simply forwarded to P_k's cache hierarchy. The write W is considered complete with respect to another processor P_j (i.e., not the same as P_k) either when the exclusive copy within P_k's hierarchy is modified or when all previous writes to the same line that are serialized before W are complete with respect to P_j, whichever occurs later. The write is considered complete with respect to P_k at the time all stale copies within its hierarchy are appropriately modified.

Finally, the *third* case for a write corresponds to the directory indicating that one or more cache hierarchies (other than P_i's hierarchy) are sharing clean copies of the line. In addition to modifying the memory copy, invalidate or update messages are sent to each of the relevant cache hierarchies to eliminate the stale copies. Within each cache hierarchy, the invalidate or update message traverses the hierarchy until all copies there are appropriately modified. The write W is considered complete with respect to P_j ($i \neq j$) at the time all copies within P_j's cache hierarchy are appropriately modified; if P_j's cache hierarchy does not maintain a copy of

[11] Because the state is distributed in the system, the information at the caches or at the directory may not exactly capture the actual state of the system. For example, even though a cache may indicate that there are no cache copies of a line, there may be pending invalidations from previous writes that are still in route to the "stale" copies. This explains the need for the complicated second clause that requires previous writes to have completed before a new write is considered complete with respect to a processor.

[12] Modification of a copy does not necessarily imply the updating of that copy. The modification may only comprise of changing the relevant state to signify a stale copy.

Conditions on $\xrightarrow{xo}$:

(a) the following conditions must be obeyed:

Condition 4.5: termination condition for writes (may apply to only a subset of write sub-operations).

Condition 4.6: return value for read sub-operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

coherence: W $\xrightarrow{co}$ W (may apply to only a subset of write sub-operations)

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure 5.8: Generic conditions for a cache coherence protocol.

the line, then the write completes with respect to P$_j$ $(i \neq j)$ either when the memory copy at the home node is modified or when all previous writes to the same line that are serialized before W are complete with respect to P$_j$, whichever occurs later. The write W is considered complete with respect to its issuing processor P$_i$ in the time interval *after* the completion with respect to P$_i$ of any writes to the same line that are serialized before W and *before* the completion with respect to P$_i$ of any writes to the same line that are serialized after W. In some protocols, a message is sent back from the home node to the issuing processor (i.e., an early read-exclusive or exclusive reply before invalidations are acknowledged) to signal that the write is serialized with respect to other writes to the same location; this message may be used to denote the completion of the write with respect to the issuing processor. Similar to read requests, transient cases may lead to a write request traversing the above paths more than once before it completes. In addition, the processor that issues the write request detects the completion of the write only after receiving the appropriate reply messages.

Given the above mapping between abstract and real events, the implementation is considered to be correct if the order among the real events is consistent with the order imposed by the specification among the abstract events. The above is a sufficient and not a necessary condition since an implementation is considered correct as long as it *appears* as if the specification is upheld (i.e., it provides the same results as the specification). However, given the aggressive nature of the specifications developed in Chapter 4, all the implementations we consider satisfy the sufficient condition.

5.3.2 Correctness Issues for Cache Coherence Protocols

This section describes the issues related to correctly implementing a cache coherence protocol. Figure 5.8 presents a formal set of conditions, based on the abstraction framework developed in Chapter 4, that a cache coherence protocol must satisfy. These conditions constitute a strict subset of, and are common across (except for a more selective termination or coherence requirement in some specifications), the model specifications presented in Chapter 4. Picking a specific set of common conditions to represent the specification for the cache coherence protocol can be somewhat arbitrary. We chose the common subset by excluding the following: conditions that restrict ordering based on the program order relation (which eliminates the initiation condition for writes, the uniprocessor dependence condition, and the multiprocessor dependence chains), the reach condition, and the atomicity condition for read-modify-write operations. We believe that this subset formally captures the intuitive correctness requirements for a cache coherence protocol. Condition 5.1 is a straightforward translation of the conditions in Figure 5.8 from abstract events to physical events based on the relationship described in the previous section.

Condition 5.1: Implementation Conditions for a Cache Coherence Protocol

The following conditions must be satisfied by the cache coherence protocol:

(a) Termination Condition for Writes: Every write issued by a processor eventually completes (i.e., within a finite time) with respect to all processors.

(b) Return Value for Read Operations: A read operation by P_i returns a value that satisfies the following conditions. Below, we assume the read and write operations are to the same address. If there is a write operation issued by P_i that has not yet completed with respect to P_i before the read completes, then the value returned by the read must be from the last such write that has been issued by P_i. Otherwise, the value returned by the read must be from the latest write (from any processor) that has completed with respect to P_i before the read completes. If there are no writes that satisfy either of the above two categories, then the read must return the initial value of the location.

(c) Coherence Requirement: Writes to the same address complete in the same order with respect to every processor.

Conditions (a) and (c) above may be restricted to only a subset of the write operations.

The termination and coherence requirements are especially relevant to systems that replicate data through caching. The *termination condition* requires an issued write to eventually appear in the completion order with respect to every processor. Furthermore, our specification framework implicitly requires each issued write to appear *at most once* in the completion order with respect to a given processor. The above constraints imply that each issued write appears *exactly once and only once* in the completion order with respect to each processor. The *coherence requirement* further requires that writes to the same address complete in the *same* order with respect to every processor. Finally, the *value condition* constrains the value that a read must return.

The remaining part of this section presents implementation techniques for satisfying the restrictions outlined by Condition 5.1 above. We begin by discussing implementation techniques for satisfying the value condition and coherence requirement. We next discuss the importance of supporting the coherence requirement and issues that arise if this condition is not supported for all writes. We finally describe techniques for satisfying the termination condition. The more subtle issues in correctly implementing a coherence protocol (e.g., transient cases, cross checking, maintaining point-to-point order, etc.) are described in Appendix K.

Supporting the Value Condition and Coherence Requirement

This section presents a few different techniques for supporting the value condition and coherence requirement. There are a number of subtle issues in a cache coherence protocol for correctly supporting the above two conditions that are covered in Appendix K.

The simplest way of supporting the coherence requirement is to serialize writes to the same location at the current owner. This serialization is used to ensure that writes to the same location complete in the same order with respect to every processor. In a typical invalidation-based protocol, the owner may be either the issuing processor's cache hierarchy, the memory at the home node, or another processor's cache; the read-exclusive or exclusive reply typically signals the issuing processor that its write is serialized. In a simple update-based protocol, the memory at the home node is the only possible owner; the update reply typically signals the serialization in this case.

One of the key aspects in supporting the value and coherence requirements is to correctly deal with outgoing and incoming requests at a processor that already has a pending write to that location. Consider an invalidation-based coherence protocol with a write buffer and a single level write-back cache per processor. We consider two different implementations below. In the *first* implementation, an exclusive or read-exclusive request is issued to the memory system and the pending write remains in the write buffer until the cache obtains exclusive ownership for the line. Consider subsequent operations to this cache line by the issuing processor while the original write is pending. A subsequent write is simply queued in the write buffer; the write can be merged or coalesced with the pending write. Since an exclusive copy of the line has already been requested, no new requests are generated for the memory system. A subsequent read must return the value of the latest write to the same address from the write buffer if any such write exists. In case of a match, the system may support forwarding or may simply stall the read until the conflicting writes are retired from the write buffer. Requests from other processors destined to this cache are handled in the usual way. The value condition is satisfied because pending writes are only visible to the issuing processor; this is because the value of the write is not placed in the cache until after the write is serialized at the owner. Appendix K describes the message orderings between incoming invalidate requests and read or read-exclusive replies that may be important to maintain especially within the cache hierarchy.

Note that from the time the write miss request is issued by P_i to the time the write is serialized with respect to other writes to the same location, conflicting writes from other processors that are serialized and complete with respect P_i are actually not visible to P_i. Therefore, reads from P_i may never return the value of such writes since they instead return the value of P_i's outstanding write.

The *second* implementation differs from the implementation above by allowing the write to be retired and the write data to be written to the cache while the write is still pending (similar to the lockup-free implementation described in Section 5.2.3). This implementation is trickier since we have to ensure that the value of a pending write is visible to its own processor and yet disallow other processors from observing the value placed in the cache. Furthermore, the data placed in the cache must be appropriately merged with the reply data that returns in response to the write (in the case of a read-exclusive reply). Mechanisms for achieving the above functionality were discussed in the context of lockup-free caches in Section 5.2.3. For instance, while the line is pending, incoming read or read-exclusive requests are either delayed or turned back with a negative acknowledge reply. Similarly, incoming invalidate requests are serviced by changing the state of the line from pending clean to pending, but do not get rid of the values in the cache line corresponding to pending writes. Designs such as DASH [LLG$^+$90] use a hybrid scheme that is a merge of the two techniques discussed above. In DASH, a write updates the write through first level cache immediately (similar to the second technique),[13] but the update to the second level write back cache is delayed by the write buffer until ownership is granted (as in the first technique).

Appendix M considers implementation issues for supporting the value condition and coherence requirement in update-based designs.

Importance of Upholding the Coherence Requirement

While the system-centric models discussed in this thesis impose the coherence requirement on all writes, the programmer-centric models relax this condition by imposing it on only a subset of the write operations (i.e., competing writes for PL1 and PL2, and non-loop and non-sync writes for PL3). While it may seem beneficial to exploit the above relaxation, this section describes the several advantages of providing coherence for all writes in the context of hardware shared-memory implementations.

The primary reason for providing coherence is to simplify the implementation. Consider the sufficient

[13] Dealing with a write through cache is simpler than dealing with write back caches because it is alright to invalidate or replace the line even before the write to the line is serialized. In addition, there are no incoming read or read-exclusive requests.

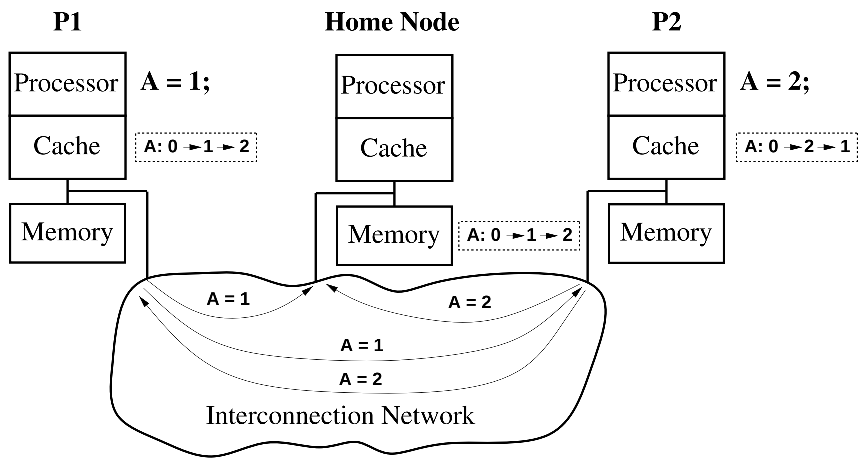


Figure 5.9: Updates without enforcing the coherence requirement.

conditions for satisfying the PL1 model shown in Figure 4.14 in the previous chapter, with the coherence requirement only imposed among competing writes. Nevertheless, it turns out that satisfying constraints such as the uniprocessor dependence condition (of the form W1 $\xrightarrow{po}$ W2) or multiprocessor dependence chain (of the form Wc $\xrightarrow{co'}$ Rc $\xrightarrow{spo}$ W) is easier if we guarantee coherence among all writes. For example, for the uniprocessor dependence condition, enforcing W1(i) $\xrightarrow{xo}$ W2(i) for all i is much simpler if the coherence requirement is upheld for the two writes. The same holds for the constraints imposed by the atomicity condition for read-modify-write operations (Condition 4.7). Finally, upholding the coherence requirement for all writes can be done quite efficiently in the context of hardware cache-coherent implementations. As we will see in Section 5.3.5, the simplest implementations for keeping track of outstanding invalidation or update requests inherently satisfy the coherence requirement for all writes by allowing only a single write to have outstanding coherence transactions.

The second reason for enforcing the coherence requirement for all writes is to provide a sensible semantics for programs that are *not* properly-labeled. Figure 5.9 illustrates an update-based coherence protocol which does not satisfy the coherence requirement for all writes. Assume location A is initialized to the value 0, P1 and P2 both maintain copies of this location, and the home for A resides on a third node. The example shows both P1 and P2 writing to location A. Each processor updates its own cache and sends update requests to other copies. If the coherence requirement is not upheld for the two writes, it is possible for the copy at P2 to transition from 0 to 2 to 1 while the copy at memory and at P1 transition from 0 to 1 to 2, allowing P1 and P2 to permanently observe different values for location A. The semantics is even more intractable if P2 replaces its cache copy, making the copy at memory visible to P2. In this scenario, the value of A visible to P2 transitions from 0 to 2 to 1 and back to 2, making it appear as if P2's write occurred *more than once*. Without maintaining a complete copy of memory at each node, it is virtually impossible to alleviate the above behavior caused by replacements unless the coherence requirement is upheld for all writes.

Migrating a process to a different processor can present similar problems. Consider migrating the process at P2 to either P1 or the home node. Again, the process can observe the effect of its write more than once, with the value transitioning from 0 to 2 to 1 while the process is on P2 and then transitioning back to 2 after it is migrated. In fact, indefinitely migrating a process among different processors can lead to the process observing the effect of a single write infinitely many times. Furthermore, even maintaining a full copy of memory does not solve the problems caused by migration. Therefore, supporting the coherence requirement on all writes is worthwhile unless designers are willing to limit an implementation to only executing properly-labeled programs. Even though this limitation is imposed by several software-supported shared-memory implementations (see Section 5.6), hardware implementations are often required to support a larger set of programming styles.

Supporting the Termination Condition

The termination condition requires a write that is issued to eventually complete with respect to all processors, thus ensuring all processors will eventually observe the effect of a given write.[14] For some specifications (e.g., RCsc, RCpc, and the sufficient conditions for PL1, PL2, and PL3), the termination condition is imposed on only a subset of the write operations (i.e., competing writes).[15] A number of software-supported shared-memory implementations (see Section 5.6) benefit from this relaxation. However, most hardware cache-coherent implementations end up inherently ensuring the termination condition for all write operations.

To satisfy the termination condition, an implementation must ensure that a write eventually affects all stale copies in the system. A designer must carefully consider places in a design where the servicing of a write is delayed either due to queueing or due to arbitration (e.g., if reads have priority over writes) to ensure that the write operations will eventually get serviced. Furthermore, the implementation cannot depend on the arrival of new operations to trigger the completion of previous write operations. For example, a merging write buffer that delays write operations in order to increase the window of opportunity for merging may need to periodically flush the write operations in the buffer. Similarly, incoming buffers within a cache hierarchy must ensure queued invalidate and update requests will eventually get serviced.

Appendix H describes a more aggressive form of the termination condition for the PL models, along with its implementation implications.

5.3.3 Supporting the Initiation and Uniprocessor Dependence Conditions

The initiation and uniprocessor dependence conditions interact closely to maintain the intuitive notion of uniprocessor data dependence; i.e., to ensure that the "effect" of a processor's conflicting read and write operations are consistent with program order. A simple way to ensure this is to require conflicting operations from the same processor to complete in program order. Thus, given conflicting operations from P_i, R $\xrightarrow{po}$ W implies R_{init}(i) $\xrightarrow{xo}$ W_{init}(i) and R(i) $\xrightarrow{xo}$ W(i), W1 $\xrightarrow{po}$ W2 implies $W1_{init}$(i) $\xrightarrow{xo}$ $W2_{init}$(i) and W1(j) $\xrightarrow{xo}$ W2(j) for all j,[16] and W $\xrightarrow{po}$ R implies W_{init}(i) $\xrightarrow{xo}$ R_{init}(i) and W(i) $\xrightarrow{xo}$ R(i). Note that the initiation and uniprocessor dependence conditions do not impose any ordering constraints on two reads to the same location.

Except for the IBM-370 model, the other models discussed in this thesis relax the above constraints in the case of W $\xrightarrow{po}$ R by only requiring W_{init}(i) $\xrightarrow{xo}$ R_{init}(i) (and not W(i) $\xrightarrow{xo}$ R(i)) to hold. This relaxation

[14] The termination condition implicitly depends on the uniprocessor correctness condition (Condition 4.1 in Chapter 4) which requires every write in the program to be issued (i.e., the W_{init}(i) sub-operation for every write must appear in the execution order).

[15] For some asynchronous algorithms, enforcing that all writes will eventually become visible to all processors may be important for convergence. This can be guaranteed by periodically flushing any writes whose completion has been delayed by the system.

[16] In an implementation that maintains the coherence requirement for all writes, it is sufficient to ensure W1(i) $\xrightarrow{xo}$ W2(i) since the coherence requirement will then ensure W1(j) $\xrightarrow{xo}$ W2(j) for all j.

allows a processor to read the value of its own write before the write completes with respect to it. Even though it may seem that allowing a read R that is after a conflicting write W in program order to complete before the write may cause a possible violation of uniprocessor data dependence, the initiation condition still requires $W_{init}(i) \xrightarrow{xo} R_{init}(i)$ and along with the value condition, the read is guaranteed to return the value of W (or a later conflicting write that is between W and R in program order). Therefore, the read still sees the effect of the last conflicting write that is before it in program order.

Techniques for satisfying the initiation and uniprocessor dependence conditions are well understood since similar conditions must be maintained in uniprocessors. A conservative way to satisfy the conditions is to delay issuing an operation until the previous conflicting operation in program order completes. Delaying an operation does not necessarily require the processor to be stalled; read and write buffers may be used for this purpose. Below, we discuss the implementation techniques for supporting the required orders in more detail.

Given a conflicting read followed by a write, the system must disallow the read to return the value of the write by ensuring the read completes before the write. Processors with blocking reads inherently satisfy the above condition. A more efficient technique may be to delay the completion of the write without stalling the processor. For example, the write may be placed in the write buffer, or may be retired into a lockup-free cache, but the sending of the exclusive or read-exclusive request may be delayed until the read completes.[17] In some protocols, it may be possible to issue the exclusive or read-exclusive request for the line before the read completes; this typically requires a guarantee that the read will complete without retries and that point-to-point order is maintained between the two requests. Such optimizations are more difficult to support in protocols that can force the read to retry (e.g., through a negative acknowledgement reply). For example, if the write is delayed in a write buffer, it is important for a read that is being retried to unconditionally bypass the write buffer (i.e., not forward the value of the write). A similar precaution applies to implementations that do an early retire of the write into the cache. Retrying the read from a lower level in the hierarchy can eliminate some of these issues. Finally, the presence of retries makes it difficult to correctly implement the optimization of issuing the exclusive request before the read completes since point-to-point order no longer inherently translates to the correct completion order.[18]

For two conflicting writes in program order, the simple approach is to delay the servicing of the second write until the first write completes with respect to the issuing processor. Again, this does not necessarily imply a processor stall since the second write may be delayed in a buffer or may be retired to the cache and merged when the reply for the first write arrives (as discussed in Section 5.2.3). Retiring the write to the cache is more complex for update-based schemes or write through invalidation-based schemes since the second write generates a new request to the memory system which may get out of order with respect to the first write's request either if point-to-point order is not maintained or if the protocol can force the first write to be retried. For protocols that enforce the coherence requirement for all writes, ensuring that the writes complete in program order with respect to the issuing processor automatically ensures that the writes complete in program order with respect to other processors. As discussed in Section 5.3.2, this is one of the ways in which enforcing the coherence requirement on all writes simplifies the implementation.

Finally, in the case of a write followed by a conflicting read in program order, only the IBM-370 model

[17] In protocols that do an early retiring of the write values into the cache, an exclusive or read-exclusive request caused by any write that lies in the same cache line as the read (i.e., even if the write and read do not conflict) may also need to be delayed until the read completes.

[18] For example, to maintain the correct order, the protocol may need to ensure that the write request will also be forced to retry if the previous read request to the same line is retried (by keeping additional state at the home node).

	P1		P2
a1:	A = 1;	*a2:*	A = 2;
b1:	u = A;	*b2:*	v = A;

Figure 5.10: Example showing interaction between the various conditions.

requires the read to complete after the write completes with respect to the issuing processor. The techniques described above for ensuring in-order completion of conflicting requests are applicable in this case as well.

Maintaining the initiation and uniprocessor dependence conditions has special implications for systems with out-of-order issue processors. This is because the processor can continue issuing memory operations even if a previous memory operation's address, or value in case of a write, has not yet been computed. The conservative approach is to treat an unknown address as potentially matching addresses of future reads and writes. In case of a read operation with an unknown address, the completion of future write operations is delayed until at least the read's address is resolved. In case of a write with an unresolved address, the completion of future read and write operations is delayed for the address to be resolved. Finally, in case of an unresolved value for a write, future read and write operations that conflict with the write are held back. Section 5.4.3 describes an optimization in the case of an unresolved write address, whereby a following read is allowed to be speculatively serviced and is retried (through a roll-back mechanism) if the write's address turns out to match that of the read.

Finally, as we will see in Section 5.3.5, enforcing the multiprocessor dependence chains in stricter models such as SC, PC, or TSO ends up inherently satisfying the initiation and uniprocessor dependence conditions. Therefore, no additional mechanisms are needed in such implementations.

5.3.4 Interaction between Value, Coherence, Initiation, and Uniprocessor Dependence Conditions

There is a subtle interaction between the value, coherence, initiation, and uniprocessor dependence conditions when all of them are to be satisfied by a set of operations.[19] Understanding this interaction provides intuition for the form of the value condition and the exclusion (in most models) of W $\xrightarrow{po}$ R from the uniprocessor dependence condition.

Consider the code segment in Figure 5.10. Both P1 and P2 write to location A and then read this location. Assume location A is initialized to 0, and each processor caches a clean copy. Let W1 and R1 denote the write and read on P1, and W2 and R2 the operations on P2. Consider W1 and W2 being issued at approximately the same time. Figure 5.11 depicts this scenario in two different implementations, both with write-back caches and an invalidation-based coherence scheme. A one word line size is assumed for simplicity. In both implementations, the write request leads to a miss on a clean line. The implementation in Figure 5.11(a) delays the write in the write buffer until the exclusive request is satisfied, while the implementation in Figure 5.11(b) allows the write to immediately retire into the cache with the corresponding cache line remaining in the pending state for the duration of the miss. These correspond to the two implementations discussed in Section 5.3.2 for

[19]This interaction was described first in a revision to the release consistency paper [GGH93b], and also in a later technical report [GAG+93].

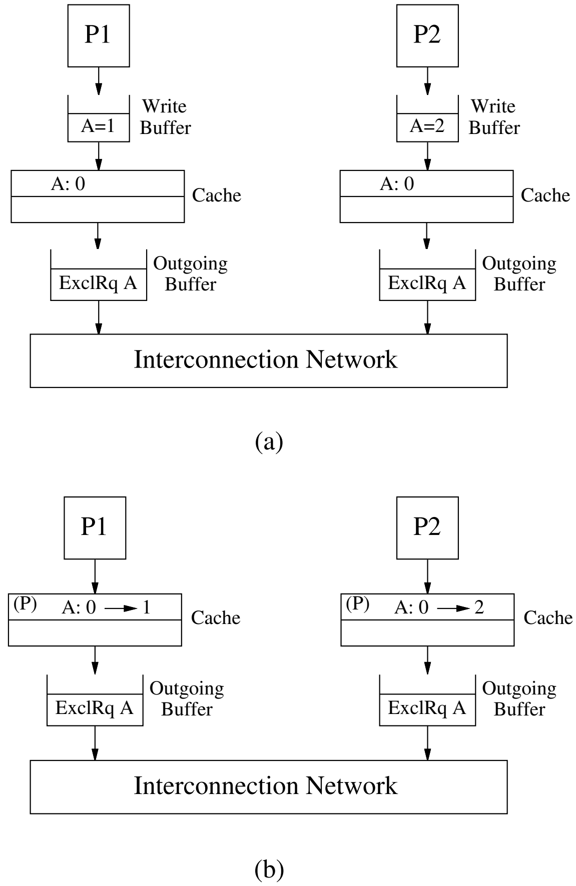


Figure 5.11: Simultaneous write operations.

supporting the coherence and value conditions. For the traditional notion of data dependences to be satisfied, the read on each processor must return the value of its own write on A or a write that completes after its own write. Thus, in our example code segment, it is safe for the reads to return the value of their own processor's write while the exclusive request is outstanding. The two key questions that arise are: (a) *at what point in time can we consider a write complete with respect to its issuing processor?* and (b) *can a subsequent read (e.g., R1 or R2) return the value of its own processor's write and complete before the write completes with respect to that processor?*

First consider the question about completion of a write with respect to the issuing processor. Without the coherence requirement, a write can be considered complete with respect to its own processor immediately after it issued. However, pinpointing this event is more subtle when the coherence requirement is imposed on a pair of writes. The coherence requirement requires writes to the same location to complete in the same order with respect to all processors: either W1(i) $\xrightarrow{xo}$ W2(i), or W2(i) $\xrightarrow{xo}$ W1(i), for all i. If the write is assumed to complete with respect to its own processor as soon as it is issued (i.e., before it is serialized with respect to other writes to the same location), then the completion events with respect to the issuing processors, W1(1) and W2(2), would be considered to occur before either W1(2) and W2(1) which are the completion events with respect to the other processor. This clearly violates the coherence requirement. The above example motivates why the completion event for a write with respect to its own processor is related to the reception of the exclusive or read-exclusive reply, which signals that the write has been serialized with respect to other writes to the same location.

Next consider the question about a read returning the value of its own processor's write before the write completes with respect to that processor. This optimization is uninteresting for models that do not impose the coherence requirement on a given write since, as we discussed above, the write can be considered complete with respect to its own processor immediately after it is issued. Therefore, the optimization only applies to models that impose the coherence requirement on the given write. The optimization can be supported in either implementation depicted in Figure 5.11; the implementation on the top can forward the value from the write buffer, while the implementation on the bottom can forward the value from the cache. The read-forwarding optimization is not safe for every memory model, however. Referring back to the example in Figure 5.10, consider a model with a strict uniprocessor dependence condition which would require W1(1) $\xrightarrow{xo}$ R1(1) and W2(2) $\xrightarrow{xo}$ R2(2). If the model also imposes the coherence requirement, either W1(2) $\xrightarrow{xo}$ W2(2) $\xrightarrow{xo}$ R2(2) or W2(1) $\xrightarrow{xo}$ W1(1) $\xrightarrow{xo}$ R1(1) must hold in any execution of the code. To ensure the above, the implementation must disallow a read to return the value of its own write until the write is serialized. The need to delay the read arises from the subtle interaction of the initiation, value, coherence, and uniprocessor dependence conditions: a read must return the value of its own processor's write or a later write in the execution order (initiation and value condition); the read cannot complete (i.e., return a value) until the conflicting write that precedes it in program order completes with respect to this processor (strict form of the uniprocessor dependence condition); and the write is considered complete with respect to its issuing processor after it has been serialized with respect to other writes to the same location (indirectly imposed by the coherence requirement).

Except for the IBM-370 model, all other models described in this thesis safely allow the read forwarding optimization by relaxing the uniprocessor dependence condition through the elimination of the W(i) $\xrightarrow{xo}$ R(i) requirement given W $\xrightarrow{po}$ R. Therefore, referring back to the example in Figure 5.10, execution orders such as R1(1) $\xrightarrow{xo}$ R2(2) $\xrightarrow{xo}$ W1(1) $\xrightarrow{xo}$ W1(2) $\xrightarrow{xo}$ W2(1) $\xrightarrow{xo}$ W2(2) are allowed. Note that the coherence requirement is still satisfied. Furthermore, even though the reads complete before the writes in the above execution order, the initiation and value conditions still maintain the traditional notion of data dependence by requiring R1 and R2 to return the value of their own processor's write (i.e., outcome of (u,v)=(1,2)). The subtle interaction among the conditions is broken by the following two things: (a) the relaxed form of the uniprocessor dependence condition allows a read to complete before a conflicting write that precedes it in program order, and (b) the value condition disassociates the visibility and completion events for writes, allowing a read to return the value of its own processor's write before the write completes. Switching between the strict and relaxed versions of the uniprocessor dependence condition affects the semantics of several of the models: the IBM-370 model depends on the strict version, while the TSO, PSO, PC, RCsc, RCpc, and RMO models depend on the relaxed version. For example, the program segments shown in Figure 2.14 in Chapter 2 distinguish the IBM-370 and the TSO (or PC) models based on whether a read is allowed to return the value of its processor's write early.

5.3.5 Supporting the Multiprocessor Dependence Chains

The multiprocessor dependence chains represent the most distinctive aspect of a memory model specification. Intuitively, these chains capture the orders imposed on pairs of conflicting operations based on the relative

define $\xrightarrow{spo}$**:** X $\xrightarrow{spo}$ Y if X and Y are to *different* locations and X $\xrightarrow{po}$ Y

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co'}$ Y
R $\xrightarrow{co'}$ W $\xrightarrow{co'}$ R

Conditions on $\xrightarrow{xo}$**:**

...
(b) given memory operations X and Y, if X and Y conflict
and X,Y are the first and last operations in one of
...
multiprocessor dependence chain: one of
W $\xrightarrow{co'}$ R $\xrightarrow{po}$ R
RW $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ RW
W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ {A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ }+ R
then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure 5.12: Multiprocessor dependence chains in the SC specification.

program and conflict orders of other operations. This section describes the relevant mechanisms for supporting multiprocessor dependence chains, including mechanisms for providing the ordering information to the hardware (e.g., through fences or operation labels), for keeping track of outstanding operations, and for enforcing the appropriate order among operations.[20]

Much of the discussion and examples in the following sections pertain to the SC and PL1 specifications which represent the strict and relaxed sides of the spectrum. For easier reference, Figures 5.12 and 5.13 show the isolated multiprocessor dependence chains for these two specifications.

Overview of Implementing Multiprocessor Dependence Chains

Multiprocessor dependence chains comprise of specific program and conflict orders that impose an execution order between the conflicting pair of operations at the beginning and end of the chain. In our specification notation, most of the relevant program order and conflict order arcs that constitute these chains are represented by the $\xrightarrow{spo}$ and $\xrightarrow{sco}$ relations, respectively. For implementation purposes, it is useful to separate the multiprocessor dependence chains into three different categories:

1. Chains of the form W $\xrightarrow{co}$ R $\xrightarrow{po}$ R or W $\xrightarrow{co}$ R $\xrightarrow{po}$ W consisting of three operations to the same location.
2. Chains that begin with a $\xrightarrow{po}$ *and* do not contain conflict orders of the form R $\xrightarrow{co}$ W $\xrightarrow{co}$ R.
3. Chains that begin with a $\xrightarrow{co}$ *or* contain conflict orders of the form R $\xrightarrow{co}$ W $\xrightarrow{co}$ R.

The above categories do not directly match the way the chains are separated in the model specifications. For example, the second chain representation in the specification for SC can fall in either the second or the third category above depending on the presence of conflict orders of the form R $\xrightarrow{co}$ W $\xrightarrow{co}$ R.

[20]Even though the reach relation appears as part of the multiprocessor dependence chains (through $\xrightarrow{spo}$) in system-centric models such as RCsc, we postpone the discussion of this relation until Section 5.3.6.

define $\xrightarrow{spo}$,$\xrightarrow{spo'}$**:**

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ Rc

Rc $\xrightarrow{po}$ Wc

Wc $\xrightarrow{po}$ Rc, to *different* locations

Wc $\xrightarrow{po}$ Wc

X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ RW

RW $\xrightarrow{po}$ Wc

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

Wc $\xrightarrow{co'}$ Rc

Rc $\xrightarrow{co'}$ Wc

Wc $\xrightarrow{co'}$ Wc

Rc $\xrightarrow{co'}$ Wc $\xrightarrow{co'}$ Rc

Conditions on $\xrightarrow{xo}$**:**

...

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

...

multiprocessor dependence chain: one of

Wc $\xrightarrow{co'}$ Rc $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ $\{$Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ $\}$* $\{$Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo}$ $\}$ RW

Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ $\{$Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ $\}$* $\{$Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo}$ $\}$ RW

RWc $\xrightarrow{spo'}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo'}$ $\}$+ RWc

Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo'}$ $\}$+ Rc

...

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure 5.13: Multiprocessor dependence chains in the PL1 specification.

P1	P2
a1: A = 1;	*a2:* u = A;
	b2: v = A;

(a) Category 1: outcome (u,v)=(1,0) disallowed

P1	P2
a1: A = 1;	*a2:* B = 1;
b1: u = B;	*b2:* v = A;

(b) Category 2: outcome (u,v)=(0,0) disallowed

P1	P2	P3
a1: A = 1;	*a2:* u = B;	*a3:* v = C;
b2: B = 1;	*b2:* C = 1;	*b3:* w = A;

(c) Category 2: outcome (u,v,w)=(1,1,0) disallowed

P1	P2	P3
a1: A = 1;	*a2:* u = A;	*a3:* v = B;
	b2: B = 1;	*b3:* w = A;

(d) Category 3: outcome (u,v,w)=(1,1,0) disallowed

P1	P2	P3	P4
a1: A = 1;	*a2:* u = A;	*a3:* B = 1;	*a4:* w = B;
	b2: v = B;		*b4:* x = A;

(e) Category 3: outcome (u,v,w,x)=(1,0,1,0) disallowed

Figure 5.14: Examples for the three categories of multiprocessor dependence chains.

Figure 5.14 provides examples of the three multiprocessor dependence chain categories. The disallowed outcomes assume the SC specification, with all locations initially 0. The first two categories of chains can be satisfied by enforcing specific program orders. The third category of chains also require enforcing multiple-copy atomicity on some writes.

The first category is of the form W $\xrightarrow{co}$ R $\xrightarrow{po}$ Y, where all three operations are to the same location. Assume R and Y are issued by P$_i$. W $\xrightarrow{co}$ R already implies W(i) $\xrightarrow{xo}$ R(i), i.e., the write completes with respect to P$_i$ before the read completes. If Y is a write, the required order, W(j) $\xrightarrow{xo}$ Y(j) for all j, is automatically enforced by the combination of the uniprocessor dependence condition and the coherence requirement (i.e., if the implementation enforces this requirement between W and Y). If Y is a read, the required order, W(i) $\xrightarrow{xo}$ Y(i), can be trivially satisfied by forcing R and Y to complete in program order with respect to P$_i$, i.e., R(i) $\xrightarrow{xo}$ Y(i). Referring to the example in Figure 5.14(a), this correspond to maintaining program order between the two reads on P2. In a relaxed model such as PL1, program order between reads to the same location needs to be enforced only if the first read is a competing read.

Figures 5.14(b) and (c) provide examples of the second category of chains. A simple way to satisfy such chains is to conservatively enforce the relevant program orders. Consider the SC specification, for example. For every A $\xrightarrow{spo}$ B in the chain, the implementation can ensure A(i) $\xrightarrow{xo}$ B(j) for all i,j by delaying any sub-operations of B until all sub-operation of A have completed. Given X and Y are the first and last operations in the chain, maintaining the strict execution order at every point in the chain enforces X(i) $\xrightarrow{xo}$ Y(j) for all i,j. This conservatively satisfies the specification requirement of X(i) $\xrightarrow{xo}$ Y(i) for all i. In fact, such an implementation satisfies conservative specification styles such as those presented for SC in Figure 4.6 of Chapter 4.

Satisfying the third category of chains requires extra ordering restrictions beyond maintaining program order since these chains expose the multiple-copy behavior in an implementation. Consider the example in

Figure 5.14(d). We will use the statement labels to uniquely identify operations; e.g., W_{a1} refers to the write on P1. Given an execution with the conflict orders shown in Figure 5.14(d), the SC specification requires $W_{a1}(3) \xrightarrow{xo} R_{b3}(3)$ to be enforced. The conflict orders already imply $W_{a1}(2) \xrightarrow{xo} R_{a2}(2)$ and $W_{b2}(3) \xrightarrow{xo} R_{a3}(3)$. Maintaining the program order among operations on P2 and P3 would enforce $R_{a2}(2) \xrightarrow{xo} W_{b2}(j)$ for all j and $R_{a3}(3) \xrightarrow{xo} R_{b3}(3)$. However, maintaining the program order is insufficient for disallowing the outcome (u,v,w)=(1,1,0) since it allows $R_{b3}(3) \xrightarrow{xo} W_{a1}(3)$.

The third category of chains can be enforced by ensuring that certain writes appear atomic with respect to multiple copies.[21] A simple mechanism to achieve this is to require a write to complete with respect to all other processors before allowing a conflicting read from another processor to return the value of this write. Referring to Figure 5.14(d), this restriction ensures $W_{a1}(i) \xrightarrow{xo} R_{a2}(2)$ for i=(2,3). Assuming program order is maintained on P2 and P3, the atomicity restriction enforces $W_{a1}(i) \xrightarrow{xo} R_{b3}(3)$ for i=(2,3), which conservatively satisfies the execution order constraints of the SC specification. In general, the atomicity constraint must be enforced for a write that starts a chain with a conflict order or a write that is part of a conflict order of the form $R \xrightarrow{co} W \xrightarrow{co} R$ (e.g., the write on P3 in Figure 5.14(e)). In practice, satisfying the SC specification requires enforcing this requirement for all writes, while the PL1 specification limits this requirement to competing writes only.

An alternative mechanism for enforcing atomicity is inspired by the conditions Dubois and Scheurich proposed for satisfying SC (see Sections 2.3 and 4.1.3), whereby the program order from a read to a following operation is enforced by delaying the latter operation for both the read to complete *and* for the write whose value is read to complete with respect to all other processors. Therefore, given $W \xrightarrow{co} R \xrightarrow{po} Y$ in a chain, the above requirement enforces $W(i) \xrightarrow{xo} Y(j)$ for all i,j except for i equal to W's issuing processor. Compared to the previous mechanism, the delay for the write W to complete simply occurs after, instead of before, the read R. Referring back to the example in Figure 5.14(d), the above enforces $W_{a1}(i) \xrightarrow{xo} W_{b2}(j)$ for all i,j except for i=1. Along with enforcing the program order on P3, this implies $W_{a1}(i) \xrightarrow{xo} R_{b3}(3)$ for i=(2,3), which conservatively satisfies the specification. Maintaining the program order conservatively past a read is required for the following reads that appear in third category chains: (i) the first read (R) in the chain if the chain begins with $W \xrightarrow{co} R$, or (ii) the second read (R2) in a conflict order of the form $R1 \xrightarrow{co} W \xrightarrow{co} R2$. In practice, the SC specification requires every $R \xrightarrow{po} Y$ to be enforced in the conservative way, while the PL1 specification requires this only for $Rc \xrightarrow{po} Yc$.

The following sections describe the various mechanisms for enforcing multiprocessor dependence chains in more detail. The techniques described maintain the execution orders at all intermediate points in the chain, and therefore do not exploit the aggressive form in which such chains are expressed in the specifications. Section 5.4.1 describes alternative techniques that satisfy the multiprocessor dependence chains more aggressively.

Providing Ordering Information to the Hardware

To aggressively support multiprocessor dependence chains, the information about significant program orders and write operations that must obey multiple-copy atomicity needs to be communicated to the underlying system. This information is implicit for models such as SC, TSO, and PC. In SC, for example, all program

[21] Section 4.4 in Chapter 4 described indirect ways of supporting multiple-copy atomicity by transforming specific reads into dummy read-modify-write operations. This section describes more direct methods for supporting this type of atomicity.

Table 5.2: How various models inherently convey ordering information.

Model	*Mechanism for Providing Information on* *Program Order*	*Multiple-Copy Atomicity*
SC, TSO, PC	-	-
IBM-370, PSO, Alpha, RMO, PowerPC	fence	-
WO, RCpc	label	-
RCsc, PL1, PL2, PL3	label	label
TSO+, PSO+	fence	-
PC+, PowerPC+	fence	label
RCpc+	label, fence	label

orders are significant and every write must obey multiple-copy atomicity. For most relaxed models, however, explicit information that is provided through operation labels or fences must somehow be explicitly communicated to the hardware. Table 5.2 shows how such information is inherently provided by various models (models such as TSO+ are extensions that were defined in Section 4.4).

There are several ways to communicate the information inherently provided by a model to the underlying hardware:

- The information may be conveyed in the form of an operation *label* encoded either in the *type* or the *address* of a memory instruction.
- The information may be conveyed through additional instructions, for example explicit *fence* instructions.
- The default implicit ordering or implicit operation labels may be altered through special *mode* instructions.

The above mechanisms may be used in a hybrid manner. Furthermore, the way information is conveyed to the hardware does not necessarily have to be the same as the way the information is inherently conveyed by a memory model. For example, a program written for the PL1 memory model conveys information through operation labels, yet the program may be executed on hardware that supports explicit fence instructions. This requires the information provided by the labels to be transformed to explicit fences, similar to the way a program is ported to different models (see Section 4.4). There may be some loss of information in such a transformation, resulting in more conservative ordering constraints.

Piggybacking ordering information on top of the instruction type or address has the advantage of incurring no additional instruction bandwidth. For example, consider the PL1 model which inherently distinguishes between competing and non-competing operations. The first option is to encode the information in the operation type by providing two flavors for every read and write memory instruction. Some operations, such as special synchronization operations, may be represented with a single flavor corresponding to the conservative label to reduce the number of extra instructions. Nonetheless, this approach is only feasible if the architecture provides sufficient additional opcode space. The second option is to encode the information in the effective virtual or physical address for a memory operation (using a double mapping technique), thus alleviating the need for additional instruction flavors.[22] Similar to the address-based approach discussed in Section 3.5.2, the main disadvantage of the latter option is that the ordering information is not known until after the effective virtual or physical address is computed. This can limit some desirable optimizations. For example, given two reads in program order, the hardware may have to delay issuing the second read until the address of the first read has been computed since it would otherwise not know whether the first read is a competing operation. In contrast, encoding the information as extra instruction flavors reveals the operation labels as soon as the instruction is fetched. However, as mentioned above, using extra instruction flavors may not be practical in most architectures due to opcode space limitations.

Explicit fence instructions are an alternative to operation labels. The most common use of *fence* instructions is to communicate the significance of specific program orders among instructions that occur before and after the fence. One way to classify a fence instruction is by the set of previous operations and the set of future operations that are related by the fence [GLL+90]. For example, the set of future operations may be: (a) all future read and write operations (*full fence*); (b) all future write operations (*write fence*), (c) all future read operations (*read fence*), and (d) only the operation immediately following the fence (*immediate fence*). Likewise, the set of previous operations may be a combination of specific read and write operations that precede the fence in program order. For example, for supporting PL1, an immediate fence may be used to capture the program order from previous read and write operations to a competing write. A more conservative way to convey this program order is to use a write fence before the competing write, which has the side effect of also imposing an order to the writes following the competing write; the use of conservative fences obviously leads to a loss of information. In implementations that do not directly support explicit fence instructions, achieving the functionality of a fence may require issuing a sequence of other instructions that end up enforcing the desired operation ordering.[23]

The use of fences as described above is not suitable for distinguishing write operations that must appear atomic with respect to multiple copies. There are several alternatives, however. The simplest is to provide atomic behavior for all writes if a model requires it for any write. Alternatively, the third category chains can be supported through the second technique described earlier in this section, whereby for some reads, the program order from the read to a following operation is enforced by delaying the latter operation for both the read to complete and for the write whose value is read to complete with respect to all processors; two flavors of fences may be used, with one imposing the more conservative ordering. Finally, a different mechanism may be used to identify the writes that must appear atomic. Explicit fence instructions can theoretically be used to directly provide equivalent information to operation labels. For example, a fence instruction may be used to logically identify the proceeding memory operation as a competing operation.[24] However, we are not aware of any systems that use fence instructions in this manner.

The primary disadvantage of fence instructions is the extra cost of the additional instructions especially in programs that require frequent use of fences. As mentioned in Section 4.4.1, porting a program written

[22]As an example of double mapping, consider two virtual addresses that only differ in the high order bit being mapped to two physical addresses that also only differ in the high order bit. This allows the hardware to treat the high order bit in the physical address as a "flavor" bit to distinguish competing and non-competing operations. Of course, the size of both the virtual and physical address spaces is effectively reduced.

[23]For example, delaying for previous writes to complete may require issuing a set number of dummy writes to flush previous writes out of a write buffer.

[24]Since a thread can be interrupted between the fence and the actual memory instruction, the implementation may need to save and restore the presence of an unmatched fence on context switches (e.g., just like a condition code).

for SC to an implementation with fences may require a fence instruction to be inserted between every pair of shared memory operations; if we assume memory operations constitute 50% of all instructions in the original program, the additional fence instructions would constitute a 1.5 times increase in the total number of instructions.

To alleviate the inefficiency of explicit fence instruction, an implementation may provide multiple *modes* corresponding to different default orderings, ideally with the ability to switch among modes through user-level instructions.[25] For example, an implementation that supports the RMO memory model through explicit fence instructions may provide modes for supporting SC, TSO, and PSO as well, where implicit fences are imposed appropriately by the implementation to alleviate the need for inserting explicit fence instructions. The above approach can be quite efficient for programs that do not require mode changes frequently.

Explicit modes can also be used to communicate default labels for operations that follow the explicit mode instruction. For example, a couple of mode instructions may be used to switch between competing and non-competing default labels for implementing a model such as PL1. This option has the advantage of not requiring the extra instructions flavors that would be required to explicitly specify labels on a per operation basis.

Operation labels or fences may be passed down by the processor to various levels of the memory system. In many cases, this information is consumed within the processor cache hierarchy. However, information such as whether a write must appear atomic may need to be passed all the way down to the serialization point at the home node in some designs.

Keeping Track of Outstanding Memory Operations

Enforcing the appropriate program order and multiple-copy atomicity constraints requires mechanisms to keep track of outstanding memory operations. The following two sections describe techniques for tracking the completion of individual memory operations as well as the completion of specific groups of memory operations.

Completion of a Single Memory Operation The following describes mechanisms for detecting the completion of a given memory operation. For concreteness, we will assume an invalidation-based protocol with single-level write back caches. Detecting the completion of a read operation is trivial since it is signaled by the read reply. Therefore, the rest of this section focuses on write operations.

A write that misses in its processor's cache generates a read-exclusive or exclusive request to the memory system. In response, the processor expects two logical replies from the memory system: (i) a read-exclusive or exclusive reply (depending on whether the processor has the up-to-date data for the line) that signifies exclusive *ownership*, and (ii) a final acknowledgement reply to signify the *completion* of the write with respect to other processors. Detecting the completion of a write with respect to other processors may in turn require gathering invalidation acknowledgement messages from cached copies.

There are three important protocol design issues with respect to writes:

- when (i.e., how early) an acknowledgement reply can be sent back in response to an invalidation request,

[25] The mode becomes part of the process state and must be saved/restored on context switches.

- whether invalidation acknowledgements are gathered at the home node or at the requesting node, and
- whether the read-exclusive or exclusive reply may be sent to the requesting node before the write is actually complete with respect to other processors (as a means to reduce the latency to the first response), or whether this reply must always be logically piggybacked with the final acknowledgement reply (delaying the response until the write is complete).

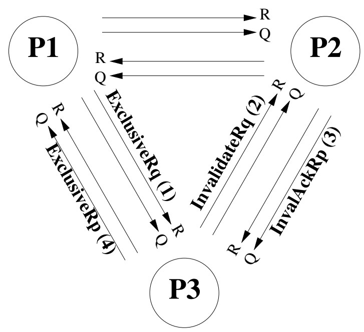
The above issues are primarily relevant for writes to lines with shared cache copies. For example, a write to a line that is uncached or cached in exclusive mode at another node does not generate any explicit invalidation messages; therefore, there are no invalidation acknowledgements and no latency benefit in separating the read-exclusive or exclusive reply and the final acknowledgement into two messages.

First consider the option of early invalidation acknowledgements. For simplicity, we have been assuming that an invalidation acknowledgement reply is sent back by a cache hierarchy after all stale copies in the hierarchy are eliminated. However, virtually all practical designs send back the acknowledgement as soon as the invalidation request is placed in the cache hierarchy's incoming queue. The above optimization clearly changes the semantics of the acknowledgement: the early acknowledgement only signals that the write is *committed*, but not necessarily completed, with respect to the target cache. Nevertheless, as we will discuss in Section 5.4.1, the multiprocessor dependence chains and other memory model ordering constraints imposed by a memory model can still be enforced in the same way as long as a few ordering constraints are imposed among incoming writes after they are committed. For the purpose of this section, we continue to assume the conservative approach.

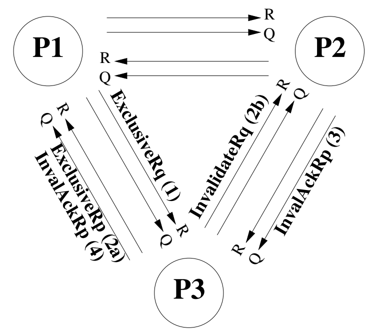
Figure 5.15 illustrates some possible options with respect to where invalidation acknowledgements are gathered and whether an exclusive reply is sent back possibly before the write is complete. Figure 5.15(a) shows the option of gathering invalidation acknowledgements at the home along with a *delayed* exclusive reply. In the scenario shown, P1 is the write requester that already has a clean copy, P2 also maintains a clean copy of the line, and P3 is the home node. The figure depicts the logical request and reply paths in the network as "R" and "Q", respectively; the number beside each message depicts the sequence in time. Because the exclusive reply is delayed until invalidations are complete, this reply implicitly signals completion of the write.[26] Figure 5.15(b) shows the scenario where the exclusive reply is sent back as soon as the write is serialized at the home. The *eager* exclusive reply reduces the latency for the first reply to the requester. Nevertheless, a final invalidation acknowledgement reply is still sent later to notify the requesting processor that all invalidations sent on behalf of this write are complete; in cases where no invalidations are generated from the home, the final invalidation acknowledgement is implicitly piggybacked with the exclusive reply. The base cache coherence protocol for the Stanford FLASH [KOH$^+$94] supports both of the above modes of operation.[27] Note that the eager reply optimization is only useful for the more relaxed models, especially those that allow multiple outstanding write operations; a strict model such as SC requires the processor to wait for the write to complete almost immediately anyway.

[26] In designs that use early invalidation or update acknowledgements, the exclusive reply is considered delayed as long as it signals the fact that the write is *committed* with respect to all processors even though the write may not be complete yet.

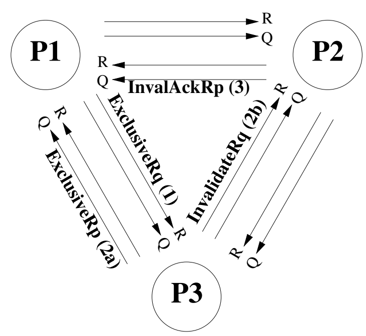
[27] There is a subtle protocol design issue with the above two options since an incoming reply to the home (i.e., invalidation acknowledgement reply) may need to generate a reply to the requester (i.e., either a delayed exclusive or a final acknowledgement reply). This violates the request-reply convention used in the protocol to avoid deadlock among network messages. The FLASH protocol deals with this by reserving sufficient resources upon reception of a write request at the home to later allow the sending of the reply to the requester to be delayed in case the outgoing reply network queue at the home is full.



(a) invalidation−ack to home, delayed exclusive reply



(b) invalidation−ack to home, eager exclusive reply



(c) invalidation−ack to requester

Figure 5.15: Protocol options for a write operation.

Figure 5.15(c) shows the alternative of gathering invalidation acknowledgements at the requesting node, which is the option used in the Stanford DASH [LLG+90]. The invalidation acknowledgement from P2 is directly forwarded to P1 instead of going through the home node, thus reducing the latency for notifying the requester about the write completion. In this scenario, there is no distinction between a delayed versus eager exclusive reply at the level of network messages; the requester node can support both modes by either forwarding the exclusive reply to the processor immediately (eager) or gathering both the exclusive and the acknowledgement replies before responding to the processor (delayed).

Detecting completion of writes requires maintaining a count of pending invalidation acknowledgements for *each* outstanding write. The *invalidation-ack count* may reside either at the requester node or at the home node, depending on where acknowledgements are gathered. The number of expected acknowledgements is determined by the number of sharers for the line that is recorded at the home directory. In the FLASH design, the invalidation count is kept within the directory entry associated with each memory line, allowing the directory state and the count to be accessed in the same lookup. In the DASH design, the requester is responsible for gathering acknowledgements and each requester maintains a limited number of counters that are allocated to outstanding writes and accessed through an associative lookup. The number of sharers is communicated from the home to the requester node as part of the exclusive reply message (refer to Figure 5.15(c)); since the exclusive reply and the invalidation acknowledgements may traverse different paths in the network, the requester must correctly deal with the reception of acknowledgements before the expected number of acknowledgements is known.

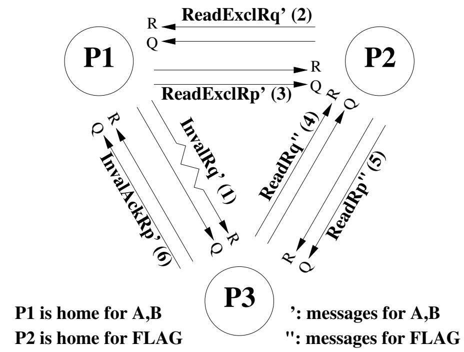
The choices of where to gather acknowledgements and whether to support eager or delayed exclusive replies lead to trade-offs in complexity and performance. For example, gathering invalidation acknowledgements at the requester can reduce the latency to detect the completion of a write (compare Figure 5.15(a) or (b) with Figure 5.15(c)), which can be especially beneficial for strict models such as SC that cannot hide the write latency. At the same time, gathering acknowledgements at the home can simplify the design. For example, as we will discuss below, blocking the service of requests (e.g., by forcing a retry) to lines with pending invalidations eliminates several corner cases in a coherence protocol. Gathering acknowledgements at the home makes it easy to support this blocking functionality since all cache misses must first visit the home node and the information on lines with pending invalidations is readily available at the home.

Similar trade-offs arise for the choice between delayed or eager exclusive replies. Most commercial processors expect only a single reply to a write that signals both ownership and completion (and also provides the data for the line in case of a read-exclusive reply). The advantage of the eager exclusive reply is that the processor or cache resources (e.g., write buffer entry) that are committed for pending writes are relinquished earlier. However, the design becomes more challenging since the processor may assume the write is complete as soon as the exclusive reply arrives. For correctness, the design must provide extra functionality external to the processor to determine the actual completion of writes by keeping track of the follow-on acknowledgements; this functionality is discussed in the next section.

Another key design issue that affects designs with eager exclusive replies is whether future read or write requests from other processors can be serviced while a line has pending invalidations or updates due to a previous write. By far the simplest solution is to disallow such requests from being serviced (e.g., either through nacking or buffering). Protocols with delayed exclusive replies inherently satisfy the above since the requester with the pending write does not receive ownership for the line until all invalidations are

	P1		P2		P3
a1:	A = 1;	*a2:*	B = 1;	*a3:*	while (**FLAG** == 0);
b1:	… …	*b2:*	**FLAG** = 1;	*b3:*	u = B;

(a)



(b)

Figure 5.16: Subtle interaction caused by eager exclusive replies.

acknowledged and neither the home nor this requester can service new requests during the time the write is outstanding. The same is not necessarily true for protocols that support eager exclusive replies.

Figure 5.16 shows an example scenario with eager exclusive replies where servicing a *write* while there are invalidations pending for a line can lead to complications. Consider the program segment in Figure 5.16(a), with operations to Flag labeled as competing under the PL1 model. A correct implementation of PL1 must disallow the outcome (u=0). However, as we will show below, guaranteeing this requires extra support in designs with eager exclusive replies. Assume the following conditions for the example in the figure: all locations are initialized to zero, A and B are co-located within the same memory line, P1 is the home for A/B, P2 is the home for FLAG, and P3 initially maintains a copy of line A/B. Consider a design with eager exclusive replies whereby the protocol allows a future write to be serviced even though there are pending invalidations for a line. Figure 5.16(b) depicts the sequence of network messages corresponding to the scenario described below. P1 attempts to write to A, leading to an invalidation request to be sent to P3's copy. Assume the invalidation request is delayed on its path to P3, for example, due to previous messages that are congesting this path. With eager exclusive replies, P1 obtains the line in exclusive mode before the invalidation is acknowledged. Meanwhile, P2's write to B generates a read-exclusive request which is sent to P1. Since P1's cache already has ownership of line A/B, it can surrender ownership to P2 even though invalidations are still pending for the line. At this point, if P2 assumes its write to B is complete, it can proceed with its write to Flag. Therefore, it is possible for P3 to read the new value for Flag and proceed to read its own stale copy of B with the value 0 all before the invalidation request from P1 reaches it. This execution clearly violates the PL1 model.

The problem in the above example arises because P2 receives a line in dirty state while there are invalidations pending to that line due to a previous write, and P2 is neither aware of these invalidation nor able to detect their completion. Consequently, in a design with eager exclusive replies, a write request to a block with pending invalidations must be handled in one of two ways: (a) the write must be delayed (by buffering or forcing retry) until the invalidations are complete, or (b) if the write is serviced, the requesting processor must be notified about the pending invalidations and must be later notified of their completion. The second option leads to a more complex design since the system must maintain pointers to the new requesters and notify them when the pending invalidations are complete. The first option also has the added benefit of trivially satisfying the serialization of writes that is necessary for supporting cache coherence (see Section 5.3.2). Supporting the first option is simple in designs that gather invalidation acknowledgements at the home since the home can easily deny service to requests to lines with pending invalidations. For designs that gather acknowledgements at the requester, such as in DASH, it is the requester awaiting for the invalidation acknowledgements that must nack new requests; it is also important to delay the writing back of the line to the home node while invalidations are pending since this could otherwise expose a window for a future request to be serviced at the home even though there are pending invalidations. The above functionality is supported by the remote-access-cache (RAC) in DASH [LLG$^+$90].[28]

Delaying the service of *read* requests to a line with pending invalidations also simplifies the design, especially for supporting the third category of multiprocessor chains where we must maintain the illusion of multiple-copy atomicity.[29] At the other extreme, aggressive designs that attempt to allow both read and write requests to be serviced while a line has pending invalidations can get quite complex. For example, consider a write W1 by P1 with pending invalidations, followed by a read by P2 that obtains a cached copy of the line, followed by a write W2 by P3 to the line. To service W2, we not only have to notify P3 of when W1 completes, but we also need to invalidate the copy at P2 and notify P3 of the completion of this invalidation that is caused by its own write.

Update-based protocols are also greatly simplified if we delay the service of write requests to a line that has pending updates. Consider a write W1 by P1 with pending updates followed by a write W2 by P2 to the same line. Unlike invalidation-based protocols, W1 does not eliminate the copies, and therefore W2 must also send update requests to these copies. Allowing simultaneous pending updates from multiple writes to the same line gets complicated because we must notify each requester on the completion of its updates. Thus, an aggressive design would require a counter per outstanding write in addition to sufficient tagging of the updates and acknowledgements to identify the requesting processor (and appropriate counter). This is more feasible in a DASH-like design where acknowledgements are gathered at the requesting node since acknowledgement messages are already forwarded to the appropriate node and counter. Another issue in update-based protocols is that it is difficult to disallow a read from observing the new value of a write that has outstanding updates; the protocol inherently updates (i.e. provides the new value to) copies in a non-atomic manner. This makes it more difficult to support the illusion of multiple-copy atomicity for writes in update-based protocols.

Completion of a Set of Memory Operations The significant program orders for a given model can be represented by a directed acyclic graph with the nodes representing memory operations from the same processor and the edges representing the partial orders among these operations. For any given memory operation O, an ideal implementation would be capable of tracking the completion of the set of operations that are ordered before O in this graph. It may be impractical to maintain such level of detail for some relaxed models, however. In what follows, we describe both conservative and aggressive techniques for keeping track of the completion of various sets of operations for different models. Choosing the appropriate technique inherently depends on the mechanisms used for enforcing program orders, which is the topic of the next section. For purposes of this section, we assume ordering is enforced by delaying the issue of an operation

[28] The DASH design actually supports a hybrid version of the (a) and (b) solutions above. Requests from processors on a different cluster than the original requester are rejected. However, requests from other processors in the same cluster are serviced since it is simple to later inform them about the completion of the invalidations.

[29] It is actually alright to return the old value of the location to a read while invalidations are pending, but this old value should not be cached since it constitutes a stale copy.

until the appropriate set of previous operations are complete.

Sequential consistency is the simplest model to support because it imposes virtually a total order on operations from the same processor. This implies at most one outstanding memory operation at any given time. Therefore, the only information we need is whether the previous read or write operation has completed. The IBM-370, TSO, and PC models are also quite simple to support since they allow at most one outstanding read and one outstanding write operation at any time. On a read, the implementation can delay the issue of all future operations until the read completes (e.g., blocking read implementations trivially satisfy this); this alleviates the need to explicitly check for previous reads. Thus, an operation needs to at most check for completion of the previous write operation.

Unlike the models above, the PSO model allows more than a single write to be outstanding at any given time. As with the TSO model, the implementation can delay the issue of all future operations on a read. However, PSO still requires a mechanism to check for the completion of potentially multiple previous writes. One option is to keep a *count* of *outstanding writes* at each processor which is incremented when a read-exclusive or exclusive request is generated and is decremented on the corresponding exclusive or acknowledgement reply (depending on whether exclusive replies are delayed or eager, respectively; the two replies are piggybacked in the eager case if the home does not generate any remote invalidations).[30] Some implementations may not require an explicit counter. For example, implementations with a pending write buffer may simply wait for all entries in the buffer to be flushed as a sign that previous writes have completed. This latter technique may not be applicable if the protocol supports eager exclusive replies; a processor or cache hierarchy that expects a single reply to a write may deallocate entries associated with the write upon receiving the eager exclusive reply even though invalidations may be pending.

The remaining models all allow multiple outstanding read and write operations. The simplest set of models to support are WO, Alpha, and PowerPC, because the primary mechanism they require is to sometimes delay all future operations until all previous read and write operations are complete.[31] For implementations with blocking reads, there is no need to explicitly keep track of read operations. A simple solution for implementations that support multiple outstanding reads is to keep a combined count of outstanding read and write operations at each processor. As with PSO, some implementations may not require an explicit count. For example, mechanisms such as a read transaction buffer (see Section 5.2.3) that are already used to keep track of outstanding reads may be sufficient for detecting the completion of previous reads. Writes may still require an explicit count if the protocol implements eager exclusive replies.

Aggressive support for the RMO model is a little more complex than the above. RMO provides four types of fence operations that may be combined to selectively order previous reads or writes with respect to future reads or writes. Therefore, an aggressive implementation must keep track of previous read and write operations separately, e.g., by implementing separate counters for each. However, the main source of complexity arises from the occasional need to wait for previous operations of one type without delaying future operations of the same type. This scenario arises whenever the combination of fences requires waiting for previous reads without delaying future reads or waiting for previous writes without delaying future writes.

Consider the example in Figure 5.17 with a fence from previous writes to future reads; the arcs show the

[30] For designs that use counters, the size of the counter sets an upper limit on the number of outstanding operations from each processor. A subtle issue arises in designs that support eager exclusive replies: supporting n outstanding requests by the processor typically requires a counter that can keep track of more than n requests because even though only n may be waiting on the exclusive reply there may be more than n requests with pending invalidations.

[31] The Alpha model also requires program order to be maintained between read operations to the same address.

```
a1: A = 1;
b1: u = B;
c1: Fence(WR);
d1: v = C;
e1: D = 1;
f1: E = 1;
```

(a)

Figure 5.17: Difficulty in aggressively supporting RMO.

ordering imposed by RMO. Assume the implementation provides a count of outstanding write operations, and that the write to A is pending (i.e., count=1) when the fence instruction is issued. Thus, the read to C will be delayed until the count reaches zero. The problem is that writes past the fence, such as the writes to D and E, can be issued and can increment the count of outstanding writes before the write of A actually completes (an analogous problem arises with the read-to-write fence). This raises two issues: (i) the read of C may be unnecessarily delayed for the completion of future writes, and (ii) theoretically, the read of C may be delayed indefinitely since the count may never reach zero if the processor continuously issues write operations that miss in its cache. One possible solution to the latter problem is to limit the number of writes that can be issued past the fence if there are any operations waiting for the count to reach zero; the degenerate case of limiting the number of such writes to zero actually eliminates issue (i) as well. Nevertheless, both solutions constrain orderings beyond the minimum imposed by RMO.

More aggressive implementations of RMO can involve undue complexity. For example, a design based on counters requires multiple counters corresponding to intervals between fence operations. Request and reply messages may also need to be tagged with the corresponding interval in order to distinguish the counter that should be incremented or decremented. Referring back to the example in Figure 5.17, such an aggressive design would use separate counters to keep track of the writes before and after the fence, allowing the read of C to be delayed for only the writes before the fence. However, the performance gains from such an implementation may not justify the extra complexity especially if we expect fence operations to be infrequent.

Compared to RMO, achieving aggressive implementations for RCsc, RCpc, and the three PL models can be even more challenging. These models exhibit the same asymmetric behavior as RMO of waiting for previous operations of a certain type without necessarily delaying future operations of the same type. The additional complexity arises from the further distinctions of read and write operations. For example, the PL1 model (which is the simplest among these models) distinguishes between competing and non-competing reads and writes, leading to four distinct types of operations.

Consider the program orders imposed by PL1: (a) a competing read is ordered before any future operations, (b) a competing write is ordered after any previous operations, and (c) a competing write is ordered before a competing read (the first two conditions already imply read-read, read-write, and write-write order among competing operations). Condition (a) can be easily satisfied by delaying the issue of all future operations after a competing read until the read is complete. Condition (b) is more difficult to support aggressively since while a competing write is delayed for all previous operations, an implementation is allowed to issue future non-competing reads and writes. An aggressive implementation based on counters would require multiple counters corresponding to intervals between competing writes (along with message tags to choose the appropriate counter) so that operations before and after a competing write can be tracked by separate counters. Similar to RMO, conservatively using a single counter would require limiting the number of future operations that can be issued when an operation is waiting for the count to decrease to zero. Condition (c) is not as difficult to support aggressively since there can at most be one competing write outstanding before the competing read (future competing writes are delayed for the read by condition (a)). Therefore, we need to only determine whether the previous competing write has completed. More conservative implementations may use the same counter(s) as for condition (b) which implies that the competing read may be unnecessarily delayed for previous non-competing operations as well.

The PL2 model requires slightly more aggressive support since it further distinguishes competing operations into sync and non-sync categories (sync reads and writes are referred to as acquires and releases, respectively). Support for acquires and releases is virtually analogous to the support for competing reads and writes (i.e., conditions (a) and (b)) in PL1: an acquire is ordered before any future operations and a release is ordered after any previous operations. The ordering among competing operations limits the number of outstanding competing operations to at most one. Therefore, before issuing any competing operation, we need to know whether the previous competing operation (read or write, sync or non-sync) has completed.[32]

The PL3 model further distinguishes competing sync operations into loop and non-loop categories. The important optimization in PL3 is that the order from a competing write to a competing read does not need to be maintained if either the write is a loop write or the read is a loop read. One simplification may be to ignore the distinction between sync and non-sync operations (i.e., treat non-sync as sync) if non-sync operations are known to be infrequent in the target programs. With this simplification, keeping track of outstanding operations for PL3 is no more difficult than for PL1. In fact, since condition (c) of PL1 can be reduced to only maintaining an order from a non-loop write to a non-loop read (which are likely to be less frequent than loop reads and writes), less aggressive implementations of this condition may be tolerable in PL3.

There is often a latency associated with detecting the completion of previous memory operations. In some designs, this may be as simple as waiting for the MSHRs (or alternatively the pending-write buffer) corresponding to write operations to become empty, and the latency for the check may be quite small if there are no outstanding writes. In other designs, the latency for detecting completion of previous writes may be substantially larger. For example, in designs with eager exclusive replies, the counter that keeps track of outstanding writes may actually be external to the CPU chip. Therefore, checking for write completion may involve flushing any outgoing buffers to ensure that the counter is properly incremented by all previous writes and then waiting for the count to reach zero. Even when there are no outstanding writes, determining that the appropriate queues are flushed and checking the external count may take a long time. While this latency may not be directly or fully visible to the processor, it ultimately affects the rate at which write operations are retired especially if the counter must be checked often. Therefore, it does not make sense to support eager exclusive replies with external counters if our goal is to efficiently support a model like SC. However, for models that enforce program order less frequently, the benefit of eager exclusive replies may easily outweigh the higher latency to check an external counter.

[32]There is a subtle issue with respect to protocols with eager replies for writes when we wait for competing non-sync writes. Consider the sequence of a non-competing write followed by a non-sync write to the line. An eager exclusive or read-exclusive reply to the non-competing write may make the non-sync write hit in the cache. Nevertheless, we cannot assume the non-sync write is complete since the invalidations sent on behalf of the non-competing write may still be outstanding. This issue does not arise for a sync write (release) since the write inherently waits for all previous writes to complete.

There are a number of alternatives for decreasing the latency to detect the completion of writes in systems with counters that are external to the processor core. One alternative is to actually propagate the final acknowledgement replies all the way up the cache hierarchy so that mechanisms such as counters can be placed closer to the processor core. The trade-off here is the extra incoming messages propagated within the cache hierarchy. Another alternative is to keep an external counter but provide a continuous signal to the processor core as to whether the count is zero. Once the processor flushes the appropriate internal queues, it can check the signal after a specified bounded time (i.e., allowing enough time for any flushed requests to reach the counter and for the signal to be adjusted). This can reduce the latency for checking the counter since it alleviates the round trip latency of sending an external message to read the counter and is especially effective for quickly detecting zero counts.

Mechanisms for Enforcing Order

Enforcing the ordering imposed by multiprocessor dependence chains typically requires two different mechanisms: (i) a mechanism to maintain specific program orders among operations, and (ii) a mechanism to maintain the illusion of multiple-copy atomicity for specific write operations. The first is required for all types of multiprocessor dependence chains, while the second is only required for the third category of chains. This section describes various methods of supporting the above functionality.

Maintaining Program Order The previous section described techniques for keeping track of the completion of memory operations. This information can be used to enforce the appropriate partial orders among operations from the same processor. As with other implementation aspects, conservatively enforcing program ordering constraints for a model can in some cases lead to significantly simpler designs. There are effectively two ways to delay future operations for the appropriate set of previous operations. The simplest mechanism is to stall the processor until the previous operations complete. However, this leads to a conservative implementation for most relaxed models since all future memory and non-memory instructions are delayed while the processor is stalled. The alternative is to issue memory operations into one or more buffers and to enforce the appropriate ordering by controlling how operations are retired from these buffers. To enforce SC, for example, a combined read and write buffer can maintain sufficient order by retiring operations one at a time and waiting for each operation to complete before retiring the next operation. Buffers can play an important role in enforcing program orders since they allow the processor to proceed with future instructions.

Some implementations may use a hybrid of the above techniques. For example, consider supporting the TSO (or IBM-370 or PC) model in a design with blocking reads. As a result of blocking reads, the processor inherently maintains the program order from reads to future read and write operations by stalling on every read until the read completes. To maintain the required order among write operations, we can use a write buffer that delays retiring the write at its head until the previously retired write is complete. Thus, the processor does not directly stall on write operations, and is only stalled if it attempts to issue a write to a full buffer. By avoiding processor stalls in this way, much of the latency associated with write operations can be hidden even though the design maintains strict program order among write operations. The above example also illustrates how a conservative design choice, such as blocking reads, inherently enforces a number of the required program ordering constraints.

Memory operation buffers can be effective in enforcing the program order requirements for more relaxed models as well. Consider the ordering of writes in the PSO model, for example. PSO allows write operations to be reordered unless there is an explicit write fence between them. The processor can issue both write operations and write fence operations into a write buffer. The write buffer can retire writes in a pipelined fashion (i.e., without waiting for a write to complete before issuing the next write) until a write fence reaches the head of the buffer. A write fence at the head of the buffer stalls the retiring of writes until all previously retired writes are complete. This has the effect of ordering all writes before the write fence with respect to all writes after the write fence.

An analogous mechanism can be used for ordering writes in a model like PL1. In PL1, a competing write must be ordered after any previous read or write operations. The processor can issue writes into a write buffer. The buffer can retire writes in a pipelined fashion until a competing write reaches the head of the buffer. A competing write at the head of the buffer stalls the retiring of writes until all previous read and write operations are complete. Again, note that delaying a competing write for previous operations does not involve stalling the processor but is achieved through controlling the retiring of writes from the write buffer. The above is a conservative implementation of PL1 since writes after (in program order) the competing write may be unnecessarily delayed for the writes before the competing write. A more aggressive implementation requires a substantially more complicated buffering structure since the competing write must be delayed without delaying future non-competing writes;[33] as discussed in the previous section, detecting completion of operations also becomes more complex since we must separately keep track of operations before and after a competing write.

The above has mainly discussed delaying a single operation or a select set of future operations for a set of previous operations. Another common form of order is delaying future operations for a single previous operation. This type of order is also quite simple to support with buffers. Consider delaying future operations for a competing read in PL1. After a competing read is retired from the head of the buffer, retiring the remaining operations in the buffer can be delayed until the competing read is complete.

For most models, conservative design choices alleviate the need for extra mechanisms to enforce the appropriate program orders. As discussed before, blocking reads automatically maintain the order from reads to future operations. Similarly, blocking caches or strict FIFO queues can also automatically enforce many of the required program orders. Another example, common in dynamically scheduled processors that support precise exceptions, is the conservative delaying of write operations until all previous instructions and operations are complete (rolling back write operations would be too complex). In an analogous way, models that impose strict program ordering constraints alleviate the need for extremely aggressive implementations. For example, a lockup-free cache design for models such as IBM-370, TSO, or PSO needs to support a maximum of one read and one write operation outstanding at any given time.

The implementations described above enforce the required program ordering constraints at the early stages in the memory hierarchy. Once these orders are enforced, the lower components of the memory hierarchy can safely exploit optimizations that lead to reordering of memory operations to different lines (as discussed in the previous sections, some ordering requirements may still need to be guaranteed for operations to the same location or line). Therefore, the remaining part of the cache hierarchy on the outgoing path, the network, the memory system, and the incoming path in the cache hierarchy can all exploit reordering optimizations among messages to distinct memory lines.

[33] The conservative implementation is analogous to placing a write fence before a competing write instead of the more aggressive immediate fence.

P1	P2	P3		P1	P2	P3
a1: A = 1;	*a2:* u = B;	*a3:* v = C;	$\xrightarrow[a1,b1]{merge}$		*a2:* u = B;	*a3:* v = C;
b2: B = 1;	*b2:* C = 1;	*b3:* w = A;		*b2:* A,B = 1;	*b2:* C = 1;	*b3:* w = A;

(a) merging writes to different locations; outcome (u,v,w)=(1,1,0) should remain disallowed

P1	P2	P3		P1	P2	P3
a1: B = 1;	*a2:* u = B;	*a3:* v = C;	$\xrightarrow[a1,b1]{merge}$		*a2:* u = B;	*a3:* v = C;
b2: B = 2;	*b2:* C = 1;	*b3:* w = B;		*b2:* B = 2;	*b2:* C = 1;	*b3:* w = B;

(b) merging writes to same location; outcome (u,v,w)=(2,1,0) should remain disallowed

Figure 5.18: Merging writes assuming the PC model.

Bypassing, Merging, and Forwarding Buffering optimizations such as bypassing, merging, and forwarding must also be restricted appropriately to satisfy program ordering constraints imposed by a memory model (a number of these optimizations apply to lockup-free caches as well as to buffers). The initiation, value, and uniprocessor dependence conditions already impose constraints on such optimizations by imposing program ordering on operations to the same location. However, we must also satisfy program ordering constraints with respect to operations to different locations. For an operation O to bypass pending operations that are before it in program order, reordering operation O with respect to those operations must be allowed by the model. Similarly, for read forwarding, the read must be allowed to be reordered with respect all pending operations that are before it in program order. Finally, for merging operation O2 with an earlier operation O1, we must determine whether it is safe to reorder operation O2 with respect to the pending operations between O1 and O2 (excluding O1 itself).[34] For systems that support operation labels, merging also requires the resulting operations to inherit the more conservative labels from the constituent operations. For example, if we merge a non-competing write with a competing one in the PL1 model, the merged operation should have the competing label. It is interesting to note that limited forms of buffering, merging, and even read forwarding are actually possible even with strict models such as SC. For example, in SC, writes to the same location can be merged into a single write as long as no other operations appear among these writes in program order.

Merging writes can create a subtle correctness issue for specific combinations of models and implementations. Consider the PC model, for example, with a couple of write merging scenarios shown in Figure 5.18. The first example involves merging a pair of writes with consecutive addresses within the same memory line into a single write with larger granularity, while the second example illustrates merging of writes with the same address. In both cases, the set of possible outcomes after write merging should be a strict subset of the possible outcomes allowed by the model before the merging optimization. Aggressive implementations of PC may fail to satisfy the above requirement, however, if we allow write merging. The reason is write merging in the above examples effectively transforms a category two chain (refer back to categorization earlier in this section) into a category three chain, yet the system specification for PC does not require the order imposed

[34]For example, assuming the PL1 model, and given a non-competing write W1 followed in program order by a competing write W2, it is possible to merge the two writes even though W2 is not allowed to be reordered with respect to W1. In effect, merging makes it appear as if both writes happen at the same time.

by the category three chain to be upheld. Nevertheless, write merging is safe in any implementation of PC that conservatively upholds multiple copy atomicity for all writes. Therefore, whether write merging is safe depends both on the model and the specific implementation. Here is a list of other models along with pairs of writes that should not be merged if the implementation is sufficiently aggressive (i.e., does not support multiple-copy atomicity for the second type of write in the pair): a write followed by a competing write in RCsc or RCpc, two writes separated by a fence in PowerPC, and a write followed by a loop write in PL3.

Maintaining Multiple-Copy Atomicity As we discussed earlier in this section, enforcing the third category of multiprocessor dependence chains requires providing the illusion of multiple-copy atomicity in addition to maintaining the program order among operations. This section describes the implementation of the two techniques for enforcing multiple-copy atomicity (that were introduced earlier) in more detail.

The *first* technique is to require the write to complete with respect to all other processors before allowing a conflicting read from another processor to return the value of this write. Therefore, given a write W on P_i and a read R on a different processor P_j, W(j) $\xrightarrow{xo}$ R(j) must imply W(k) $\xrightarrow{xo}$ R(j) for all k except k=i. For invalidation-based protocols, the above condition can be enforced by simply disallowing a read request from another processor (i.e., different from P_i) to return the value of the write while there are invalidations pending for the line. The simplest and most practical way to achieve this is to delay (e.g., through nacking) the read request if any invalidations are pending for the line.[35] This can be supported in an analogous way to the implementations described earlier in this section for disallowing new write requests while there are pending invalidations. As before, this functionality can be supported either at the home for the line or at the requester depending on where invalidation acknowledgements are gathered.

Depending on the given model, the above restriction may apply to only a subset of the write operations. The functionality is not needed at all for the PC, RCpc, and PowerPC models since these models do not enforce any category three multiprocessor dependence chains. For the SC, TSO, IBM-370, PSO, WO, RCsc, Alpha, and RMO models, all writes must be treated conservatively. Finally, the PL1 and PL2 models impose this restriction on competing writes only, and the PL3 model is similar except it excludes competing loop writes from this restriction.[36] For the latter models, extra information must be communicated to the memory system to distinguish writes that must be treated conservatively.

The first technique is more difficult to apply to update-based protocols since shared copies of a location are inherently updated at different times, allowing a read from a different processor to return the new value while updates to other copies are pending. One solution is to use a two-phase update protocol (a similar technique is described by Wilson and LaRowe [WL92]). The first phase involves sending updates to the cached copies and receiving acknowledgements for these updates. In this phase, a processor (other than the processor that issued the write) must not allow its reads to return the value of the updated location. In the second phase, a

[35] Alternatively, we can service the read request by providing an uncached version of the old value for the line (i.e., value before the current write is done). Note that providing the new value to a read-exclusive request from another processor can also cause problems since a subsequent read on that processor can see the value too early; however, as we mentioned earlier in this section, most designs delay read-exclusive requests while there are outstanding invalidations in order to simplify detecting the completion of writes.

[36] There can be a subtle correctness issue for aggressive implementations of PL2 and PL3 that do not delay a read from returning the value of a non-competing write even if the write has pending invalidations. The problem arises if the implementation also supports eager exclusive replies. Consider a non-competing write followed in program order by a competing write to the same cache block. With eager replies, the reply to the non-competing write may make the competing write appear as a cache hit even though there are still pending invalidations due to the non-competing write. Meanwhile, the memory system may be forwarding read requests to this cache, allowing them to see the new value of the competing write while there are outstanding invalidations. The solution for both PL2 and PL3 is to delay a competing non-sync write for previous non-competing writes to complete (a sync write already satisfies this requirement).

confirmation message is sent to the updated copies to signal the receipt of all acknowledgements. A processor can use the updated value from its cache once it receives the confirmation message from the second phase. The processor that issued the write can consider the write complete at the end of the first phase since all stale copies are eliminated.

Another solution for update protocols is to use a hybrid two-phase scheme, with an invalidation phase followed by an update phase. The first phase involves invalidating (instead of updating) the copies and gathering the acknowledgements. Read requests are not serviced while invalidations are pending, similar to the solution described for a pure invalidation protocol. After the invalidations are acknowledged, the second phase sends updates with the new value to the copies (that were invalidated).[37] The write is considered complete once the invalidations are acknowledged; the update requests in the second phase do not require acknowledgements since they simply constitute data movement within the memory hierarchy.

The *second* technique for maintaining multiple-copy atomicity allows the new value for a line to be read while there are pending invalidations or updates, but alters the way program order is enforced. To enforce the program order from a read to a following operation, the latter operation is not only delayed for the read to complete but is also delayed for the write that provided the value for the read to complete with respect to all processors (other than the processor that issues the write). Therefore, given W $\xrightarrow{co'}$ R $\xrightarrow{po}$ Y where R and Y are on a different processor than W on P_k, the above condition enforces W(i) $\xrightarrow{xo}$ Y(j) for all i,j except i=k (the first technique discussed above enforces the stricter condition of W(i) $\xrightarrow{xo}$ R(j) for all i,j except i=k). Consider an invalidation protocol. The first phase involves invalidating all copies as usual. The memory system must keep track of any processors whose read is serviced while there are pending invalidations to the line and must notify those processors once the invalidations are complete. Furthermore, each processor must appropriately keep track of read operations that expect the separate notification and must appropriately delay for this notification when enforcing program order from these reads to future operations. The write is considered complete as soon as the invalidations are acknowledged. The above technique can be easily adapted to update-based schemes as well. The program orders that must be treated conservatively are model dependent. For SC, IBM-370, TSO, and PSO, any R $\xrightarrow{po}$ Y must be treated conservatively. Here are the program orders that must be treated conservatively for the other models ($\xrightarrow{spo}$ is defined differently for each model): R $\xrightarrow{spo'}$ Y for WO, Rc $\xrightarrow{spo'}$ Y for RCsc, R $\xrightarrow{spo}$ Y for Alpha, R $\xrightarrow{spo}$ Y for RMO, Rc $\xrightarrow{spo'}$ Y (only if R returns the value of a Wc) for PL1 and PL2, and Rc $\xrightarrow{spo''}$ Y (only if R returns the value of a Wc_nl_ns) for PL3. Except for the PL models, the other models require all writes to support the second phase notification. For PL1 and PL2, only competing writes, and for PL3, only competing non-loop or non-sync writes must support the second phase notification.

Except for implementations of the first technique for invalidation-based protocols, the other implementations are less practical since they incur extra design complexity and require extra messages to be communicated. For this reason, it is typically cumbersome and impractical to support update protocols in a scalable shared-memory system for models that require multiple-copy atomicity for writes. The properly-labeled models allow a more efficient implementation of update protocols since non-competing writes (which are typically much more frequent than competing writes) do not need to behave atomically in these models and can be implemented using a simple single-phase update protocol; competing writes may be supported by using either a two-phase update scheme or a simple invalidation scheme.

[37] Depending on the granularity of valid bits in the caches, the invalidation and update requests may either be at the granularity of a whole cache line or a smaller portion of the line.

5.3.6 Supporting the Reach Condition

Section 4.2 from the previous chapter describes the intuition behind the reach condition, while Appendices F and I provide the formal definition for the reach relation ($\xrightarrow{rch}$) for both programmer-centric and system-centric models. The reach condition applies only to models that relax the program order from read to write operations; this includes WO, RCsc, RCpc, Alpha, RMO, and PowerPC among the system-centric models and PL1, PL2, and PL3 among the programmer-centric models. While the programmer-centric models specify the reach condition as a separate constraint, the system-centric models support the reach condition more conservatively by incorporating the reach relation into the significant program orders that comprise the multiprocessor dependence chains.

The main purpose for the reach condition is to disallow anomalous executions that arise if we allow "speculative" write sub-operations to take effect in other processors' memory copies. The reach relation is defined between certain R $\xrightarrow{po}$ W pairs and consists of two components. The first component captures the notions of uniprocessor data and control dependence. Informally, a read operation reaches a write operation (i.e., R $\xrightarrow{rch}$ W) that follows it in program order (R $\xrightarrow{po}$ W) if the read determines whether the write will execute, the address accessed by the write, or the value written by it. The second component of reach is particular to multiprocessors. A read operation R reaches a write operation W if R controls the execution, or address of, or value written (in case of a write) by another memory operation that is before W in program order and must be ordered before W in the execution order (i.e., if we were to conservatively satisfy the uniprocessor dependence condition or the multiprocessor dependence chains).

Given R $\xrightarrow{rch}$ W, the simplest way to satisfy the reach condition is to delay the write for the read to complete. The formal definition for the reach relation presented in Appendix F is quite complicated primarily because we tried to provide an extremely aggressive set of conditions. However, most hardware implementations end up trivially satisfying the reach condition because of their conservative designs. For example, designs with blocking reads trivially satisfy this condition since for any R $\xrightarrow{po}$ W, the read is guaranteed to complete before the write. At the same time, even aggressive dynamically scheduled processors with non-blocking reads may trivially satisfy the reach condition since they may conservatively delay writes for all previous instructions to complete as a simple way of supporting precise exceptions. Nevertheless, a key advantage of our aggressive formulation for the reach condition is that it does not impose any order on future read operations, thus allowing the processor to issue "speculative" reads (e.g., issuing reads past a branch before the branch is resolved).

The conditions presented in the appendix allow for more aggressive implementations. For example, a processor can allow a write W to complete even when there are outstanding reads before it as long as (a) it is known that W will indeed be executed (i.e., will not have to be rolled back) with the given address and value, and (b) for any memory operation O before W in program order, whether O will be executed along with its address and write value (in case of writes) are already resolved. Thus, a write needs to only be delayed until the control flow for the execution, and the addresses and values of any operations before the write and the write itself, are resolved. Even more aggressive hardware designs may be achieved by more closely matching the specification in the appendix. However, the performance gains from doing this may be marginal.

5.3.7 Supporting Atomic Read-Modify-Write Operations

Read-modify-write operations are commonly used for implementing synchronization algorithms in shared-memory multiprocessors. Examples include test-and-set, compare-and-swap, fetch-and-op, and the load-locked and store-conditional instructions [Sit92] which have gained popularity more recently. Condition 4.7 from the previous chapter specifies the consistency requirements for supporting atomic read-modify-write operations. Intuitively, this constraint requires conflicting write operations from other processors to appear to execute either before or after a read-modify-write.

The read-modify-write functionality may be supported either at the processor or at the memory. From a performance point-of-view, operations supported at the processor can exploit reuse locality through caching. On the other hand, supporting high contention operations at the memory can minimize serialization latencies (e.g., consider a fetch-and-increment used to implement barrier synchronization).

A read-modify-write operation consists of both a read and a write component. The read component is allowed to complete as soon as the write component is serialized with respect to other writes (load-locked and store-conditional operations will be discussed later). Assuming an invalidation protocol with write back caches, a cache-based implementation can service the read component as soon as the cache acquires ownership for the line; with eager exclusive replies, this event can actually occur before invalidations caused by the write component are complete. Of course, the read-modify-write operation can be serviced immediately if the cache already has exclusive ownership for the line. Memory-based implementations are simpler to support if we guarantee that memory is always the owner for the location by disallowing caches from maintaining a dirty copy, or more conservatively any copy, of the line. In this way, the read-modify-write can be serialized with respect to other conflicting writes as soon as it arrives at the memory and we can respond to the read component in parallel with sending out any invalidations.

Condition 4.7 is a reasonably aggressive specification since it only requires *conflicting* write operations from *other* processors to appear to execute either before or after the read-modify-write. A more conservative specification could require *any write* operation (regardless of its address) to behave atomically with respect to a read-modify-write. In fact, the original specifications for the SPARC V8 and V9 models (i.e., TSO, PSO, and RMO) impose the more conservative semantics.[38] Figure 5.19 provides an example that illustrates the semantic difference between the aggressive and conservative specifications (with other constraints the same as TSO or PSO). Assume all locations are initially zero. The test-and-set on each processor reads the value zero and sets the value to 1, leading to (u,v)=(0,0) in all executions. Meanwhile, the outcome (w,x)=(0,0) is allowed by the aggressive semantics and disallowed by the conservative one. The more conservative specification, as used in TSO, PSO, and RMO, is less desirable since it disallows the servicing of the read component until all invalidation or update requests generated by the write component are complete, making it more difficult to implement the read-modify-write operations at the memory, or in caches that support eager exclusive replies.

Load-locked and store-conditional operations are slightly different from a typical read-modify-write. The load-locked operation returns the value of a given location, which can be modified and written back using the store-conditional operation. The store-conditional successfully modifies the location if no other

[38]The specifications for TSO, PSO, and RMO in Appendix I use the aggressive constraints for read-modify-writes and capture the effect of the conservative constraints by imposing extra program orders.

P1		P2	
a1:	u = test&set(A);	*a2:*	v= test&set(B);
b2:	w = B;	*b2:*	x = A;

Figure 5.19: Semantics of read-modify-write operations.

processor performs a conflicting write between the time the load-locked completes and the time the store-conditional is retired. Otherwise, the store-conditional simply fails without modifying the location. The store conditional's return value indicates its success or failure; the load-locked/store-conditional sequence can be simply repeated until the store-conditional succeeds. In this way, the load-locked/store-conditional pair can be used to implement an arbitrary set of atomic read-modify-write operations.

An implementation can support the above functionality by logically maintaining a lock-flag and a lock-address register at each processor. A load-locked operation sets the lock-flag and stores the address for the lock location in the lock-address register. Incoming invalidation and update requests from other processors are monitored and matched against the lock-address register, and a successful match resets the lock-flag. Therefore, the lock-flag is used as a conservative indicator of whether another processor has performed a store since the last load-locked operation. For correctness, the lock-flag is also cleared if the line corresponding to the lock-address is replaced from the cache or if there is a context switch.[39]

Even though the above functionality seems simple to support, there are a number of subtle issues that make correct implementations of load-locked and store-conditional operations quite challenging. Appendix N describes some of the implementation issues, including the complexity that arises from the need to avoid livelock conditions whereby no process can successfully complete a load-locked/store-conditional sequence.

There are also important efficiency issues associated with the use of load-locked/store-conditional sequences in scalable shared-memory systems. Performing a read-modify-write to a location that is not already cached by a processor leads to at least two misses, a read miss for the load-locked followed by a write miss for the store-conditional, both of which may involve large round-trip latencies. One possible optimization is to prefetch the line in exclusive mode either before the sequence or at the same time the load-locked is serviced, which incurs just a single long-latency miss. However, sequences that exploit this optimization must support some sort of exponential back-off to reduce the probability of livelock.[40]

5.3.8 Comparing Implementations of System-Centric and Programmer-Centric Models

The system specifications for the properly-labeled (PL) models are more aggressive than even the most relaxed system-centric specifications (e.g., RCsc, RCpc), primarily because the multiprocessor dependence chains are more selective in the PL specifications. However, taking full advantage of the aggressive specifications in a hardware design can lead to excessive complexity and can have diminishing performance returns.[41] For this

[39] A replacement makes the cache unaware of future writes to the line. A context switch in the middle of a load-locked/store-conditional sequence can incorrectly allow the load-locked of one process to lead to the success of the store-conditional in another process.

[40] Livelock may arise because grabbing the line in exclusive mode can lead to the failure of other processors' sequences without a guarantee that this processor's store-conditional will succeed.

[41] For example, Section 5.3.2 explained why an implementation may still uphold the coherence requirement for all writes even though PL specifications require it for competing writes only.

reason, practical hardware implementations of a programmer-centric model such as PL2 and an aggressive system-centric model such as RCsc are often quite similar in performance and in the type of optimizations they exploit. Nevertheless, programmer-centric models still enable a few extra optimizations (not allowed by the RCsc or RCpc specifications) that are relatively simple to exploit in hardware.

One key advantage of the PL specifications compared to the RCsc specification is that non-competing writes (and loop writes in PL3) do not need to appear atomic with respect to multiple copies. This is especially beneficial for update-based designs since non-competing and loop writes can safely use a simple update protocol for maintaining coherence among copies (Section 5.3.5 described the difficulties of supporting multiple-copy atomicity for updates). Even though the PC, RCpc, and PowerPC models allow all writes to appear non-atomic, the PL1 and PL2 models are simpler to program (e.g., they do not require the extra categorizations of PL3) and yet enable this optimization for the frequent write operations. Another potential advantage of the PL specifications is that a more aggressive version of the termination condition may be used, as discussed in Appendix H.[42]

In contrast to practical hardware designs, designs that support cache coherence in software benefit more, and can better tolerate the extra complexity, from exploiting the aggressive nature of the PL specifications. This issue is further discussed in Section 5.6.

5.3.9 Summary on Maintaining Correct Order

The previous sections provided a detailed overview of various design issues related to correctly and efficiently implementing a given memory model. As we saw, there is often a trade-off between design complexity and supporting more aggressive reordering optimizations. Furthermore, there are numerous subtle issues in maintaining correctness with the more aggressive optimizations.

5.4 More Aggressive Mechanisms for Supporting Multiprocessor Dependence Chains

This section describes three different techniques for achieving more efficient implementations for a given memory model. The first technique exploits the aggressive nature of our specifications (i.e., the fact that execution order is only imposed among conflicting operations) to enable the *earlier acknowledgement* of invalidation and update requests. The second technique involves automatic *prefetching* of values for operations that are delayed due to consistency model constraints. Finally, the third technique employs *speculative execution* to speculatively service memory operations even though the consistency model would require the operation to be delayed. Most of the ideas for the prefetching and speculative execution techniques have been published in our previous paper [GGH91b]. All three techniques are applicable across the full range of models, even though implementations of stricter models are expected to gain the most from these optimizations. Furthermore, the latter two techniques allow us to exploit many of the hardware optimizations that motivate the use of relaxed memory models while still maintaining the illusion of a stricter model such as SC.

[42] A more minor advantage of PL specifications is mentioned in Section 5.4.1 in the context of ordering incoming requests whereby an incoming non-competing update request is allowed to bypass previous update or invalidate requests.

5.4.1 Early Acknowledgement of Invalidation and Update Requests

So far, we have assumed that invalidation or update requests are acknowledged only after all stale copies of the line are eliminated from a processor's cache hierarchy. Therefore, the acknowledgement reflects the completion of the write with respect to a given processor. To reduce the latency for write acknowledgements especially in designs with deep cache hierarchies, a common optimization is to acknowledge an invalidation or update request as soon as the request is placed in the cache hierarchy's incoming queue which is before all stale copies are actually eliminated.[43] This optimization also simplifies the buffer deadlock issues in handling incoming invalidation or update requests within the cache hierarchy since an acknowledgement reply is no longer generated from within the hierarchy (see Section 5.2.3). However, naive applications of this optimization can lead to incorrect implementations since an acknowledgement reply no longer signifies the completion of the write with respect to the target processor. This section describes two distinct implementation techniques that enable the safe use of early acknowledgements.

To simplify discussing the above optimization, we introduce an additional set of physical events for a write operation. A write operation currently consists of multiple completion events with respect to each of the n processors. For each write, we also define a *commit event* with respect to each processor. The commit event corresponds to the time when the write is either explicitly or implicitly acknowledged and precedes the completion event with respect to a processor in cases involving an early acknowledgement. The commit and completion events (with respect to a given processor) coincide in time if no invalidation or update requests are exchanged, or if the early acknowledgement optimization is not used.

In designs with early acknowledgements, the program order between a write W and a following operation Y is enforced by waiting for W to *commit* with respect to every processor before issuing Y (there is no longer an explicit message that signals the completion of the write). This does not directly satisfy conservative specifications (e.g., conditions for SC in Figure 4.4) which prescribe enforcing program order from a write by waiting for the write to *complete* with respect to every processor. Instead, such designs exploit the flexibility of the more aggressive specifications (e.g., conditions for SC in Figure 4.7) whereby completion order is only enforced among the endpoints of a multiprocessor dependence chain and not at every program order arc.

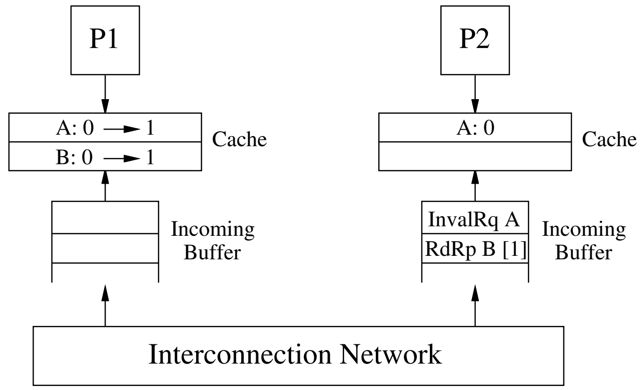
Figure 5.20 shows an example to illustrate the issues related to early acknowledgements. Consider the program segment in Figure 5.20(a) with all locations initialized to zero. The outcome (u,v)=(1,0) is disallowed under the SC model. Consider an invalidation-based protocol with write-through caches. Assume P1 initially caches both locations and P2 caches location A. Without early acknowledgements, P1 issues the write to A, waits for it to complete, and proceeds to issue the write to B. Therefore, the stale copy of A at P2 is eliminated before P1 even issues its second write. Furthermore, as long as P2 ensures its reads complete in program order, the outcome (u,v)=(1,0) will indeed be disallowed.

Now consider the scenario with early invalidation acknowledgements. P1's write to A causes an invalidation request to be sent to P2. The invalidation is queued at P2 and an acknowledgement reply is generated. At this point, the write of A is committed but has yet to complete with respect to P2 (i.e., P2 can still read the old value of A). While the invalidation request remains queued, P1 can proceed to issue its write to B, and P2 can issue its read request to B. Figure 5.20(b) captures the state of P2's incoming buffer at this point, with both the invalidation request for A and the read reply for B (with the return value of 1) queued. A key issue

[43] One reason we can acknowledge the request before checking the state of the line in the caches is that an update or invalidate request sent to a clean cache copy is always acknowledged (i.e., never nacked) even if the line is no longer present in the cache hierarchy (e.g., due to a replacement). In other words, it is trivial to service the request when the stale copy is no longer cached.

P1	P2
a1: A = 1;	*a2:* u = B;
b1: B = 1;	*b2:* v = A;

(a)



(b)

Figure 5.20: Example illustrating early invalidation acknowledgements.

is that allowing the read reply to bypass the invalidation request in the buffer, which would be allowed in an ordinary implementation, will violate SC because P2 can proceed to read the stale value for A out of its cache after obtaining the new value for B.

Figure 5.21 depicts an alternative view of the above scenario by showing the order among completion events. Write operations are shown with a completion event with respect to each of the two processors, while read operations are shown with a single completion event with respect to the issuing processor. Figure 5.21(a) shows the scenario without early invalidation acknowledgements. As shown, P1 ensures the write to A completes with respect to both processors before the write to B completes with respect to any processor, and P2 ensures its two reads complete in program order. The fact that the read of B returns 1 implies that the write of B completes with respect to P2 before the read of B completes. The above orders imply that the write of A completes with respect to P2 before the read of A completes, which correctly disallows the outcome (u,v)=(1,0). Figure 5.21(b) shows the scenario with early invalidation acknowledgements. For each write, there is now also a commit event with respect to each of the two processors. The program order on P1 is upheld by ensuring the write to A commits with respect to both processors before the write to B commits with respect to any processor. The order imposed on P2 and the implied order between the write of B and the read of B are the same as those in Figure 5.21(a). These orders imply that the write of A is indeed committed with respect to P2 before the read of B or the read of A complete. However, to disallow (u,v)=(1,0), we need to somehow ensure that the write of A completes with respect to P2 before the read of A completes (shown as a dashed arrow).

The *first* solution we discuss imposes ordering constraints among incoming messages with respect to previously committed invalidations and updates. Referring back to the example, this solution would disallow the read reply from bypassing the invalidation request, which forces the committed write to A to complete with respect to P2 before the read of B completes. The *second* solution does not impose any ordering constraints among incoming messages. Instead, it requires previously committed invalidation and update requests to be serviced anytime program order is enforced for satisfying a multiprocessor dependence chain. In the example, this latter solution would force the invalidation request to be serviced (e.g., by flushing the incoming queue) as part of enforcing the program order from the read of B to the read of A. Thus, both solutions correctly

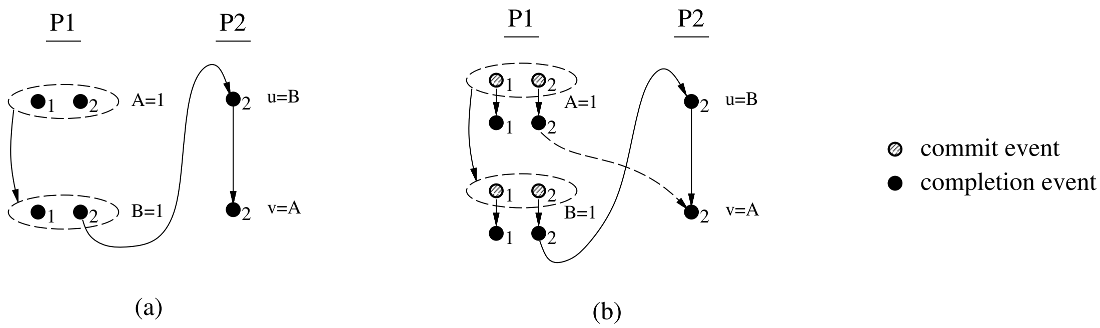


Figure 5.21: Order among commit and completion events.

disallow the outcome (u,v)=(1,0). In what follows, we present each solution in more detail, discuss their trade-offs, describe hybrid combinations of the two techniques, and also compare them to previous work in this area.

Imposing Orders among Incoming Messages

Designs without early acknowledgements can safely reorder incoming messages to different memory lines; the only ordering constraints are imposed among a subset of incoming messages that address the same memory line (e.g., refer to Sections 5.3.2 and 5.3.3, and Appendix K). In contrast, arbitrary reordering of incoming messages to different memory lines can violate correctness in designs that use early acknowledgements.[44] One possible solution is to impose a total FIFO ordering among all incoming messages. However, as we will see below, the actual ordering constraints required for correctness are much less restrictive.

Invalidation-Based Protocol Table 5.1 in Section 5.1.2 enumerates the typical set of incoming request and reply messages used within a cache hierarchy. Consider an invalidation protocol with write back caches. The possible incoming messages are as follows:

- incoming requests: read, read-exclusive, invalidate, and
- incoming replies: read, read-exclusive, exclusive, invalidate-ack.[45]

To achieve correct behavior, it is *sufficient* to maintain the queue order from an incoming reply message to a previous incoming invalidate request.[46] No other ordering constraints are required for incoming messages to different memory lines.[47]

[44] In our discussion, we implicitly assume that the ordering constraints imposed among incoming messages that address the same memory line are identical to those in a design without early acknowledgements.

[45] If the design maintains counters that keep track of previous writes, the response from checking the counter must be considered the same as an invalidate-ack reply since it signals the fact that the write has been committed or completed with respect to all processors. Replies to explicit fence instructions must be treated in a similar way.

[46] For designs with multiple paths for incoming messages, maintaining this order can be simplified by placing incoming invalidate requests in the same path as incoming replies. This is possible because invalidate requests do not generate a reply from within the hierarchy, thus alleviating the deadlock issues associated with general requests.

[47] In a cache hierarchy, an incoming read-exclusive request may generate an invalidation request when it reaches the top most write back cache that has the dirty copy in order to eliminate any copies at the higher levels. At the same time, the request also generates an outgoing read-exclusive reply typically merged with an implicit early invalidate-ack reply. The write is considered committed with respect to the processor at this time and the newly generated invalidation request must not be reordered with respect to future incoming replies that reach that point in the hierarchy.

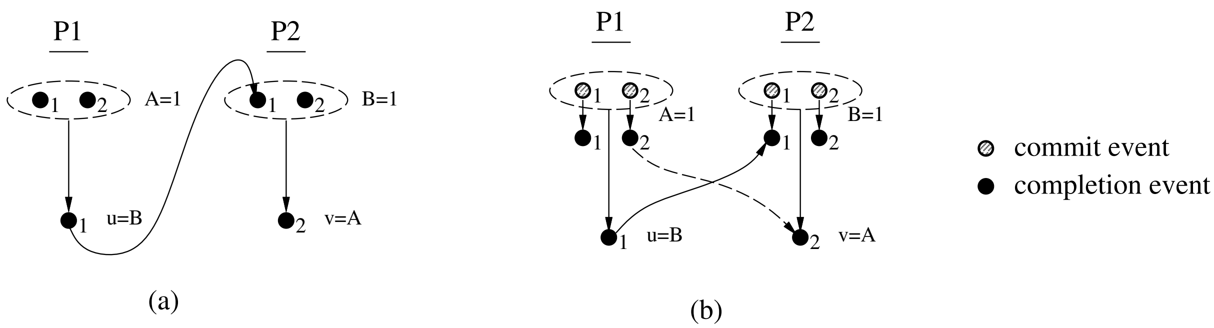


Figure 5.22: Multiprocessor dependence chain with a R $\xrightarrow{co}$ W conflict order.

Therefore, incoming requests can be safely reordered with respect to one another, including the reordering of two invalidation requests. Incoming replies can also be reordered with respect to one another. Incoming read or read-exclusive requests can be reordered with respect to incoming replies. And finally, incoming invalidation requests can be reordered with respect to previous incoming replies. The above conditions are model independent, at least for the models discussed in this thesis. Therefore, even a design that supports a strict model such as SC can exploit the reorderings discussed above.[48]

We refer back to Figure 5.21 to build some intuition for how sufficient order is maintained for correctness. The fact that the read of B on P2 returns the value 1 implies that the read reply for B arrived at P2 sometime after the write of B completes with respect to P2. Furthermore, we know that the write of A is committed with respect to P2 before the above completion event. The above two facts imply that the incoming invalidate request for location A is queued at P2 before the reply for B arrives. Therefore, maintaining the order from the reply to the previous incoming invalidate guarantees that the write of A completes with respect to P2 before the read of B completes, correctly disallowing the outcome (u,v)=(1,0).

Reasoning about the correctness of multiprocessor dependence chains consisting of R $\xrightarrow{co}$ W conflict orders is more challenging. Figure 5.22 shows the behavior of one such chain (same chain as in Figure 5.14(b)). The left side of the figure depicts the behavior in a system without early acknowledgements, which clearly ensures that the write of A completes with respect to P2 before the read of A. The right side depicts a system with early acknowledgements. To show that the chain is correctly upheld, we need to first show that the write of A is committed with respect to P2 sometime before the read of A completes. Second, we need to show that P2 receives an incoming reply after the write of A is committed with respect to it and consumes this reply before the read of A completes. Along with the ordering constraints among incoming messages, the above combination ensures that the write of A completes with respect to P2 before the read of A.

Figure 5.23 graphically shows the steps involved in reasoning about the correctness of the example chain from Figure 5.22. For each write operation, we have added an extra operation labeled as a *miss* with its own commit and completion events. For a write operation that misses in the cache, the "miss" operation is the *same* as the original write operation. For cache hits, the extra operation represents the last operation that

[48] The following trade-offs should be considered in exploiting the above flexibility. Delaying the service of incoming replies and incoming read or read-exclusive requests has a direct effect on the latency of operations for this processor or other processors. Meanwhile, delaying the service of incoming invalidate or update requests does not directly affect the latency of any processors' operations since an early acknowledgement is sent for these requests. In fact, delaying incoming invalidation requests can have a positive effect by reducing the amount of false sharing [DWB+91]; however, delaying invalidation or update messages for too long can hurt performance by delaying the time when the processor observes the most recent value for a location (e.g., if a processor is waiting for a lock or flag to be released).

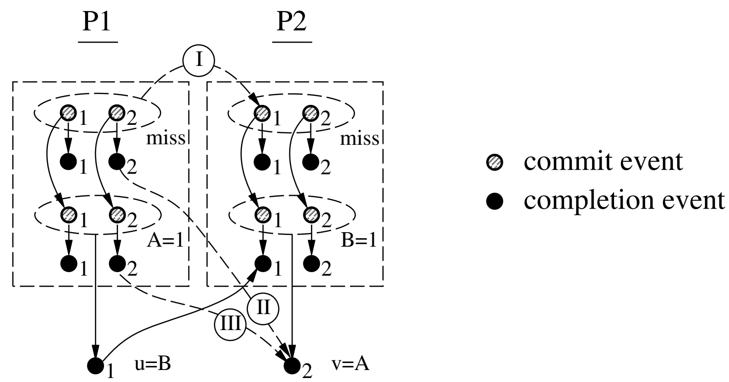


Figure 5.23: Reasoning with a chain that contains a R $\xrightarrow{co}$ W conflict orders.

fetched a dirty copy of the line into the cache (e.g., due to an earlier write to the same word, or a different word in the same line); the cache hit itself does not lead to external communication. We need to consider the miss event because it is the reply to this operation that pushes previously committed invalidations to completion. The dashed box enclosing the miss operation and the actual write operation on each processor denotes the atomicity relationship between the two events; by definition, a read or write event from a different processor to the same line may not be ordered between these two operations, otherwise the line would not remain dirty in the cache.

Consider the reasoning for the above chain. Assume the miss operation on P2 commits with respect to P1 either before the miss operation on P1 commits with respect to P1 or before the latter commits with respect to P2. The above implies that the miss operation on P2 completes with respect to P1 before the read of B completes, which further implies (by the atomicity relationship between the miss and write operations) that the write of B completes with respect to P1 before this read. This leads to a contradiction with an already existing order from the read of B to the completion event for the write of B. Therefore, it has to be that the miss operation on P2 commits with respect to P1 after the miss operation on P1 commits with respect to P1 and P2, as depicted by the dashed arrow labeled (I). Given this order, the second dashed arrow labeled (II) follows based on the incoming message orders. Finally, the third dashed arrow labeled (III) follows from the second one based on the atomicity relationship between the miss and write operations on P1.

Appendix O provides a number of other examples and further intuition about the above technique.

Update-Based Protocols The relevant incoming messages for an update-based protocol are listed below:

- incoming requests: read, update-read, update, and
- incoming replies: read, read-exclusive, update, update-exclusive, and update-ack.

Similar to the case for invalidation protocols, an incoming reply must be ordered with respect to a previous incoming update request. In addition, there is an extra requirement that incoming update requests may not bypass one another.[49] To illustrate the importance of this latter constraint, Figure 5.24 illustrates the same scenario as in Figure 5.20 except with updates instead of invalidations. P2 is assumed to initially have copies of both A and B. As shown, the updates due to the writes of P1 are queued in order and maintaining this order

[49]The incoming update-read request behaves in an analogous way to the read-exclusive request in a hierarchy of caches.

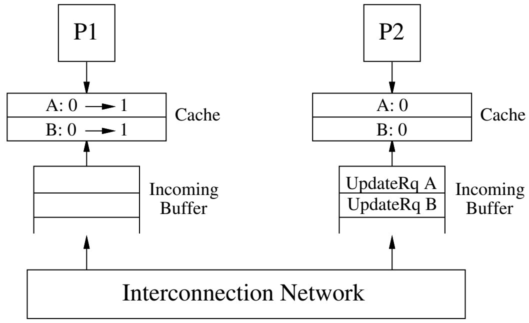


Figure 5.24: Example illustrating early update acknowledgements.

is important for correctness since P2 may otherwise read the new value for B and later read the old value for A.

In systems with hybrid invalidation and update protocols, the sufficient condition is a combined version of the conditions for each type of protocol: an incoming reply or an incoming update request must be ordered with respect to a previous incoming invalidate or update request.[50]

The order from an incoming update request to previous incoming invalidate or update requests may be relaxed for the PL models due to the aggressive specification of multiprocessor dependence chains in these models: any conflict order within a chain is between a pair of operations labeled as competing. The above may be exploited if incoming update requests carry a label to distinguish whether the write is competing or non-competing. Given such information, it is sufficient to only ensure the order from an incoming competing update request to previous incoming invalidate or update requests. In other words, an incoming non-competing update request is allowed to bypass previous update or invalidate requests.

The techniques described above for invalidation and update protocols can be easily extended for use in multi-level cache hierarchies. The ordering among messages in the incoming queues is the same as described above. Furthermore, a reply from a level i cache (i.e., if the cache services an outgoing request) is placed into the incoming queue for the level i-1 cache as usual. This reply stays ordered with respect to any invalidation or update requests that are already in the incoming queues for the level i-1 cache and above. The relaxed ordering constraints on incoming messages can also be beneficial in designs with interleaved or multi-banked caches for exploiting the overlap of operations among multiple banks.

Related Work on Imposing Orders among Incoming Messages Afek et al. [ABM89, ABM93] proposed the idea of *lazy caching* for supporting sequential consistency. The implementation they propose uses a bus-based update protocol for maintaining cache coherence. When a processor issues a write, it only waits for the updates of its write to be queued into the FIFO incoming queues of the other processors and can continue with its next operation as soon as the reply from the bus comes back to it which may be before the updates actually take effect in the other processors' caches. The authors prove that this implementation satisfies sequential consistency. However, this work has several major limitations. First, the algorithm depends on the broadcast mechanism of a bus interconnect and does not apply to systems with general interconnects where

[50]Early acknowledgement of incoming update requests can create subtle correctness issues if a protocol sends an update request to a dirty cached copy mainly because of a race with the possible writing back of the line. The protocol we have discussed avoids this by always sending an update-read request to a dirty copy; the update-read is implicitly acknowledged only after it finds the dirty copy in the hierarchy and a negative acknowledgement is sent if the line has already been written back.

writes to different addresses may commit in a different order with respect to different processors. Second, the algorithm assumes that every write goes to the bus and comes back through the incoming queue, thus effectively flushing the incoming queue on every write before the processor can continue with a read. This fails to handle designs that maintain a dirty state in a write back cache, and in effect disallows the processor from executing reads in isolation (i.e., without servicing the incoming queue) after a write. Third, the fact that write back caches are not handled implies the incoming queue for a processor never receives a read or read-exclusive request from another processor and also the cache never needs to reply to such a request. Finally, the algorithm assumes a single level of caching.

Scheurich proposes a similar idea to lazy caching for supporting SC in a bus-based system [Sch89]. Although Scheurich does not make the same restrictive assumptions as in the lazy caching work, his implementation still depends on the broadcast mechanism of a bus and on FIFO ordering among incoming and outgoing messages. Furthermore, the description of the implementation is sketchy and informal.

Our work provides a completely general solution that alleviates the limitations discussed above. We show that a strict FIFO ordering among incoming messages is not necessary for correctness. In addition, our technique applies not only to SC, but to all other relaxed models. Section 5.5 describes how this technique can also be adapted to systems with restricted interconnects such as buses.

Servicing Incoming Messages when Enforcing Program Order

As an alternative to imposing extra constraints on the order of incoming messages, it is possible instead to require invalidation or update requests that are in the incoming queues to be serviced whenever the implementation enforces certain program orders that are part of a multiprocessor dependence chain.

Given $X \xrightarrow{po} Y$ is part of a chain, the program order can be enforced by delaying Y's completion or commit events until X completes or is committed with respect to all processors (depending on whether X is a read or a write, respectively). Conservatively, Y may also be delayed for the service of any invalidation or update requests that have entered the incoming queue before X completes (or commits with respect to all processors, if X is a write). One way to do this is to force the incoming queue to be flushed.[51] The above forces any writes that have committed with respect to this processor before X to complete with respect to the processor before Y is issued. Referring back to the example in Figures 5.20 and 5.21, the write of A commits with respect to P2 before the read of B completes and the read of A is delayed for the read of B to complete. Therefore, the above solution indeed guarantees that this write completes with respect to P2 sometime before the read of A is allowed to complete.

Reasoning about the correctness of chains with the above optimization is much simpler than with the optimization discussed in the previous section. Consider the example chain from Figures 5.22. Figure 5.25 shows the reasoning for the chain pictorially. The dashed arrow labeled (I) can be shown to be true by contradiction (with similar arguments to those used for the example in Figure 5.23). Given that the write of A commits with respect to P2 before the write of B is committed with respect to all processors, flushing the incoming queue ensures that the write of A also completes with respect to P2 before the read of A. Compared to the reasoning steps in Figure 5.23, there is no need to introduce the miss operations in this case because enforcing a program order already forces incoming invalidations to be serviced; Appendix O shows another example to illustrate the above point.

[51] The only requirement is to service the invalidation and update requests that are already in the incoming queue.

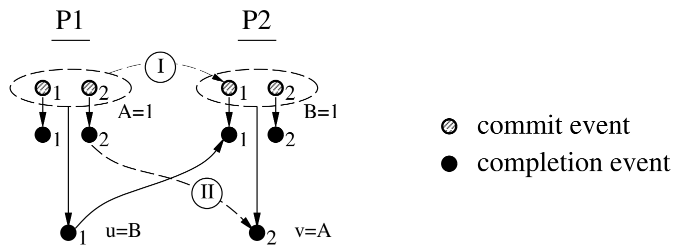


Figure 5.25: Multiprocessor dependence chain with R $\xrightarrow{co}$ W.

In contrast to the solution discussed in the previous section, the efficiency of the above solution is heavily dependent on the model. For SC, the incoming queue must be flushed after almost every memory operation since virtually all program orders are enforced. The frequency of flushes can be much less in a model such as PL1. The only program orders that must be enforced for multiprocessor dependence chains are Xc $\xrightarrow{po}$ Yc, Rc $\xrightarrow{po}$ Y, and X $\xrightarrow{po}$ Wc, making it sufficient to flush the queue after a competing read, before a competing write, and between a competing write followed by a competing read.[52] More conservatively, the queue can be flushed before and after every competing operation. Even so, the frequency of flushes will be proportional to the frequency of competing operations which is low in most applications.

There are a number of optimizations to the above solution that can further reduce the frequency of checking for or flushing incoming invalidation and update requests. The *first* optimization is based on the following observation: it is sufficient to force a flush only on program orders where the second operation is a read (i.e., X $\xrightarrow{po}$ R). Therefore, SC can be implemented by only forcing a flush before every read. For PL1, the relevant program orders are Rc $\xrightarrow{po}$ R and Wc $\xrightarrow{po}$ Rc. The first program order can be satisfied by flushing after every competing read. To satisfy the second program order, the implementation can aggressively perform a flush only between a competing write followed by a competing read, or more conservatively do the flush either before every competing read or after every competing write (e.g., if competing writes are less common).

The *second* optimization uses the intuition behind the solution described in the previous section to further reduce the frequency of checking or flushing the incoming queue. Specifically, there is no need to check or flush the incoming queue if the processor (i.e., first level cache) has not serviced any incoming replies (or incoming update requests) since the last flush. For a model like SC, this optimization can substantially reduce the frequency of checking and possibly flushing the incoming queue; the number of flushes goes from being proportional to the number of reads to being proportional to the number of cache misses (and updates). The optimization is not as effective for a model like PL1 since flushes are already infrequent (e.g., likely to be less frequent than cache misses); therefore, there is a high likelihood that the processor will have serviced an incoming reply (or incoming update request) since the last flush.

Finally, the *third* optimization tries to make the flushes more efficient when the first operation of the enforced program order is a cache miss (i.e., X $\xrightarrow{po}$ R is enforced and X is a miss). The conservative way to achieve the flush is to first wait for the processor to receive the reply for X and to then service any invalidation or update requests that are present in the incoming queue. A slightly more aggressive option is to flush of the queue while the processor waits for the reply to X. The implementation must still check for and flush any new invalidation or update requests that enter the incoming queue before the reply gets back; however, there

[52]The design can maintain a single bit that keeps track of whether a competing write has occurred and can clear the bit every time a flush is done. This bit can be used to determine whether a flush is necessary before a competing read.

may often be no new invalidations and updates during this period. A third option is to specially designate the reply for X to disallow it from bypassing incoming invalidation or update requests that are queued before it. This last option alleviates the need for explicitly flushing the queue in cases where the first operation in the program order pair misses. For a model such as SC, every reply must be specially designated to disallow bypassing. The optimization is more effective for a model like PL1 since it is sufficient to only designate replies to competing reads and competing writes in this way.

The above techniques can be easily extended to deal with multi-level cache hierarchies. The conservative approach would be to flush any incoming invalidation or update requests within the whole hierarchy every time a relevant program order is enforced. Extending the optimizations described above can lead to more efficient solutions. For example, the second optimization can be used as follows: if a cache at level *i* has not serviced an incoming reply or update request since the last flush of the incoming queues below it, the implementation can avoid checking or flushing those queues.

Related Work on Servicing Incoming Messages when Enforcing Program Order Dubois et al. propose an implementation of release consistency called delayed consistency [DWB+91] that delays servicing the invalidations at a destination cache until the destination processor completes an acquire operation. This idea is similar to the idea of flushing an incoming queue when enforcing certain program orders. In contrast to our work, however, the delayed consistency work assumes that locations used for synchronization are disjoint from locations used for data and seems to implicitly assume that a different mechanism is used to maintain the order among synchronization operations. For example, to correctly support the RCsc model,[53] the implementation must service all committed invalidations not only after an acquire, but also in between a release followed by an acquire. The work on delayed consistency does not expose the need for servicing invalidations in the latter case. In contrast, our work clearly outlines the requirements for correctness and explores several important optimizations.

Comparing the Two Early Acknowledgement Techniques

The relative efficiency of the techniques discussed in the previous two sections is extremely model dependent. The first technique is better suited for strict models such as SC where the frequency of enforcing a program order (for supporting multiprocessor dependence chains) is much higher than the frequency of cache misses. On the other hand, the second technique is better suited for more relaxed models where enforcing a program order occurs less frequently. Both techniques provide fast service of incoming requests; for more relaxed models, the second technique may provide faster servicing of incoming replies by allowing them to bypass previous invalidate and update requests.

Multi-level cache hierarchies may reduce the relative advantage of using the second technique for supporting relaxed models. Even though flushing of the incoming queues may be infrequent, it may be cumbersome to check and clear incoming queues throughout the hierarchy. This motivates a hybrid design that employs the first technique at the lower levels and the second technique at the higher levels of the cache hierarchy. Consider a two-level cache hierarchy with an incoming queue for each level. The incoming queue to the second level cache can use the first technique to disallow replies or update messages from overtaking previous

[53] As in the delayed consistency work, the distinction between sync and non-sync operations is ignored (i.e., all operations are treated as a sync).

invalidations or updates. This alleviates the need to explicitly flush this incoming queue. Meanwhile, the incoming queue to the first level can use the second technique to relax the ordering from replies to previous incoming messages by requiring explicit flushes at program ordering points. The above hybrid solution has a couple of other desirable properties. The number of second level misses is typically much smaller than the number of first level misses, making the first technique more suitable at the lower levels. Furthermore, since the first level cache is typically occupied by processor accesses, there is a higher possibility of build up of incoming invalidations and updates in its incoming queue. Therefore, allowing incoming replies to bypass previous incoming messages can have a bigger impact at the higher level caches.

Section 5.5 describes other designs based on restricted network topologies where the above techniques can be used to efficiently support early acknowledgements.

5.4.2 Simple Automatic Hardware-Prefetching

Section 5.3.5 described the conventional mechanism for enforcing program order whereby the servicing of an operation is delayed until a previous operation completes (or is committed in case of writes with early acknowledgements) with respect to all processors. The prefetching technique described in this section provides one method for increasing performance by partially proceeding with the servicing of the an operation while the previous operation is still pending. The following subsections describe this prefetch technique and provide insight into its strengths and weaknesses.

Description of the Prefetch Technique

Prefetching can be classified based on whether it is *binding* or *non-binding*, and whether it is controlled by *hardware* or *software*. With a binding prefetch, the value of a later reference (e.g., a register load) is bound at the time the prefetch completes. This places restrictions on when a binding prefetch can be issued, since the value will become stale if another processor modifies the location during the interval between prefetch and reference. Hardware cache-coherent architectures can provide prefetching that is non-binding. With a non-binding prefetch, the data is brought close to the processor (e.g., into the cache) and is kept coherent until the processor actually reads the value. Thus, non-binding prefetching does not affect correctness for any of the consistency models and can be simply used to boost performance. The technique described in this section assumes *hardware-controlled non-binding prefetch*. Section 5.8 covers the interaction between prefetching and memory models in more detail.

Automatic prefetching can be used to enhance performance by partially servicing large latency accesses that are delayed due to program ordering constraints. For a read operation, a *read prefetch* can be used to bring the data into the cache in a clean state while the operation is being delayed for ordering purposes. Since the prefetch is non-binding, we are guaranteed that the read operation will return a correct value once it is allowed to complete, regardless of when the prefetch completed. In the majority of cases, we expect the result returned by the prefetch to be the correct result. The only time the result may be different is if the location is written to between the time the prefetch returns the value and the time the read is allowed to complete, in which case the prefetched copy will either be invalidated or updated appropriately.

For a write operation, a *read-exclusive prefetch* can be used to acquire exclusive ownership of the line, enabling the write to be serviced quickly once it is allowed to complete.[54] A read-exclusive prefetch is only applicable for invalidate protocols; in an update protocol, it is difficult to partially service a write operation without making the new value visible to other processors.[55] Similar to the read prefetch, the line is invalidated if another processor writes to the location between the time the read-exclusive prefetch completes and the actual write operation is allowed to proceed. In addition, exclusive ownership is surrendered if another processor reads the location during this time.

Implementation of the Prefetch Technique

The conventional way to enforce program ordering is to delay the issue of an operation in the processor's buffer until certain previous operations complete. Prefetching can be incorporated in this framework by having the hardware automatically issue a prefetch (read prefetch for reads and read-exclusive prefetch for writes) for operations that are in the processor's read or write buffer, but are delayed due to program ordering constraints.

The prefetch request behaves similarly to an ordinary read or write request from the memory system point of view. The prefetch first checks the cache hierarchy to see whether the line is already present in the appropriate state. If so, it is discarded. Otherwise, the prefetch is issued to the memory system. When the prefetch reply returns to the processor, it is placed in the cache. While a prefetch is outstanding, future requests to the same line can be combined with the prefetch request (using techniques similar to those described in Section 5.2.3) so that a duplicate request is not sent out.

One of the issues with prefetching is that the caches will be more busy since references that are prefetched access the cache twice, once for the prefetch and another time for the actual reference. As previously mentioned, accessing the cache for the prefetch request is desirable for avoiding extraneous traffic. Overall, we do not believe that the double access will be a major issue since prefetch requests are generated only when ordinary operations are being delayed due to consistency constraints, and it is likely that there are no other requests to the cache during that time.

Lookahead in the instruction stream is also beneficial for hardware-controlled prefetch schemes. Aggressive lookahead is possible with processors that support dynamic instruction scheduling, branch prediction, and speculative execution past unresolved branches. Such lookahead can provide the hardware with several memory requests that are being delayed in the load and store buffers due to consistency constraints (especially for the stricter memory consistency models) and gives prefetching the opportunity to overlap such operations. The strengths and weaknesses of hardware-controlled non-binding prefetching are discussed in the next subsection.

Strengths and Weaknesses of the Prefetch Technique

Figure 5.26 presents two example code segments to provide intuition for the circumstances where prefetching boosts performance and where prefetching fails. We make the following assumptions to make the examples more concrete. We assume a processor with non-blocking reads and branch prediction machinery. The caches

[54] Depending on the state of the cache line, a read-exclusive prefetch leads to either no request, an exclusive request, or a read-exclusive request.

[55] A two-phase hybrid scheme similar to the one described in Section 5.3.5 may be used for update protocols. In the first phase, the prefetch can invalidate all copies, which completes the write from a consistency point of view. Issuing the actual write leads to updating the same copies; no acknowledgements are needed for these updates.

Example 1		Example 2	
lock L	(miss)	**lock L**	(miss)
write A	(miss)	**read C**	(miss)
write B	(miss)	**read D**	(hit)
unlock L	(hit)	**read E[D]**	(miss)
		unlock L	(hit)

Figure 5.26: Example code segments for hardware prefetching.

are assumed to be lockup-free with an invalidation-based coherence scheme. We assume a cache hit latency of 1 cycle, a cache miss latency of 100 cycles, and a memory system that can accept an access on every cycle. Finally, we assume that no other processes are writing to the locations used in the examples and that the lock synchronizations succeed (i.e., the lock is free).

First consider the code segment on the left side of Figure 5.26. This code segment resembles a producer process updating the values of two memory locations. Given a system with sequential consistency, each operation is delayed for the previous operation to complete. The first three operations miss in the cache, while the unlock operation hits due to the fact that exclusive ownership was gained by the previous lock operation. Therefore, the four operations take a total of 301 cycles to perform. In a system with a relaxed model such as PL1 or release consistency, the write operations are delayed until the lock operation completes, and the unlock operation is delayed for the write operations to complete. However, the write operations can be overlapped. Therefore, the operations take 202 cycles.

The prefetch technique described in this section boosts the performance of both the strict and the relaxed model. Concerning the loop that would be used to implement the lock synchronization, we assume the branch predictor takes the path that assumes the lock synchronization succeeds. Thus, the lookahead into the instruction stream allows locations A and B to be prefetched in read-exclusive mode. Regardless of the consistency model, the lock operation is serviced in parallel with prefetch of the two write accesses. Once the result for the lock operation returns, the two write accesses will be satisfied quickly since the locations are prefetched into the cache. Therefore, with prefetching, the accesses complete in 103 cycles for both SC and PL1. For this example, prefetching boosts the performance of both SC and PL1 and also equalizes the performance of the two models.

We now consider the second code segment on the right side of Figure 5.26. This code segment resembles a consumer process reading several memory locations. There are three read operations within the critical section. As shown, the read to location D is assumed to hit in the cache, and the read of array E depends on the value of D to access the appropriate element. For simplicity, we will ignore the delay due to address calculation for accessing the array element. Under SC, the operations take 302 cycles to perform. Under PL1, they take 203 cycles. With the prefetch technique, the operations take 203 cycles under SC and 202 cycles under PL1. Although the performance of both SC and PL1 are enhanced by prefetching, the maximum performance is not achieved for either model. The reason is simply because the address of the read access to array E depends on the value of D and although the read access to D is a cache hit, this access is not allowed to perform (i.e., the value cannot be used by the processor) until the read of C completes (under SC) or until the lock access completes (under PL1). Thus, while prefetching can boost performance by pipelining several accesses that are delayed due to consistency constraints, it fails to remedy the cases where out-of-order consumption of return values is important to allow the processor to proceed efficiently.

In summary, prefetching is an effective technique for pipelining large latency references when the reference is delayed for ordering purposes. However, prefetching fails to boost performance when out-of-order consumption of prefetched values is important. Such cases occur in many applications, where accesses that hit in the cache are dispersed among accesses that miss, and the out-of-order use of the values returned by cache hits is critical for achieving the highest performance. The next section describes a speculative technique that remedies this shortcoming by allowing the processor to consume return values out-of-order regardless of the consistency constraints. The combination of prefetching for writes and the speculative execution technique for reads will be shown to be effective in regaining substantial opportunity for overlapping operations even for strict models such as SC.

5.4.3 Exploiting the Roll-Back Mechanism in Dynamically-Scheduled Processors

This section describes the speculative execution technique for read operations. An example implementation is presented in Appendix P. As we will see, this technique is particularly applicable to dynamically-scheduled processors which already support a roll-back mechanism for branch misprediction.

Description of the Speculative Execution Technique

The idea behind speculative execution is simple. Assume u and v are two operations in program order, with u being any large latency access and v being a read operation. In addition, assume the program order constraints for the model require the completion of v to be delayed until u completes. *Speculative execution for read operations* works as follows. The processor obtains or assumes a return value for operation v before u completes and proceeds. At the time u completes, if the return value for v used by the processor is the same as the current value of v, then the speculation is successful. Clearly, the computation is correct since even if v was delayed, the value the operation returns would have been the same. However, if the current value of v is different from what was speculated by the processor, then the computation is incorrect. In this case, we need to throw out the computation that depended on the value of v and repeat that computation. The implementation of such a scheme requires a *speculation mechanism* to obtain a speculated value for the access, a *detection mechanism* to determine whether the speculation succeeded, and a *correction mechanism* to repeat the computation if the speculation was unsuccessful.

Let us consider the speculation mechanism first. The most reasonable thing to do is to complete the read operation and use the returned value. In case the access is a cache hit, the value will be obtained quickly. In the case of a cache miss, although the return value will not be obtained quickly, the operation is effectively overlapped with previous operations in a way similar to prefetching. In general, guessing on the value of the access is not beneficial unless the value is known to be constrained to a small set (e.g., lock operations).

Regarding the detection mechanism, a naive way to detect an incorrect speculated value is to repeat the access when the memory model would have allowed it to proceed under non-speculative circumstances and to check the return value against the speculated value. However, if the speculation mechanism performs the speculative access and keeps the location in the cache, it is possible to determine whether the speculated value is correct by simply monitoring the coherence transactions for that location.[56] Thus, the speculative

[56]The above statement implies that the location is cached. Therefore, the speculative read technique does not apply to uncached operations since no coherence transactions will be sent to the requesting processor.

execution technique can be implemented such that the cache is accessed only once per access versus the two times required by the prefetch technique. Let us refer back to accesses u and v, where the memory model requires the completion of read operation v to be delayed until u completes. The speculative technique allows access v to be issued and the processor is allowed to proceed with the return value. The detection mechanism is as follows. An incoming invalidation or update request for location v before u has completed indicates that the value of the access may be incorrect.[57] In addition, the lack of invalidation and update messages indicates that the speculated value is correct. Cache replacements need to be handled properly also. If location v is replaced from the cache before u completes, then invalidation and update requests may no longer be sent to the cache. The speculated value for v is assumed stale in such a case (unless one is willing to repeat the access once u completes and to check the current value with the speculated value). Appendix P provides further implementation details for this mechanism.

Once the speculated value is determined to be wrong, the correction mechanism involves discarding the computation that depended on the speculated value and repeating the operation and the following computation. This mechanism is almost the same as the correction mechanism used in processors with branch prediction machinery and the ability to execute instructions past unresolved branches. With branch prediction, if the prediction is determined to be incorrect, the instructions and computation following the branch are discarded and the new target instructions are fetched. In a similar way, if a speculated value is determined to be incorrect, the read operation and the computation following it can be discarded and the instructions can be fetched and executed again to achieve correctness.

The speculative technique overcomes the shortcoming of the prefetch technique by allowing out-of-order consumption of speculated values. Given speculative execution, read operations can be issued as soon as the address for the access is known, regardless of the consistency model supported. Referring back to the second example in Figure 5.26, let us consider how well the speculative technique performs. We still assume that no other processes are writing to the locations. Speculative execution achieves the same level of pipelining achieved by prefetching. In addition, the read operation to D no longer hinders the performance since its return value is allowed to be consumed while previous operations are outstanding. Thus, both SC and PL1 can complete the operations in 104 cycles.

The speculative read technique described above can also be easily adopted to efficiently support the initiation condition between a conflicting write followed by a read. Given a write with an unresolved address, the processor can allow reads that follow it to be speculatively issued and completed. Once the address of the write is resolved, it can be matched against the address of the speculative reads. In case of a match, we can use the correction mechanism described above to retry the matching read and any computation that used its value.[58] This optimization is also applicable to uniprocessors since a read is required to return the value of the latest conflicting write that is before it in program order.

Appendix P describes an example implementation of the speculative read technique.

[57] There are two cases where the speculated value remains correct. The first is if the invalidation or update occurs due to false sharing, that is, for another location in the same cache line. The second is if the new value written is the same as the speculated value. We conservatively assume the speculated value is incorrect in either case.

[58] A similar optimization can be used for the Alpha memory model which requires reads to the same address to appear in program order. The latter read can be performed speculatively, and in case the former read is to the same address, either the latter read can be retried or its return value can be supplied to the former read.

Related Work on Speculative Reads

We recently became aware of a couple of related patents in this area. There is an IBM patent from 1993 by Frey and Pederson [FP93] that describes a similar idea to that presented in our earlier publication from 1991 [GGH91b] regarding the use of speculative reads and rolling back in case a correction is needed. An earlier IBM patent from 1991 by Emma et al. [EKP+91] describes a mechanism for detecting speculative reads that may have returned incorrect data. However, the detection mechanism described in the 1991 patent is extremely conservative since it treats a speculative read as incorrect not only on an incoming invalidation but also on *any* cache miss. Such a conservative detection mechanism can substantially reduce the benefits of speculative reads since it essentially prohibits the benefits of overlapping multiple misses and can also lead to frequent false detection and rollback. Neither patent describes the automatic write prefetching technique which complements speculative reads in an advantageous way.

5.4.4 Combining Speculative Reads with Hardware Prefetching for Writes

The combination of speculative reads with hardware prefetching for writes provides significant opportunity to overlap memory operations regardless of the memory model supported. Consequently, the performance of any consistency model, even the most relaxed models, can be enhanced. More importantly, the performance difference among the various models is reduced. This latter result is noteworthy in light of the fact that relaxed models are accompanied by a more complex programming model.

The main idea behind the prefetch and speculative read techniques is to service operations as soon as possible, regardless of the constraints imposed by the memory model. Of course, since correctness needs to be maintained, the early (and speculative) service of the operations is not always useful. For these techniques to provide performance benefits, the probability that a prefetched or speculated value is invalidated (or updated, in case of a speculative read) must be small. There are several reasons for expecting such invalidations to be infrequent. Whether prefetched or speculated locations are invalidated loosely depends on whether it is critical to delay such operations to obtain a correct execution. If the supported consistency model is a relaxed model such as PL1, delays are imposed only at synchronization points. In many applications, the time at which one process releases a synchronization is long before the time another process tries to acquire the synchronization. This implies that no other process is simultaneously accessing the locations protected by the synchronization. Correctness can be achieved in this case without delaying the accesses following a competing read until the read completes or delaying a competing write until its previous operations complete. For cases where the supported consistency model is strict, such as SC, the strict delays imposed on operations are also rarely necessary for correctness. As discussed in Chapter 3, the actual frequency of competing operations is quite low in most programs. Therefore, most of the delays imposed by a typical SC implementation are superfluous. Nevertheless, it is important to substantiate the above observations in the future with extensive simulation experiments.

Both the speculative read and the write prefetching techniques provide higher performance gains (relative to a less aggressive implementation) when there is a higher frequency of long latency operations (i.e., cache misses). With infrequent long latency operations (i.e., large number of instructions separating each cache miss), limitations in buffer resources often disallow the processor from overlapping multiple long latency operations due to the limited lookahead capability. Fortunately, such applications already perform efficiently and optimizing the communication latency is less important.

A major implication of the techniques proposed is that the performance of different consistency models is somewhat equalized once these techniques are employed. Therefore, the choice of the consistency model to be supported in hardware becomes less important if one is willing to implement these techniques. The cost, of course, is the extra hardware complexity associated with the implementation. While the prefetch technique is simple to incorporate into cache-coherent multiprocessors, the speculative execution technique requires more sophisticated hardware support. Nevertheless, the mechanisms required to implement speculative execution are present in many current and next generation commercial processors. In particular, we showed how the speculative technique could be incorporated into one such processor design with minimal additional hardware.

Dynamically scheduled processor designs have gained enormous popularity since 1991 when we originally published the ideas on speculative reads and hardware prefetching for writes [GGH91b]. Furthermore, to provide more efficient support for stricter memory models, several of the next generation commercial processors have actually adopted either one or both of these techniques in their designs. The Metaflow Thunder Sparc processor claims multiple speculative memory reads while maintaining "strong consistency" (probably means TSO in the context of the Sparc architecture) [Lig94]. The MIPS R10000 uses both speculative reads and hardware prefetching for writes to support sequential consistency. Finally, Intel's Pentium Pro (or P6) seems to be using at least the speculative read technique for supporting sequential consistency.

5.4.5 Other Related Work on Aggressively Supporting Multiprocessor Dependence Chains

There have been a number of other techniques proposed for aggressively supporting multiprocessor dependence chains that do not directly relate to any of the optimizations discussed in this section. In what follows, we will focus on three separate implementations proposed by Adve and Hill for supporting either the sequential consistency or the data-race-free models [AH90a, AH90b, AH93]. All three implementations exploit the aggressive form of the specifications since they do not maintain completion or commit order at the intermediate points in a chain.

Adve and Hill [AH90a] have proposed an implementation for sequential consistency that is potentially more efficient than conventional implementations. The scheme depends on an invalidation-based cache coherence protocol with write back caches and eager exclusive replies. Instead of delaying the next operation until a previous write commits or completes with respect to all processors, the more aggressive implementation delays the next operation only until the previous write is serialized (i.e., gets ownership). In case the next operation is a write, the new value written by the write is not made visible to other processors (or written back) until all previous writes by this processor have completed. In case the next operation is a read, the processor can proceed to complete the read either if the read is a cache miss or if the read is a cache hit to a dirty copy. However, a read to a clean copy may not proceed until all previous writes are complete; the alternative is to always treat the read as a cache miss. The gains from this optimization are expected to be limited because of the following reasons: (a) only writes to lines with other shared copies benefit from the optimization, (b) the latency for obtaining ownership may be only slightly smaller than the latency for the write to commit with respect to all processors, and (c) a following read to a clean cache line cannot be serviced quickly. Furthermore, as we discussed in the previous sections, using eager exclusive replies for strict models such as SC complicates the implementation without much performance benefit. Finally, holding back visibility of a line in a cache for previous operations to complete requires complex hardware support. In contrast, the combination of the early acknowledgement, write prefetching, and speculative read techniques discussed in this section can lead to a substantially higher performance implementation of SC.

Adve and Hill have also proposed an implementation for the data-race-free-0 (DRF0) model that exploits the aggressive specification of multiprocessor dependence chains [AH90b]. Instead of delaying the service of a synchronization write (analogous to a competing write in PL1) until previous data operations are complete, the processor proceeds to service the synchronization write by obtaining ownership for the line and performing the write. However, a synchronization read (analogous to competing read in PL1) from any processor must be disallowed from observing the value of this synchronization write until the operations preceding the write complete. A synchronization write from any processor or the writing back of the line must also be delayed in a similar fashion. To avoid deadlock, however, data requests to this line from other processors must be serviced using the "remote service" mechanism [Adv93] which involves servicing a read or write from another processor without giving up ownership for the line. For a read, an uncachable copy of the selected word is sent back to the requesting processor. For a write, the new data must be merged into the current copy at the destination cache and only an acknowledgement is sent back, thus disallowing the requesting line from obtaining either a clean or dirty copy of the line. Overall, the performance gain from using the above technique over a conventional design is likely to be minimal. A conventional design uses the write buffer to delay synchronization writes, allowing the processor to proceed without stalling. Since this latency is already not visible to the processor, servicing the write slightly earlier should not affect performance much. Therefore, the performance advantage will most likely not justify the substantial complexity in supporting the remote service mechanism and holding back visibility of the write.

Finally, the most aggressive implementation proposed by Adve and Hill is for supporting the data-race-free-1 (DRF1) model [AH93]. This implementation is more aggressive than the one described in the previous paragraph because a synchronization write is not only allowed to be serviced early, but a synchronization read from another processor is allowed to read the new value before operations previous to the synchronization write are complete. To achieve correctness, the writing processor informs the reading processor of any of its incomplete operations before the synchronization write. In turn, the reading processor must ensure not to access the locations that are on the incomplete list until those operations complete. In effect, this is a very aggressive way of implementing a multiprocessor dependence chain by enforcing completion orders only at the two end points in the chain. However, supporting this functionality in hardware is overwhelmingly complex and does not map well to current system designs. Furthermore, the gains from these optimizations will likely be minimal in efficient hardware designs.

Overall, the implementation techniques suggested for DRF0 and DRF1 are more illustrative of the fact that program-centric specifications allow more aggressive implementations compared to hardware-centric specifications such as WO or RCsc. However, the techniques do not seem to provide a substantial performance gain over conventional hardware designs and may in fact be impractical due to their complexity. As we will discuss in Section 5.6, techniques similar to those proposed by the DRF1 implementation can be beneficial in the context of software-based shared memory systems where the complexity is manageable and the extra flexibility can actually provide noticeable performance gains. In fact, some of the techniques used in designs such as LRC [KCZ92, DKCZ93] are more aggressive than the DRF1 implementation.

5.5 Restricted Interconnection Networks

A restricted network such as a bus provides lower connectivity among nodes as compared to a general network, which typically translates into lower scalability and performance as well. For this reason, restricted networks are often used for connecting only a small number of nodes. There are a number of different restricted network topologies including buses, rings, and hierarchies of the above. Restricted networks may also be used in combination with general networks; for example, the DASH design uses a two-dimensional mesh network to connect multiple clusters, with each cluster consisting of multiple processors connected by a bus.

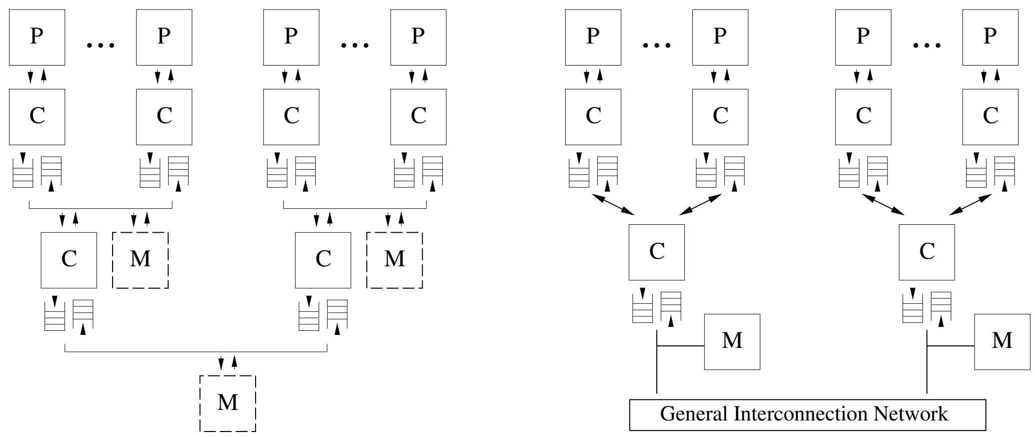
From a memory model implementation perspective, the lower connectivity in restricted networks typically translates to more ordering guarantees among messages. This section describes how such ordering guarantees may be exploited to achieve simpler and more efficient designs. As we will see, such designs benefit from the aggressive form of the system requirement specifications presented in Chapter 4.

5.5.1 Broadcast Bus

Broadcast buses are the most popular form of restricted networks. The key characteristics of a bus that can be exploited for implementing a cache consistency protocol are: (a) a message is seen by all nodes on the bus, and (b) the bus serializes all messages into a single total order. This section describes how the above characteristics affect the design assuming a snoopy protocol; the discussion can also be extended to a directory protocol implemented on a bus.

A broadcast bus greatly simplifies the support for multiprocessor dependence chains relative to the more general techniques described in Section 5.3.5. Consider the way write operations are handled assuming an invalidation-based protocol. Virtually all bus-based designs use the early acknowledgement optimization discussed in Section 5.4.1. There are two types of requests generated by a write: an exclusive or a read-exclusive request. The exclusive request commits with respect to all processors as soon as it appears on the bus; it immediately turns into an incoming invalidation request at every cache. The exclusive reply is generated implicitly by the requester's bus interface and does not appear on the bus; the early invalidation acknowledgement is implicitly piggy-backed on this reply. The outgoing read-exclusive request behaves differently. This request turns into an incoming read-exclusive request at other processors' caches and at the memory. The cache (or memory) that is the current owner of the line responds with a read-exclusive reply; non-owner caches treat the read-exclusive request simply as an incoming invalidation request. The read-exclusive request is guaranteed to be committed with respect to all processors by the time its reply appears on the bus. Again, the invalidation ack is implicitly piggy-backed with the read-exclusive reply. Writes are handled in an analogous way in update protocols.

Compared to the general case described in Section 5.3.5, detecting write completion is much simpler since a single implicit invalidation acknowledgement signals that the write is committed with respect to every processor. Therefore, there is no need to keep a count of pending invalidations for each outstanding write. Furthermore, exclusive and read-exclusive replies are inherently delayed (i.e., they are not eager); they implicitly signal that the write is committed with respect to all processors. Supporting category three multiprocessor dependence chains is also simplified greatly because writes are inherently atomic with respect to multiple copies due to the broadcast nature of the bus: a write is committed with respect to all processors before a read from another processor can read its value. This same behavior also holds for updates, making



(a) bus hierarchy **(b) hybrid design**

Figure 5.27: Bus hierarchies and hybrid designs.

it trivial to support category three chains with update writes.

Due to the lack of multiple paths and lack of message reordering, a bus-based design can also significantly reduce the number of subtle corner cases relative to those in a general network (see Appendix K). Therefore, the design can be simplified by maintaining sufficient order among messages (to the same line) in the cache hierarchy and performing sufficient cross checks among the incoming and outgoing queues.[59] For example, there is no need for negative acknowledgement replies, or the corresponding mechanisms for retrying requests, in most bus designs. Single transaction buses, which allow only a single outstanding request on the bus, eliminate more transient cases especially those resulting from simultaneous operations on the same line. For example, it is inherently impossible for incoming invalidation or update requests (from an earlier or later write) to arrive at a cache while it has an outstanding request to the same line. This alleviates the need for supporting structures for detecting the transient conditions described in Appendix K. Split-transaction buses, which allow multiple outstanding requests on the bus, can guarantee an analogous behavior by disallowing a new request from issuing on the bus if it is to the same line as a previous outstanding request from any processor.

5.5.2 Hierarchies of Buses and Hybrid Designs

Hierarchical designs based on restricted networks also exhibit special message ordering guarantees. Figure 5.27(a) shows an example design with a hierarchy of buses, while Figure 5.27(b) shows a hybrid design with a small number of processors connected hierarchically to form a cluster and clusters connected using a general network. The following describes how the message ordering guarantees can be exploited for efficiently supporting various memory models.

A hierarchical design such as the one shown in Figure 5.27(a) shares a lot of similarities to a cache hierarchy (except that a given cache may be shared by multiple processors). As in a cache hierarchy, the subset property is often enforced so that a higher level cache contains a strict subset of the data in a lower level cache. Incoming and outgoing messages are appropriately filtered by the caches in the hierarchy. An

[59] Of course, there is a trade-off since cross checking and maintaining a stricter order within the hierarchy can themselves lead to higher complexity and lower performance.

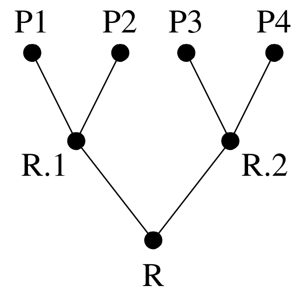


Figure 5.28: Simple abstraction for a hierarchy.

outgoing processor request traverses down the hierarchy until it is serviced, which may sometimes involve traversing up in a separate trunk to find the data. As shown in the figure, physical memory may be placed either close to the processors or near the bottom of the hierarchy.

Figure 5.28 shows a simple abstraction of the hierarchy. We use the concept of a root node for a given memory line to be the highest parent node in the hierarchy that covers the set of nodes that have a copy of the line. For example, if P1 and P2 are the only two processors that have a copy of a memory line, R.1 is the root node for the line. However, if P3 also maintains a copy, then the root node is R. The root node designation for a line may changes as copies of the location are dynamically created and eliminated within the hierarchy; the cache states at each level provide sufficient information to determine the root node as a request traverses down the hierarchy.

A key implementation issue in a hierarchical design such as the one described above is how writes are committed with respect to other processors. Consider the case where a location is cached by multiple processors. For simplicity, assume the writing processor already has a clean copy of the line. To invalidate other copies of the line, the outgoing exclusive request traverses down towards the root node for the line. On receiving this request, the root node emanates incoming invalidation requests to all the child nodes that may have a copy. The write can be considered committed with respect to all processors at this point, and the exclusive reply that is sent back to the requesting node implicitly signals the acknowledgement for the invalidations. Since the above exploits the early acknowledgement technique, extra order must be maintained among incoming messages (i.e., those moving upward in the hierarchy) to correctly support multiprocessor dependence chains. The constraints on incoming messages are virtually identical to those for an ordinary cache hierarchy as discussed in Section 5.4.1: the order from an incoming reply message to a previous incoming invalidate request must be maintained. Furthermore, the writing processor must disallow other processors from observing the new value until it receives the implicit acknowledgement reply from the root. This latter condition is especially important for supporting models that maintain category three multiprocessor dependence chains.

The following provides some intuition for why the above implementation behaves correctly. Consider a processor Pi that communicates with a processor Pj. Such communication involves a message that is guaranteed to traverse from the common root of Pi and Pj up towards Pj. By maintaining the appropriate order among incoming messages, any incoming invalidation messages that had previously arrived at the common root are guaranteed to be pushed towards Pj by the later communication message. Therefore, any writes that were committed with respect to Pi (before Pi sends out the message to Pj) also get committed with

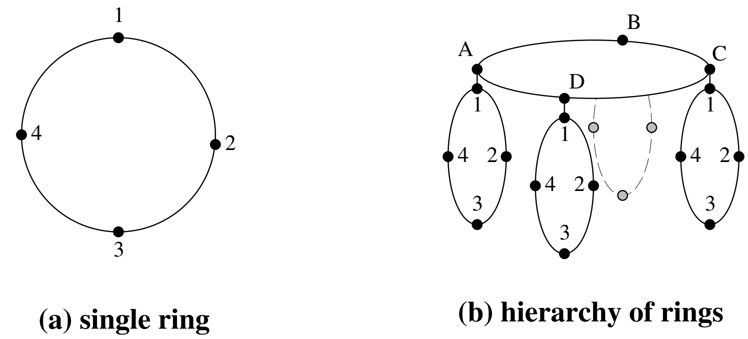


Figure 5.29: Rings and hierarchies of rings.

respect to Pj and any other nodes that are traversed on the way from the common root to Pj.

Update protocols can be implemented in an analogous manner, with the requirement that an incoming reply or an incoming update request must be ordered with respect to a previous incoming invalidate or update request. The implementation for invalidation-based protocols can be modified slightly to allow incoming invalidate requests to be sent to the child nodes as the outgoing exclusive request traverses down the hierarchy toward the node (instead of delaying the sending of invalidations until the request reaches the root node), with the acknowledgement still generated at the root. However, this alternative is not applicable to update requests since the order among updates to the same location is important and the root node plays an important role in serializing the updates.

Even though restricted networks such as hierarchies of buses do not scale well, they can be effective for connecting a small number of processors together. For this reason, some designs may use a hybrid mix of restricted and general networks as shown in Figure 5.27(b). In such a design, each cluster of processors connected with a restricted network can be treated as a single cache hierarchy, with the difference that some inter-processor communication occurs completely within a cluster. Outgoing exclusive requests that reach the root node at the interface with the general network lead to invalidation messages that are sent to other clusters with a copy. Each invalidation request is acknowledged as soon as it is propagated (through the general network) to the bottom root node of its destination cluster and the requesting cluster receives a final invalidation acknowledgement (implicit in case of delayed exclusive replies) after all invalidations are acknowledged. For update-based protocols, supporting category three multiprocessor dependence chains is difficult in a hybrid design for the same reasons discussed in Section 5.3.5.

5.5.3 Rings

A ring is another example of a restricted network whose ordering properties can be exploited to efficiently support a memory model. Figure 5.29(a) shows an example of a single ring, while Figure 5.29(b) shows multiple rings connected into a hierarchy. A ring may either be unidirectional or bidirectional; a unidirectional ring requires all messages to traverse the ring in the same direction (e.g., clockwise). The discussion below assumes rings with a snoopy coherence protocol; many of the issues apply to directory protocols as well.

Invalidation-based protocols can be implemented on a ring using conventional techniques. For example, assume an exclusive request is issued from node 3 in the ring in Figure 5.29(a). The exclusive request travels around the ring (e.g., from 3 to 4 to 1 to 2 to 3) and commits an incoming invalidation at each node along the way. Therefore, by the time the request gets back to node 3, the write has been committed with respect to all nodes on the ring. Node 3 must disallow any other processor from reading the value of its write until the exclusive request makes the loop around the ring. No other message ordering constraints are required and except for invalidations that must make the full loop, other messages may travel in either direction on the ring.

Update-based protocols are a little bit more challenging. To implement updates, a single node must be chosen as the root node in the ring. The root node acts as a serialization point for update requests. Assume node 1 is the root node and consider an update request generated by node 3. The update request must first travel from node 3 to the node 1 without affecting any nodes along the way. Once the request reaches the root node, it traverses the ring once in a single direction (i.e., same direction as other updates emanated by the root node) and generates an incoming update request at every node along the way. Update requests are not allowed to bypass one another as they loop back towards the root node. Node 3 can consider the update request acknowledged when the request reaches it; it should also not provide the new value to any other processors until this happens. Note that node 3 considers the update acknowledged even though the update has not yet been committed with respect to node 4. Correctness is still maintained because of the following constraints on reply messages. Assume all update requests travel in the clockwise direction; therefore, starting at the root, the node numbers as depicted in Figure 5.29(a) increase in this direction. To ensure correctness, a reply that is generated by node i and destined for node j must travel from node i to node j in the same direction as update requests (i.e., clockwise) if $i < j$. Furthermore, reply messages cannot overtake update requests in the clockwise direction. The above ensures that a reply from node i pushes all updates seen by node i to node j and any other nodes along the way. The reason the first constraint is not necessary when $i > j$ is because any updates received by i have already been received by j given j is "upstream" from i. Therefore, a reply from node 4 to node 3 can indeed exploit the shorter path by traveling counter-clockwise. The above technique can also be used to implement invalidation-based protocols; however, the implementation described in the previous paragraph is simpler and more efficient since it does not impose any message ordering constraints around the ring.

The above techniques can be easily extended to a hierarchy of rings. Figure 5.29(b) shows a two-level hierarchy with four rings at the first level of the hierarchy connected by a single ring at the second level. Within the rings at the first level, the root node is the node that interfaces to the second level (node 1 in each ring). In addition, one of the nodes at the second level must be chosen as the root node at that level; assume node A is the root in the discussion below. Just as in a cache hierarchy, messages can filtered at the interface between the rings to decide whether the message must be sent up or down to the next level. Consider an exclusive or update request issued by node 2 on the rightmost first level ring (connected to node C at the second level). This request must first traverse to the root node (node 1) on its own ring without affecting any nodes on its path. At the root node, the implementation must determine whether any copies of the location exist outside this ring. If there are no such copies, then the same technique described above for a single ring can be used for handling the request. Otherwise, the request must go onto the second level ring and travel to the root node at that level (node A) again without affecting any nodes on its path. Once it reaches the root node, the update or invalidate request must traverse the nodes on the second level ring in a single direction (i.e., same direction as any other requests emanated from the root node). Assume the direction is clockwise for this discussion. As the request visits each node, the interface at the node must decide if an update or invalidate request must be generated for the first level ring attached to that node based on whether any copies of the line are present on that ring. When a request is sent to the first level ring, the root node (node 1) on that ring sends the request around in a single direction (again assume this is clockwise). The requesting node can consider the write acknowledged when the request makes it back to it and it must hold back the new value from other processors until this event occurs. The following constraints must be upheld for correctness. Update requests that emanate from a root node on any ring may not bypass previous invalidate or update requests as they travel clockwise on that ring. As before, there is also a constraint on reply messages that is a simple extension of the requirements in a single ring. If a reply is destined from node i to node j on a given ring, it has to travel clockwise if $i < j$. In addition, it cannot bypass any previous invalidation or update requests on the ring as it travels clockwise. For example, if a reply is sent from node 2 on ring A to node 2 on ring C, the reply can travel counter-clockwise to node 1 on ring A, then clockwise from node A to node C on the level 2 ring (since A < C), and then clockwise from node 1 to node 2 on ring C. Again, the intuition behind the above requirement is that replies appropriately push previous invalidate and update requests as they go around the rings.

5.5.4 Related Work on Restricted Interconnection Networks

There are a number of previously proposed schemes for exploiting the extra ordering guarantees in restricted interconnection networks to implement various memory models. We briefly cover a few of these schemes below.

Landin et al. [LHH91] propose implementations of the SC and PC models on so called *race-free networks*. They define a race-free network as follows:

- A race-free network is a network with the topology of any acyclic undirected network graph.
- Transactions propagate on the arcs in the network without the possibility of overtaking each other.
- Transactions may be buffered in the network nodes but buffers must maintain a strict FIFO order between transactions.

Landin et al. describe three different implementations. The first is an implementation of SC and is similar to the implementation for hierarchies of buses described in Section 5.5.2: a write request traverses to the root node for the memory line and sends invalidations to all copies from the root; the root node also sends an acknowledge reply to the writing processor. The second implementation supports the PC model. This implementation takes advantage of the order among outgoing messages to pipeline writes. After issuing a write that misses, the processor can issue a subsequent write before the previous write is acknowledged. The write is acknowledged by the root in a similar way to the first scheme, and the value of the write is not made visible to any other operations until the write is acknowledged.[60] We do not describe the third implementation proposed by Landin et al. since it does not enforce the coherence requirement for writes, and therefore fails to support the semantics of the PC model as it is defined in our work.

One problem with the concept of race-free networks is its restrictiveness. For example, even processors connected on a shared bus do not satisfy the required restrictions if the cache hierarchy is considered as part

[60]The definition of PC used by Landin et al. for this implementation is stricter than that used in this thesis because it maintain category 3 chains. Even with the stricter definition, they could relax the above condition to allow the write to be visible to reads from its own processor.

of the network. This is because most commercial processors use multiple paths for incoming and outgoing messages within the cache hierarchy which allows messages to get reordered; for example, an outgoing reply is often allowed to bypass previous outgoing requests from within the cache hierarchy. Therefore, an implementation such as the one for PC that depends on all messages being maintained in order on the outgoing path fails to work; the processor must wait for the write to at least reach the bus before issuing the next write. Landin et al. correctly observe that the first implementation of SC is robust in this respect and does not depend on FIFO ordering among outgoing messages. However, in contrast to our solution, they still impose FIFO ordering among incoming messages (i.e., sent from a root node to its children).

Our suggested implementations for buses or hierarchies of buses are more general than the race-free work in several ways. The primary difference arises from the observation that FIFO ordering is not necessary in either the outgoing or the incoming path. For example, the only requirement on the incoming path (for an invalidation protocol) is to maintain the order from an incoming reply to a previous incoming invalidate request. This makes the implementation technique applicable to a much larger set of designs where FIFO ordering may not be maintained. Furthermore, the techniques described here are applicable to any of the relaxed models described in this thesis.

Several researchers have studied the ordering properties in rings for supporting various memory consistency models. Collier [Col92] describes a conceptual single ring system that maintains a full copy of memory at every node and uses updates to provide coherence among the copies. Collier observes that a root node can be used to serialize all writes by requiring updates to traverse in the same direction, and that the writing processor can consider the write "complete" when the update message reaches it. However, because each node is assumed to maintain a full copy of memory, Collier's solution does not deal with any read requests or corresponding read replies on the ring. In contrast, our solution for updates on a single ring is more general and prescribes the ordering constraints between updates requests and other messages such as read replies that are required for correctness in a realistic system. Furthermore, our solution for invalidations in a single ring environment is more efficient since it does not require writes to be serialized at a root node. Keith Farkas et al. [FVS92] describe a protocol for supporting sequential consistency in a hierarchy of rings in the context of the Hector multiprocessor. Their design assumptions are quite different from the implementations discussed here; furthermore, their design does not fully exploit the ordering properties of rings for supporting update protocols.

5.6 Systems with Software-Based Coherence

As an alternative to hardware cache coherence, loosely coupled multicomputers, such as workstations connected by a fast local area network, may support the single address space model in software and provide caching through replication of shared data in the local main memory of each node. To amortize the high software overhead and large latency of communication, loosely coupled systems require transfer of data to be done at larger granularities. Thus, the granularity for communication is typically chosen to be either that of *fixed size* pages of several kilobytes or large *variable size* objects. The flexibility provided by relaxed memory models can be used to boost performance in both types of designs. Compared to hardware designs, software designs exhibit an extremely large overhead and latency for communication. Therefore, in contrast to hardware designs where relaxed models are primarily used to hide the latency of communication, software designs also exploit relaxed models to both delay and reduce (e.g., through coalescing) the number of communicated messages. Below, we provide a brief overview of several software systems.

Ivy is among the first systems that implemented a page-based distributed shared memory (DSM) protocol [LH89]. Ivy allows pages to be replicated among multiple readers and exploits conventional memory management hardware to maintain coherence among the multiple copies. The protection status for each page is manipulated to trigger faults on references to invalid pages and on writes to read shared pages. The fault handlers in turn generate the appropriate coherence operations. Ivy maintains the sequential consistency model which allows the system to execute any shared memory program without modifications. However, the large coherence granularity of pages makes *false sharing* a major problem in Ivy.

The false sharing problems in page-based designs can be partially alleviated by exploiting relaxed memory models. For example, Munin [CBZ91] exploits a model similar to release consistency to efficiently support multiple writers to a single page by delaying the propagation of invalidation or update transactions due to writes until a release synchronization (e.g., unlock). In a hardware design, this optimization is conceptually analogous to gathering write operations in a write buffer and flushing the buffer on a release. To ensure that the page returns to a consistent state, each processor is required to keep track of the changes it makes to its local copy. Munin accomplishes this by duplicating a page before modifying it and comparing the dirty page to the original duplicate. These computed changes or *diffs* are then communicated to other copies at the next release synchronization. Delaying the communication of write operations and coalescing several writes into a single message improves performance in two ways: (a) the extra communication induced by false sharing is reduced, and (b) fewer messages are communicated due to the coalescing.

Lazy release consistency (LRC) [KCZ92, DKCZ93] is a more aggressive implementation technique for page-based systems that further exploits optimizations enabled by relaxed models such as PL1.[61] Instead of eagerly propagating changes to all copies at each release synchronization as in Munin, LRC delays communication until a processor with a copy attempts to acquire a synchronization variable (e.g., through a lock). This corresponds to an aggressive implementation of the multiprocessor dependence chains in a model such as PL1 where the completion of a write at the head of a chain (with respect to the last processor in the chain) is delayed until just before the conflicting operation at the tail of the chain. The more advanced optimizations in LRC lead to a more complex and higher overhead protocol than Munin. Nevertheless, simulation results have shown that reduction in the number and size of messages can lead to an overall performance gain [KCZ92, DKCZ93].

The main difficulty with software page-based systems is the mismatch between the fixed size coherence granularity and the inherent grain of sharing and communication in an application. The ability to maintain coherence at a variable grain size has the potential of alleviating many of the problems (e.g., false sharing) that arise from such a mismatch. Midway [BZ91, BZS93] is an example of a software system that allows the coherence grain to be defined at the level of arbitrary programmer-defined regions. The Midway protocol is based on an extension of release consistency called *entry consistency* (EC). Entry consistency extends the release consistency model by requiring the programmer to explicitly associate shared data with synchronization variables. Similar to LRC, EC allows communication to be delayed until a processor attempts to acquire a synchronization. Instead of maintaining duplicate copies and computing diffs as in Munin and LRC, Midway

[61] The LRC implementation is more relaxed than a release consistent system because it enforces a multiprocessor dependence chain only if all the constituent conflict orders are among pairs of competing operations (e.g., as in the PL1 model).

associates a version number with finer grain sub-regions within each region. The compiler is required to generate extra instructions for each shared store to modify the appropriate version number. By comparing the version numbers of the two copies, the system can identify the sub-regions that must be communicated. EC provides an advantage over LRC since it can exploit the information associated with each synchronization to limit the communication to *only* the data regions protected by that synchronization. This is a classic example of an optimization that can be beneficial in a software design, and yet is too complex and probably not worthwhile to support in a hardware design.

One disadvantage of Midway compared to Munin and LRC is that the programmer is required to provide additional information to associate data with synchronizations. The original Midway proposal [BZ91] required all shared data to be explicitly associated with synchronizations. Since annotating every use of every shared data can become burdensome for programmers, a follow-on proposal [BZS93] prescribes supporting default conservative information whereby entry consistency intuitively degenerates either to release consistency or to processor consistency for certain operations depending on the amount of information provided by the programmer. This approach allows the programmer to selectively provide usage information for only the subset of data regions that should be kept coherent through entry consistency. The remaining data regions are kept coherent using a page-based approach with updates that adheres to either the processor or release consistency semantics.

Software systems such as Munin, LRC, and Midway provide their own stylized synchronization primitives such as locks and barriers. These synchronization primitives are usually implemented using explicit message passing (i.e., primitives are not implemented on top of a shared memory abstraction). Furthermore, programs are required to only use the provided primitives for synchronization and sufficient synchronization must be used to avoid competing data operations. This leads to a restrictive programming model since programmers cannot define their own synchronization primitives. However, the restrictive nature of the programs greatly simplifies the implementation of these systems. Unlike general models such as PL1, the only multiprocessor dependence chain that involves data operations is a category two chain that starts and ends with a pair of conflicting data operations, with all its constituent conflict orders being "write-read" pairs of synchronization primitives (e.g., an unlock-lock pair). Similarly, a data location cannot be accessed by a synchronization primitive, so data locations are always accessed by non-competing operations. Furthermore, these systems typically exploit the fact that neither the coherence requirement nor the termination condition need to be supported for data write operations (similar to non-competing writes in PL1, for example). Finally, they do not support atomic read-modify-write operations on data locations.

5.7 Interaction with Thread Placement and Migration

So far, we have implicitly assumed that every thread or process executes on its own dedicated physical processor. However, an actual system may map multiple threads or processes onto a single physical processor. Furthermore, threads or processes may be dynamically migrated to different physical processors during their execution. Thread placement and migration may interact with the memory behavior of a program if they are not carefully implemented. In this section, we describe various techniques for transparently supporting this type of resource scheduling.

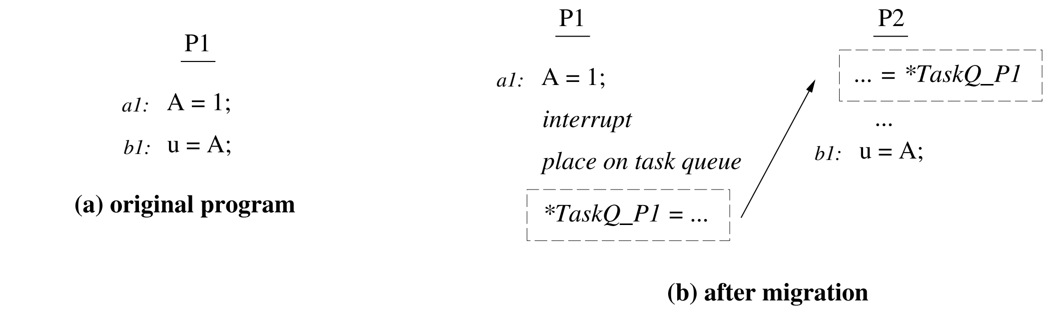


Figure 5.30: Example of thread migration.

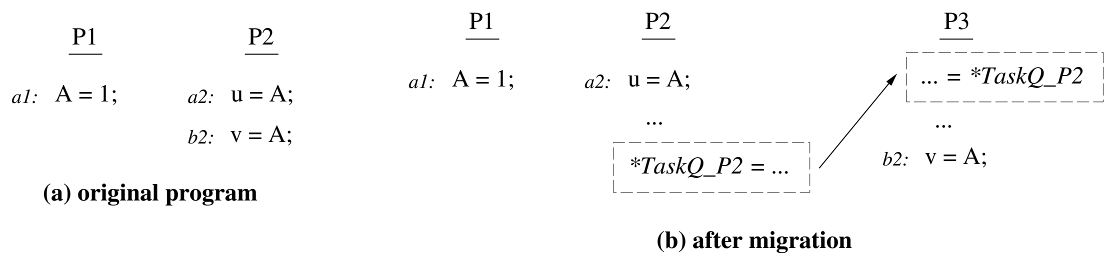


Figure 5.31: Example of a category 1 chain transformed to a category 3 chain.

5.7.1 Thread Migration

Figure 5.30 shows a simple program before and after thread migration. As shown in Figure 5.30(b), the thread executing on P1 is preempted after issuing its first operation, is placed on the task queue and removed from the queue by P2, and finally resumes its execution on P2. The overhead of achieving thread migration varies depending on how light-weight the threads are. Even in its simplest form, however, thread migration involves saving a minimum amount of context (such as registers on the source processor) and restoring this context on the target processor. There are numerous mechanisms and policies for thread migration. Regardless of how migration is achieved however, some form of communication is required between the source and target processors. For simplicity, our examples conceptually depict this communication through the write and read operations to the task queue.

Consider how thread migration can interact with memory behavior. Assume the program in Figure 5.30(a) is written for the sequential consistency model, thus requiring the read of A to return the value 1 in all executions. For thread migration to be transparent, the above condition must also be upheld after migration. Such transparency is present if the implementation maintains the multiprocessor dependence chain from the write of A on P1 to the read of A on P2 with the intermediate conflict order between the write and read of the task queue. Since this chain is maintained by any SC implementation, the above migration is indeed transparent.

In the example above, migration transforms a simple program order relation between two conflicting operations into a category 2 multiprocessor dependence chain. Similarly, a category 1 chain may be transformed into a category 3 chain, for example as shown in Figure 5.31. Finally, migration transforms category 2 and category 3 chains into longer chains of the same category. And in all cases, migration introduces extra operations to new addresses (e.g., operations to task queue) within the chain.

As discussed above, transparent thread migration requires any execution outcomes that are disallowed before migration to remain disallowed after migration. The above condition is trivially satisfied for implementations of the SC, IBM-370, and TSO models because these models enforce the resulting multiprocessor chains that arise due to migration. However, implementations of other models typically require extra precautions to satisfy this condition. For example, consider an implementation of the PSO model with the example in Figure 5.30. Since the outcome (u=0) is disallowed before migration, it must remain disallowed after. However, the program as shown in Figure 5.30(b) may yield the (u=0) outcome under PSO since the write to the task queue is allowed to complete before the write of A. To remedy this, the write to the task queue on P1 must be proceeded with a write fence (the store barrier for PSO) to ensure the chain from the write of A to the read of A is upheld. This can be easily achieved by appropriately modifying the preemption code along with the code that performs the thread migration.

Other models require similar modifications to the preemption and thread migration code to ensure correctness. For example, WO requires both the write and the read operation to the task queue to be labeled as synchronizations in the example in Figure 5.30. The RCsc and RCpc models require the two operation to be labeled as release and acquire respectively. Alpha, RMO, and PowerPC all require the write to be preceded by and the read to be followed by a fence.[62] Finally, PL1 requires competing labels, and PL2 and PL3 require sync labels for both operations.[63]

The requirements described above may be insufficient for achieving transparency in aggressive implementations of the PC, RCsc, RCpc, and PL3 models.[64] The problem arises due to certain category 1 chains that are enforced before migration, but are transformed to unenforced category 3 chains after migration.[65] Figure 5.31 shows a canonical example where a category 1 chain is transformed into a category 3 chain as a result of migration. Consider the PC model as an example. Given the original program in Figure 5.31(a), the outcome (u,v)=(1,0) is disallowed. However, PC allows the (u,v)=(1,0) outcome for the transformed program after thread migration since the resulting category 3 chain is not enforced. The RCsc, RCpc, and PL3 models can lead to analogous problems.[66] Whether an implementation exhibits the above problem depends on how aggressive it is; e.g., implementations that conservatively support multiple-copy atomicity for all writes (or in case of PL3, for all competing writes) do not exhibit the anomalous behavior described above.

Even though the example in Figure 5.31 may seem contrived, this type of interaction can easily arise in practice. Figure 5.32 shows an example program where P1 modifies the fields of a record and then sets Rec_Ptr to point to the record, and P2 waits for the pointer to become valid before reading the fields of the record. The competing operations are shown in bold (labeled as release and acquire for RCpc, and as loop for PL3). The above program provides sequentially consistent results if executed on a PC, RCpc, or PL3

[62] RMO requires a membar(RW,WW) fence before the write and a membar(RR,RW) fence after the read.

[63] The operations can be conservatively labeled as non-loop syncs in PL3; the more aggressive loop label can be used if the operations satisfy the requirements for this label.

[64] The addition of the third multiprocessor dependence chain in the sufficient conditions for PL1 and PL2 (Figures 4.14 and 4.15, respectively) presented in Chapter 4 ensures that the above two models do not exhibit the problem described here.

[65] Even though PowerPC does not support any category 3 chains, it also does not support any category 1 chains so the above problem does not arise. A similar problem can arise for ports of PL3 programs, as specified in Section 4.4 of the previous chapter, to a sufficiently aggressive implementation of PC or RCpc; one way to remedy this problem is to use a more conservative port, such as the same port used for PL2 programs.

[66] Examples that lead to the above problem in aggressive implementations of RCsc are a bit more subtle. Consider the program in Figure 5.31(a) with all operations labeled as non-competing, and insert a release to location B and an acquire to location C between the two reads of A on P2. In this example, the category 1 chain from the write of A to the second read of A is upheld under the RCsc model, but the category 3 chain for the transformed program (e.g., assuming the migration point is right before the second read of A) is not guaranteed to be enforced even if the operations to the task queue are appropriately labeled according to the rules in the previous paragraph.

Initially Rec_Ptr = nil

	P1		P2
a1:	tmp_ptr−>field1 = 1;	*a2:*	while (**Rec_Ptr** == nil);
b1:	tmp_ptr−>field2 = 2;	→ *b2:*	u = Rec_Ptr−>field1;
c1:	**Rec_Ptr** = tmp_ptr;	→ *c2:*	v = Rec_Ptr−>field2;

Figure 5.32: A more realistic example for thread migration.

implementation. However, if the execution thread on P2 is preempted (at either of the two points shown by the arrows) and migrated to a different processor, the program may erroneously dereference a null pointer (in statement (b2) or (c2)) since the new value for the pointer may not have yet been propagated to the third processor.

A general way to remedy the above problem for aggressive implementations of PC, RCsc, RCpc, and PL3 is to ensure that any writes that have completed with respect to the processor from which the thread is migrated also complete with respect to the target processor where the thread is migrated. Consider the example in Figure 5.33(a) executing on an aggressive implementation of PC. The outcomes (u,v,w)=(1,2,0) and (1,2,1) are disallowed before migration. However, either outcome may occur with the simple migration shown in Figure 5.33(a). Figure 5.33(b) shows one way to extend the migration code to guarantee correct behavior. As shown, the processor where the thread is migrated to (i.e., P4) sends an interrupt to all other processors (except the processor where the thread is migrated from) and forces a synchronization with these processors to ensure all write issued by those processors are complete with respect to it. In this way, the implementation guarantees that the read of A on P4 completes after the writes of A on P1 and P2.[67] For models such as RCsc, RCpc, and PL3, appropriate labels must be used for the additional synchronizations to ensure the relevant chains are indeed enforced. There are several ways to reduce the overhead of the above technique. For example, the runtime or operating system may be able to limit the set of processors that need to be interrupted based on knowledge of other processors that have access to the pages used by the migrated thread. Similarly, the interrupt and synchronization phases may be optimized by using tree algorithms. Nevertheless, interrupting and communicating with a set of other processors increases the overhead of thread migration, and can be especially undesirable in systems with light weight threads and frequent thread migration. This creates a trade-off for aggressive implementations of the four models discussed above.

5.7.2 Thread Placement

In specifying memory models, we inherently assumed an ideal system where each thread executes on its own dedicated processor. In practice, however, each processor may execute multiple threads by context switching from one thread to another. Nevertheless, the memory behavior for the real system must resemble that of the ideal system; i.e., outcomes that are disallowed in the ideal system must remain disallowed by the real system. This section describes several techniques for ensuring the above property.

[67] The above mechanism is quite similar to the way TLB consistency is managed by some operating systems when a page mapping is changed; the node that attempts to change the mapping communicates with other processors to ensure that all previous operations issued to the given physical page complete before the mapping is altered.

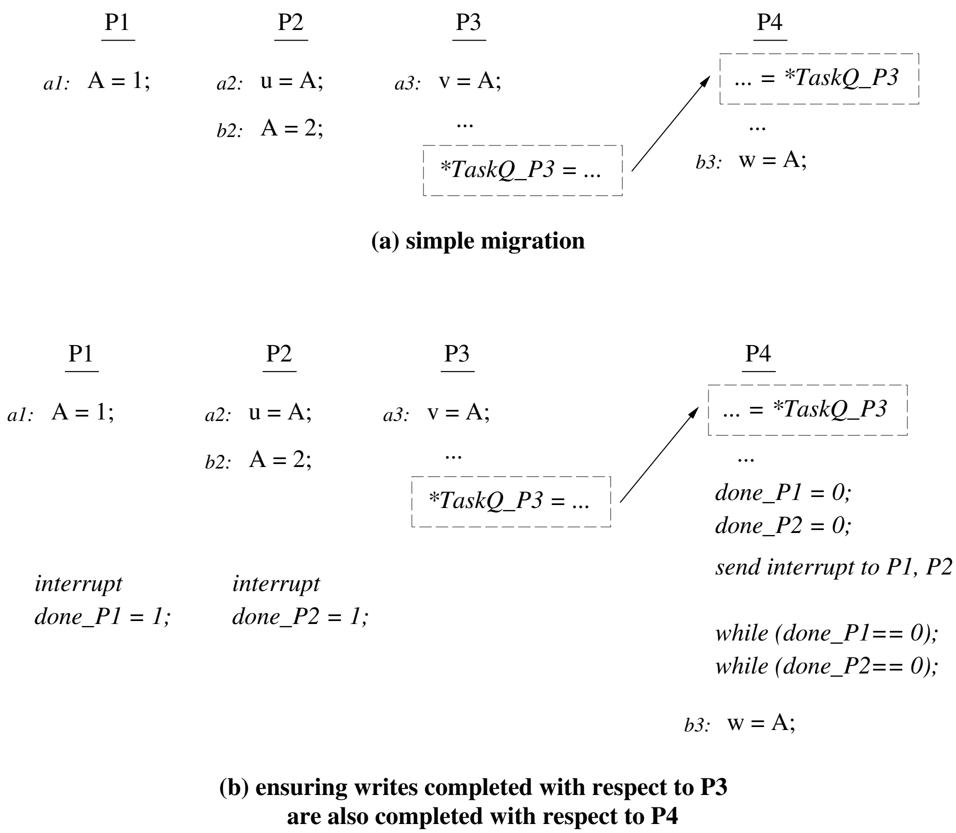


Figure 5.33: Different ways of implementing thread migration.

P1	P2	P3
a1: A = 1;	*a2:* u = A;	*a3:* v = B;
	b2: B = 1;	*b3:* w = A;

(a) original program

P1'	P2'
a1: A = 1;	*a3:* v = B;
a2: u = A;	*b3:* w = A;
b2: B = 1;	

(b) scheduled on two processors

Figure 5.34: Scheduling multiple threads on the same processor.

The main issue that arises with respect to thread placement is the potentially different behavior of conflicting operations when they are issued by different processors as compared to when they are issued by the same processor. For example, consider a write on P1 and a conflicting read on P2. If the operations are on different processors, the read is not allowed to return the value of the write until the write completes with respect to P2. Implementations that enforce multiple-copy atomicity (assuming the first technique described in Section 5.3.5) may further restrict the interaction: the read is not allowed to return the value of the write until the write commits or completes with respect to all processors. Contrast the above restrictions with the case when the conflicting operations are issued by the same processor. For example, implementations that exploit the read forwarding optimization allow the read to return the value of the write before the write is even serialized with respect to other writes. Consequently, the fact that the read has completed (i.e., it returned the value of the write) no longer implies that the corresponding write is committed or completed with respect to any processor. The above can lead to different semantics when multiple threads are mapped onto the same processor.[68]

Figure 5.34 shows a simple example of the write-read interaction described above. The original program has three independent threads. Assuming the SC model, the outcome (u,v,w)=(1,1,0) is disallowed. Figure 5.34(b) shows the effect of mapping the first two threads onto a single physical processor. Consider the interaction between the write of A and the read of A in statements (a1) and (a2), respectively. Assume an aggressive implementation of SC. When the two operations are on different processors, the read is not allowed to return the value of the write until the write is committed or completed with respect to all other processors. Therefore, by delaying the write of B for the read to complete, the implementation can ensure that the write of A has been committed with respect to all processors before the write of B commits or completes with respect to any processor. On the other hand, when the operations are on the same processor, a sufficiently aggressive implementation of SC may allow the read to return the value of the write before the write is even serialized with respect to other writes. Therefore, delaying the write of B for the read of A fails to impose an order between the write of B and the write of A. Fortunately, as long as the SC implementation treats the operations on P1' as though they belong to the same thread, the above issue does not lead to a problem; the write of B is still delayed for the write of A to commit or complete with respect to all processors and therefore, the outcome (u,v,w)=(1,1,0) remains disallowed. However, problems may arise if the SC implementation attempts to exploit the fact that the write of A and B actually belong to two different threads and does not enforce the "program order" between the two writes.

[68] The anomalous behaviors described in this section do not occur in the degenerate case where all threads are mapped to the same processor.

```
      P1              P2              P3
a1:  A = 1;      a2:  A = 2;      a3:  u = B;
                 b2:  B = 1;      b3:  v = A;
```

(a) original program

```
      P1'               P2'
 a1:  A = 1;       a3:  u = B;
 a2:  A = 2;       b3:  v = A;
 b2:  B = 1;
```

(b) scheduled on two processors

Figure 5.35: An example of thread placement with a write-write interaction.

A similar issue arises with write operations in implementations that support eager exclusive replies (see Section 5.3.5). Consider two conflicting write operations on different processors (anomalies may also arise for non-conflicting writes if they lie on the same cache line). As mentioned in Section 5.3.5, to facilitate detecting the completion of writes, most implementations delay the service of the second write until the first write is completed or committed with respect to all processors. However, if the two writes are issued by the same processor, the second write may be serviced immediately after the eager exclusive reply returns for the first write, even though the first write may not have completed yet. Figure 5.35 shows an example that illustrates the write-write interaction. Assume a model such as WO with the write and read to B labeled as a synchronization operations. The outcomes (u,v)=(1,0) and (1,1) are disallowed by the original program, but may not be disallowed in Figure 5.35(b) given an aggressive implementation with eager replies where the write of B does not get delayed for the first write of A (labeled (a1)).

Anomalies arising from the above write-write interaction can be alleviated for any memory model by conservatively treating all operations on the same processor (e.g., P1' in the above example) as belonging to the same thread. Unfortunately, the above is not a sufficient solution for anomalies arising from the write-read interaction when all combinations of models and implementations are considered. Consider the example in Figure 5.36, along with an aggressive implementation of a model such as TSO or PC. The outcome (u,v,w,x)=(1,2,0,0) is disallowed in the original program mapped onto four processors. Figure 5.36(b) shows the scenario where two threads are assigned to each processor. In this case, even if the operations on P2' are conservatively treated as if they belong to the same thread, an aggressive implementation of TSO or PC can lead to the outcome (u,v,w,x)=(1,2,0,0) (this is the same example as in Figure 2.14(b) discussed in Chapter 2). The above behavior is somewhat anomalous since it is typically the case that the same program running on fewer processors leads to fewer possible outcomes. However, the read forwarding optimization changes this (for some of the models) by actually allowing a program that runs on fewer processors to possibly lead to more outcomes than if it ran on more processors.

Similar problems can arise for implementations of other models whose semantics are stricter in the absence of the read forwarding optimization. These include TSO, PC, PSO, RCsc, RCpc, RMO, and the sufficient specifications for PL1, PL2, and PL3. The example in Figure 5.36 can be easily adapted to illustrate this issue for most of the above models. For the PL1 specification, for example, simply consider the operations to location C to be competing and the remaining operations to be non-competing.[69] Examples for RCsc are a little bit more subtle; Figure 5.37 shows one such example where the outcome (u,v,w,x)=(2,0,1,2) is

[69]This labeling does not make the program properly-labeled. However, the sufficient conditions for PL1 (or PL2 or PL3) place restrictions on non-properly-labeled programs as well. Our conjecture is that the anomalous behavior will not be noticeable for the properly-labeled programs.

P1	P2	P3	P4
a1: A = 1;	*a2:* u = C;	*a3:* B = 1;	*a4:* w = C;
b1: C = 1;	*b2:* v = B;	*b3:* C = 2;	*b4:* x = A;

(a) original program

P1'	P2'
a1: A = 1; *b1:* C = 1;	*a3:* B = 1; *b3:* C = 2;
a2: u = C; *b2:* v = B;	*a4:* w = C; *b4:* x = A;

(b) scheduled on two processors

Figure 5.36: Another example of thread placement with a write-read interaction.

disallowed by the original program, but may potentially be allowed when two of the threads are merged onto a single processor.

One conservative solution for remedying the write-read interaction described above is to simply disallow optimizations such as read forwarding. However, this solution is undesirable since it can reduce single thread performance. Another conservative solution is to wait for all writes from a processor to complete (or commit with respect to other processors) before switching to a different context. This functionality can be included as part of the context switch code, and achieving the delay may require using explicit fence instructions in some designs. In this way, a read or write operation from the next thread is inherently delayed until the conflicting writes from the previous thread are complete, and therefore the interaction among conflicting operations in the two threads is similar to when the threads are on different processors.

The solutions described above are satisfactory for systems with infrequent context switching among processes or threads. However, these solutions are not appropriate for systems that exploit context switching at a finer grain, for example, to achieve overlap among longer latency operations from different threads. Treating operations from multiple threads as though they belong to the same thread is undesirable since it imposes unnecessary program order among operations from independent threads, thus reducing the opportunity for overlap especially with stricter models such as SC. Similarly, waiting for writes in one thread to complete before switching to the next thread is undesirable because it eliminates the opportunity to overlap the latency of the write with operations in the next thread.

A more aggressive solution is possible if the implementation dynamically detects a read or write operation from one thread that is to the same line as an outstanding write operation from a different thread and delays the service of this operation relative to the outstanding write in the same way that it would be delayed if it had been issued by another processor. The above requires adding some form of thread identification to the record of the outstanding write operations from the processor (e.g., by extending the entries in the pending write buffer discussed in Section 5.2.3). In this way, optimizations such as read forwarding can be allowed

P1		P2		P3		P4	
a1:	B = 1; *(rel)*	*a2:*	A = 2;	*a3:*	u = A; *(acq)*	*a4:*	w = A;
b1:	A = 1; *(rel)*			*b3:*	v = B;	*b4:*	x = A;

(a) original program

P1'		P2'		P3'	
a1:	B = 1; *(rel)*	*a2:*	A = 2;	*a4:*	w = A;
b1:	A = 1; *(rel)*	*a3:*	u = A; *(acq)*	*b4:*	x = A;
		b3:	v = B;		

(b) scheduled on three processors

Figure 5.37: Example of thread placement for the RCsc model.

for read operations from the same thread while they are disallowed for read operations from a different thread (read forwarding from a different thread is disallowed if the write is not yet complete with respect to its processor). Similarly, for models that require multiple-copy atomicity for writes, a conflicting read from a different thread is delayed until the outstanding write actually commits (or completes) with respect to all processors.[70] Finally, to deal with the write-write interaction in implementations with eager exclusive replies, a write operation from a different thread that is within the same cache line is delayed until the outstanding write is committed with respect to all processors.

The above solution effectively disallows optimizations, such as read forwarding, that are used within a single thread from being used across threads. This technique clearly incurs more hardware support and complexity. However, it exploits the fact that no program order must be enforced among operations from different threads, and does not unnecessarily delay the next thread for completion of writes from the previous thread. In fact, delays only occur when there is an operation from one thread to the same address (or cache line in case of a write) as an outstanding write from the previous thread. Such delays are typically infrequent since the chances for a write immediately followed by a read or write to the same line is much lower across threads as compared to within a single thread. Therefore, this solution can indeed efficiently support fine grain context switching.

While the above discussion has focused on executing multiple threads on a single processor, similar issues may arise in designs that tightly couple processors to share buffers or caches within a cache hierarchy. For example, the same write-read or write-write interactions may be present if two processors share a cache and the write from one processor can become visible to the other processor before the write is serialized or committed with respect to all processors. The aggressive solution that dynamically disallows the service of conflicting operations from one thread while there is an outstanding write from another can be easily adapted for achieving correct implementations in these types of designs.[71]

[70] As discussed in Section 5.3.5, this requirement is unnecessary for the PC, RCpc, and PowerPC models. For the PL1 and PL2 models, it is sufficient to delay the read only if the outstanding write is competing. For the PL3 model, it is sufficient to delay the read only if the write is non-loop or non-sync.

[71] The remote access cache (RAC) in DASH uses a variation of this solution by actually allowing an operation from another processor

5.8 Interaction with Other Latency Hiding Techniques

This section describes the interaction of memory consistency models with two other latency hiding techniques: prefetching and use of multiple context processors.

5.8.1 Prefetching

Prefetching techniques can be classified based on the following characteristics:

- binding or non-binding,
- hardware or software controlled,
- shared or exclusive copy prefetch, and
- producer-initiated or consumer-initiated.

Section 5.4.2 introduced some of the above distinctions.

A *binding* prefetch binds the value of a later reference at the time the prefetch completes. In contrast, a *non-binding* prefetch simply brings data closer to the processor and the value is kept coherent until the processor actually accesses the data. Binding prefetches can affect the memory behavior of a program and must be treated identically to read operations in terms of their ordering with respect to other memory operations. On the other hand, non-binding prefetches do not affect the correctness of a program and can be used simply as a performance boosting technique; for example, the data within a critical section can be safely prefetched before the critical section is actually entered. The following discussion focuses on non-binding prefetches due to their greater flexibility.

The second distinction is based on whether prefetch operations are generated automatically by *hardware* or whether the compiler or programmer must invoke the appropriate prefetch operations through *software*. The next feature distinguishes whether the copy that is prefetched is a shared or an exclusive copy of the data, corresponding to a *read* versus a *read-exclusive* prefetch, respectively. Finally, *consumer-initiated* prefetch is the more common form of prefetching where the processor that issues the prefetch is the ultimate consumer of the data. The alternative is *producer-initiated* prefetch where the "producer" processor pushes a copy of the data closer to the consumer processor. The deliver operation in DASH [LLG$^+$92] and the post-store operation in KSR [Res92] are examples of producer-initiated prefetches. The following discussion primarily focuses on consumer-initiated prefetching.

To eliminate unnecessary prefetches, consumer-initiated read and read-exclusive prefetch operations should ideally check for the data within the processor's cache hierarchy before they are issued to the memory system.[72] Furthermore, it is desirable to keep track of outstanding prefetch operations to allow the merging of subsequent prefetches or ordinary operations to the same cache line. A simple way to achieve the above is to treat prefetches as ordinary read and write operations within a lockup-free cache hierarchy. Of course, keeping track of outstanding prefetch operations is actually simpler than keeping track of outstanding read or write operations since there is no destination register associated with a read prefetch and no write data

to be serviced (even though invalidations may be pending to that line) and noting this event. The appropriate delay is imposed on the next fence operation from the processor.

[72] A read-exclusive prefetch may be converted to an exclusive prefetch if the cache already has a clean copy.

associated with a read-exclusive prefetch. Furthermore, since prefetches can simply be dropped, a processor does not necessarily need to stall when there are insufficient resources for keeping track of a prefetch.[73]

Consider the required program ordering constraints for consumer-initiated non-binding prefetches. A read prefetch need not be delayed for any previous operations, and no future operations need to be delayed for the prefetch to complete. Similarly, there are no program ordering constraints for read-exclusive prefetches if the memory system services the prefetches using delayed exclusive replies (see Section 5.3.5). In fact, some designs may use a separate buffer for prefetch operations which will inherently lead to the reordering of prefetches with respect to other operations. However, program ordering constraints become important if the memory system uses eager exclusive replies to service read-exclusive prefetches. This is because an eager exclusive reply to the prefetch can allow a subsequent write W to the same line to be serviced by the cache even though there may still be outstanding invalidation due to the prefetch. However, a later operation that must be delayed for W needs to wait for these invalidations to be acknowledged. One possible solution is to increment the count of outstanding writes on a read-exclusive prefetch (that may be serviced by an eager exclusive reply) sometime before the prefetch reply returns. Therefore, subsequent operations that wait for W will inherently wait for previous read-exclusive prefetches as well.[74]

Within the memory system, consumer-initiated prefetches can be treated in virtually the same way as ordinary read or read-exclusive operations. This ensures that prefetched data will be kept coherent at all times. For example, ordering constraints such as those described in Section 5.3.2 also apply to prefetch operations. Similarly, replies to prefetch operations must be treated in the same way as ordinary replies in systems that exploit early invalidation or update acknowledgements (see Section 5.4.1). For this reason, most systems do not distinguish between prefetch and ordinary operations once the prefetch leaves the processor environment.

Finally, we briefly consider producer-initiated prefetches for pushing either a shared or exclusive copy of a line to another processor. The sole purpose of producer-initiated prefetches is movement of copies and not eliminating stale copies. The processor that issues the producer-initiated prefetch need not wait for its completion; therefore, no acknowledgements are required. Within the memory system, a producer-initiated prefetch operation that carries a shared copy of the line in many ways behaves similarly to an update request. However, unlike an update request, the prefetch can be safely dropped unless it is carrying an exclusive copy.

5.8.2 Multiple Contexts

Multiple context processors offer an alternative mechanism for hiding latency by assigning multiple threads to each processor and context switching among the threads at a fine granularity. This allows a long latency operation on one thread to be overlapped with computation and communication in the other threads.

A key decision in the design of a multiple context processor is the granularity at which context switches occur. A *blocked* multiple context processor switches to a different context only if the processor encounters a long latency operation such as a cache miss. An *inter-cycle interleaved* multiple context processor switches to a different active context on every cycle; a context remains inactive while it is waiting for a long latency

[73] A consequence of merging prefetches with subsequent operations is that the memory system may no longer drop prefetch operations silently past the merge point. An explicit negative acknowledgement reply may be used to signal a prefetch that is not serviced successfully.

[74] It is sufficient to wait only for the read-exclusive prefetches whose eager reply is back; prefetches that have not yet been issued, or prefetches with outstanding replies, do not lead to the early service of a subsequent write.

operation. Finally, *intra-cycle interleaving* allows issuing instructions from different contexts within the same cycle. Providing fast context switch times requires support in hardware. A major component of this support involves replicating state structures such as register files for the resident set of contexts in order to avoid the overhead of saving and restoring such state. The blocked scheme requires less aggressive support due to the lower frequency of switches and the fact that switch overheads of a few cycles are tolerable. On the other hand, interleaved schemes effectively require a zero overhead switch. In return, interleaved schemes are capable of hiding smaller latencies such as the latency associated with a floating point functional unit or a first level cache miss.

From a memory model perspective, the implementation should ideally exploit the independence of memory operations on different threads especially with respect to program ordering constraints. Consider supporting four contexts on each processor. Even with a strict model such as sequential consistency, each context must be allowed to have a single outstanding memory operation leading to a total of four outstanding requests from each processor. Therefore, while a single context processor implementation of sequential consistency may allow a single outstanding operation, the multiple context implementation must provide sufficient support structures to allow operations from different threads to be serviced simultaneously.

Exploiting the independence of memory operations across threads is not too difficult for a model such as sequential consistency since the implementation can simply keep track of a single outstanding operation per thread. Things become more challenging for more relaxed memory models, however. For example, consider models such as RCsc or PL1 that allow each thread to have multiple read and multiple write operations outstanding. A brute force technique for independently dealing with operations from different threads would require replicating the appropriate read and write buffers (and potentially counts of outstanding operations), thus providing separate structures to keep track of the operations from each active context. However, simpler implementations that do not fully exploit the independence of operations across threads can often provide most of the performance gains. For example, a single context design that supports the PL1 model with blocking reads can be extended to allow one outstanding read per context, but may still use a single write buffer for all write operations. This causes a competing write operation to be conservatively delayed for not only its own context's previous writes but also for the previous writes from the other contexts. However, since write latency is not visible to the thread under a model such as PL1 (i.e., writes are simply buffered and the thread does not need to stall unless the write buffer is full), the extra delay in servicing writes due to a unified write buffer does not have a direct impact on performance. Similar trade-offs can be made for designs with non-blocking reads where each context may have multiple outstanding reads. Assuming the PL1 model again, the implementation may be extended to allow each context to have a single outstanding competing read. However, non-competing reads from different contexts may still be conservatively merged into a single read buffer, and writes can be handled as described above.

The main correctness issue with respect to the interaction of memory models and supporting multiple contexts was discussed Section 5.7.2. As discussed there, one way to eliminate the anomalous behavior that arises from this interaction is to delay operations from the next context if a previous context has an outstanding write to the same memory line. This functionality may require extra hardware mechanisms in a multiple context design.

Finally, from an implementation point of view, the lockup-free cache design for a multiple context processor is complicated by the fact that cache misses typically lead to two phase transactions: (a) a thread originally requests a line and becomes inactive, (b) the reply to the line puts the thread back on the active list, (c) the request reissues when the thread is rescheduled and is satisfied if the line is still in the cache. The problem that arises is that other activity (e.g., instruction or data fetches or operations by other processors) may invalidate or replace the line between the time the reply comes back and the time the thread gets to reissue its request. The above behavior can potentially lead to livelock where the system continuously fails to fully service requests. Kubiatowicz et al. [KCA92] and Laudon [Lau94] suggest several possible solutions for alleviating such livelock problems.

5.8.3 Synergy Among the Techniques

Relaxed memory models and prefetching are alternative ways for exploiting fine grain overlap of computation and memory operations within a single thread. Multiple contexts, on the other hand, provides overlap of computation and memory operations by exploiting the parallelism across different threads. Since the three techniques are alternative ways for hiding latency in a hardware architecture, it may seem that the use of one technique eliminates the need for the others. However, it turns out that there is synergy in exploiting relaxed memory models with either the prefetching or the multiple context techniques. We illustrate this synergy by providing performance results for combinations of these techniques in the next chapter.

There is also synergy from an implementation point of view. The synergy arises from the fact that many of the architectural mechanism used to exploit relaxed memory models are also needed for fully exploiting the prefetching or multiple context techniques. This should not be surprising since all three techniques hide latency by overlapping and pipelining multiple memory operations, thus they all gain from more aggressive cache and memory designs. Lockup-free caches are a good example of a universal requirement across the three techniques. Even though prefetching allows for a simpler lockup-free cache design, such a design can be easily extended to support the extra requirements for relaxed models (e.g., forwarding read data to a register on a read reply and merging the write data on a write reply). Alternatively, a lockup-free cache design that supports multiple contexts already has sufficient mechanisms to also support relaxed memory models.

5.9 Supporting Other Types of Events

The implementation discussion has so far focused solely on the ordering of data read and write operations to a single granularity (e.g., a word). As discussed in Section 4.5 of the previous chapter, a real implementation has to deal with ordering issues among a larger set of events. The following are examples of other types of events that must be considered:

- instruction fetch,
- data operations at multiple granularities (e.g, byte, word),
- operations to I/O devices, and
- exception events (e.g., for arithmetic operations or related to virtual memory).

Most designs violate the notion of serial semantics for the above events in order to achieve higher performance. Furthermore, many of the ordering issues that arise with respect to these events are not unique to multiprocessors and must be dealt with in uniprocessor systems as well.

Given the above set of events, it is often desirable to enforce ordering constraints among not only events of the same type but also events of different types. For example, memory operations may be used to synchronize access to I/O devices. Therefore, an implementation requires mechanisms for detecting the completion of each type of event (e.g., an I/O write) and mechanisms for enforcing orders among different types of events (e.g., between an I/O write and a data write). Furthermore, events such as I/O operations often exhibit non-memory-like side effects, making some of reordering optimizations that are safe for ordinary memory operations unsafe for such operations.

Appendix Q describes some of the practical implementation issues for supporting ordering among the larger set of events, primarily focusing on solutions adopted by various commercial architectures such as the Alpha [Sit92], PowerPC [MSSW94], and Sparc V9 [WG94].

5.10 Implications for Compilers

The previous sections in this chapter have focused on the impact of memory model specifications on the hardware implementation of a system. The memory model also impacts the implementation of the compiler since many important compiler optimizations involve reordering or eliminating memory operations (e.g., register allocation, code motion, loop interchange). This section describes the implications of memory model specifications on optimizations that can be safely exploited by the compilers explicitly parallel programs (with explicit synchronization; see Section 3.5.1). Many of the ideas in this section are extensions of our earlier joint work with Adve [GAG+93].

This section begins by motivating issues related to compiler optimizations and describing some of the problems with current compiler implementations. Section 5.10.2 describes the assumptions about the level of information conveyed by a source program and the type of architecture that the compiler is targeting. Section 5.10.3 provides a framework for reasoning about various compiler optimizations and their effect on memory behavior. Finally, Section 5.10.4 describes the safe compiler optimizations for the various memory models.

5.10.1 Problems with Current Compilers

Figures 5.38 and 5.39 show a couple of examples to illustrate the effect of compiler optimizations on memory behavior. The example in Figure 5.38 shows two nested for-loops before and after loop interchange; this optimization may for example be used to increase temporal locality for the B[j] reference. The loop interchange is safe for a uniprocessor since the there are no data dependences across the loop iterations. From a memory model perspective, however, the effect of the loop interchange is a major reordering of the memory operations to arrays A and B. Such reordering can potentially translate into incorrect behaviors in a multiprocessor especially if the parallel program is written assuming a strict model such as sequential consistency (e.g., consider other processors simultaneously accessing the arrays without any synchronizations). The above example is indicative of a large group of compiler transformations that lead to a reordering of memory operations.

Figure 5.39 shows the effect of another common optimization. Figure 5.39(a) shows the original program, written for the SC model. P1 writes to A, sets a flag, then reads A, and P2 waits for the flag to be set before reading A. Under SC, the while loop on P2 always terminates and the only possible outcome is (u=1).

```
a1: for (i=0; i<m; i++)
b1:     for (j=0; j<n; j++)
c1:         A[i][j] = B[j];
```

(a) before

```
a1: for (j=0; j<n; j++)
b1:     for (i=0; i<m; i++)
c1:         A[i][j] = B[j];
```

(b) after

Figure 5.38: Effect of loop interchange.

	P1		P2
a1:	A = 1;	*a2:*	while (Flag == 0);
b1:	Flag = 1;	*b2:*	u = A;
c1:	... = A;		

(a) before

	P1		P2
a1:	r1 = 1;	*a2:*	r2 = Flag;
b1:	Flag = 1;	*b2:*	while (r2 == 0);
c1:	... = r1;	*c2:*	u = A;
d1:	A = r1;		

(b) after

Figure 5.39: Effect of register allocation.

Figure 5.39(b) shows the effect of register allocating locations A and Flag. This optimization is perfectly legal if each processor's code is treated as a uniprocessor program. However, the transformation can violate the semantics of a model such as SC. Register allocating A on P1 makes it possible for P2 to read the value of 0 for A since the write to A on P1 no longer occurs before the write to Flag. This is an example of how register allocation can effectively lead to a reordering of operations. Similarly, register allocating Flag on P2 can lead to a potentially non-terminating while loop (i.e., if the read of Flag that sets r2 returns the value 0) since P2 essentially reads Flag only once and does not monitor its value continuously. Neither of the above behaviors is allowable for the original program under SC.

Problems similar to the above may routinely arise when compiling explicitly parallel programs. The reason is most compilers that are used to compile parallel programs are not "multiprocessor-ready"; the resulting optimizations that arise from treating the program as a uniprocessor code can violate the semantics of even the most relaxed multiprocessor memory models. A common technique for remedying this problem is to use the *volatile* keyword provided by languages such as C to identify shared variables. The functionality of the volatile keyword was originally devised to deal with I/O operations in the context of uniprocessors. Compilers typically avoid register allocating volatile variables and maintain program order among operations to volatile variables. Ensuring correctness for an explicitly parallel program requires all shared locations to be declared as volatile. For example, both A and Flag must be declared as volatile in Figure 5.39.

While the functionality supported by the volatile keyword is suitable for enforcing the semantics of a strict model such as sequential consistency, it lacks sufficient flexibility for efficiently supporting more relaxed models. The main difficulty arises from the fact that operations to volatile variables are not ordered with respect to operations to non-volatile variables. For example, for a model such as WO, declaring all synchronization operations as volatile is not sufficient for correctness since the compiler may still move or register allocate data operations across synchronization operations. The alternative is to declare all shared locations as volatile which fails to exploit any of the reorderings allowed by most relaxed models. The key mechanism that is missing is a way to disallow optimization of non-volatile operations across boundaries set by volatile operations. An empirical and inelegant solution is to insert code segments that are difficult for the compiler to analyze before and after the volatile operations, in effect forcing the compiler to avoid code motion and register allocation across such boundaries. For example, for compilers that do not perform aggressive interprocedural analysis (and do not in-line procedures), a dummy procedure call may be used to force the compiler to spill registers before the call and reload registers after the call. As compiler technology advances, however, the need for a more robust and elegant solution increases.

A number of other researchers have observed the correctness issues that arise when uniprocessor compiler technology is applied to explicitly parallel programs. Midkiff et al. [MPC89] briefly explore the idea of finding more parallelism in an already parallelized program and describe the difficulties that arise due to the strict ordering constraints imposed by SC. Callahan and Smith [CS90] mention the idea of providing the compiler with further information about operations (labels similar to those used by release consistency) in order to allow more flexibility for compiler optimizations.

5.10.2 Memory Model Assumptions for the Source Program and the Target Architecture

A given source program assumes an underlying memory model. At the same time, the target architecture or hardware may support a potentially different model. Therefore, the requirements on a compiler are twofold. First, compiler optimizations must be constrained to correctly support the semantics of the memory model assumed by the source program. Second, the compiler needs to correctly map the semantics of the source memory model onto the hardware's target model when the two models are different. Section 4.4 in the previous chapter described techniques for efficiently mapping the semantics of one model onto another. The discussion here focuses on the first requirement of determining safe compiler optimizations with respect to the source program model. However, as we will see in Section 5.10.4, the compiler may still need to be aware of the target architecture's memory model even for determining the safe optimizations.

To allow for safe optimizations, sufficient information in the form of labels or fences must be communicated to the compiler. Section 3.5 in Chapter 3 described techniques for communicating the associated operation labels for properly-labeled models. Similar techniques may be used to communicate appropriate labels for models such as WO, RCsc, and RCpc. For fence-based models such as Alpha, RMO, and PowerPC, the explicit fence instructions are assumed to be directly visible to the compiler.

5.10.3 Reasoning about Compiler Optimizations

There are two types of compiler optimizations from a memory model perspective: (a) optimizations that lead to a simple reordering of memory operations (e.g., code motion, loop transformations such as interchange and blocking [LRW91]), and (b) optimizations that eliminate or substitute memory operations and other instructions (e.g., register allocation, common sub-expression elimination).

Reasoning about the first type of compiler optimizations is relatively straightforward. Given an execution E of an optimized version of the program, the optimizations are considered safe if the instructions and the sub-operations of E (including the values read and written) form a valid execution of the unoptimized program with respect to the memory model assumed by the source program. Therefore, we can reason about reordering optimizations by the compiler in almost the same way as we reason about such optimizations by the hardware.

The above reasoning precludes the use of optimizations of the second type, such as register allocation and common sub-expression elimination, that eliminate or substitute memory sub-operations and/or instructions. One way to deal with such optimizations is to start with the execution of the optimized code and conceptually alter the execution by adding back the deleted memory sub-operations and replacing new instructions by original instructions of the unoptimized code. The augmented execution is now comparable to the unoptimized execution, and the compiler optimizations may be deemed safe if the instructions and sub-operations of the augmented execution form an execution of the unoptimized code. We illustrate the above idea by discussing how an execution can be conceptually augmented to reason about the register allocation optimization. The concepts presented below can be extended to deal with other optimizations such as common sub-expression elimination or combinations thereof.

Consider a memory location that is allocated to a register within a given code interval. For simplicity, we assume a straightforward register allocation scheme. In the middle of this interval, register allocation transforms a memory read to a register read (*memory-read to register-read*) and a memory write to a register write with the same value (*memory-write to register-write*). The transformations at the two ends of the interval are slightly different. If the original interval starts with a read operation (a read-initiated interval), the memory read is still performed and the value read is written to a register (*memory-read plus register-write*). If the original interval starts with a write operation (a write-initiated interval), the memory write is simply transformed into a register write as is done with memory writes in the middle of the interval. Finally, if there are any writes to this location within the interval, the value of the register may be written back (or spilled) to memory (*register to memory write-back*).

We now describe how the execution of a program that has been register allocated may be appropriately augmented by adding the relevant memory operations back into the execution. We refer to R_{init} and W_{init} sub-operations as *initial sub-operations*. For a read, the remaining sub-operation is simply referred to as the *read sub-operation*. For a write, the remaining n sub-operations are referred to as *per-processor write sub-operations*. In transforming the optimized execution, we augment the operations from each processor independently. For each processor, we traverse its instruction instances in program order. The following enumerates the way the execution is augmented:

1. A *register read* on P_i that corresponds to a memory-read to register-read case is replaced with the R_{init}(i) and R(i) sub-operations to the corresponding memory location with the same return value as the register read.

2. A *register write* on P_i that corresponds to a memory-write to register-write case is replaced with an initial write sub-operation (W_{init}(i)) and the per-processor write sub-operations (i.e., W(1) ... W(n)) that write the same value to the corresponding memory location. For the last register write before the write-back in an interval, only the W_{init}(i) sub-operation is added to the execution (the remaining sub-operations will be contributed by the register to memory write-back event).

3. A W_{init}(i) sub-operation belonging to a write W on P_i that corresponds to a *register to memory write-back* case is removed from the execution (the previous case already adds a W_{init}(i) for the last register write before the write-back).

The sub-operations added by the above augmentation must be appropriately ordered with respect to other sub-operations in the augmented execution. The R_{init}(i) and R(i) sub-operations added by the first case above must be placed immediately after either the W_{init}(i) of the last conflicting memory write in program order or the R(i) of the last memory read to the same location which starts the interval (i.e., a memory-read plus register-write case), whichever is later in program order. The W_{init}(i) added by the second case above must be placed immediately after the R(i) or W_{init}(i) sub-operation of the last conflicting read or write (whichever is later) that is before the added write in program order. Finally, a per-processor write sub-operation W(j) added by the second case above must be placed immediately before the corresponding sub-operation W'(j) belonging to the next conflicting write-back write; multiple write sub-operations to the same memory copy corresponding to multiple register writes within an interval must be kept in program order. The intuitive effect of the above conditions is to push the W_{init}(i), R_{init}(i), and R(i) events to near the top and the W(i) events to the bottom of the register allocation interval.

Given the above augmented execution, the register allocation optimizations are considered safe if the instructions and sub-operations of the augmented execution form an execution of the unoptimized program with respect to the memory model assumed by the source program. As mentioned before, this methodology can be extended to reason about the correctness of other combinations of optimizations.

5.10.4 Determining Safe Compiler Optimizations

This section explores the optimizations that a compiler can safely perform assuming a given memory model. To determine whether an optimization is safe, the compiler must conceptually consider all possible executions for all possible sets of input data for the given program. Most compiler optimizations require full flexibility in reordering operations to different locations. Therefore, models such as SC, IBM-370, TSO, PC, and PSO disallow virtually all important compiler optimizations on shared memory operations because they do not provide the compiler with regions of instructions where read and write operations may be flexibly reordered with respect to one another. This is in contrast to hardware designs where limited flexibility such as allowing a read to bypass previous writes as in TSO can provide substantial gains through hiding the write latency.

As in the previous section, we will use register allocation as a canonical example of compiler optimizations that eliminate memory operations. Another assumption in this section is that cache coherence is supported by hardware, relieving the compiler from ensuring coherence in software. Finally, some of the specification conditions such as the coherence requirement and the atomicity condition for read-modify-write operations turn out not to be relevant to the compiler; this section considers only the relevant conditions.

Value, Initiation, and Uniprocessor Dependence Conditions

To satisfy the notion of uniprocessor data dependence upheld by the value, initiation, and uniprocessor dependence conditions, the compiler needs to ensure that if a read returns the value of its own processor's write, then the write is the last write before the read in program order. In the absence of optimizations such as register allocation, this is easily achieved if the compiler does not reorder a write followed (in program order) by a read to the same address. With register allocation, correctness is maintained if within the register allocation interval, the program order among conflicting register operations in the compiled program matches the program order among the corresponding memory operations in the original source program. In addition, the uniprocessor dependence condition requires program order among all conflicting operations to be maintained. The exception for most models is the order from a write followed by a read, thus allowing the read forwarding optimization. This relaxation is important because one of the effects of register allocation is to make it appear as if the read completes before the write and yet returns the value of the write. Overall, uniprocessor data dependence can be maintained in a similar way to a uniprocessor compiler.

The above conditions apply to operations with the same address. Since compiler optimizations are done at compile time, the compiler does not have access to dynamic runtime information about the addresses. Therefore, the compiler may sometimes have to make conservative assumptions about whether two operations conflict. On the other hand, there are cases where the compiler can do better than runtime hardware by using its knowledge about the program and data structures to determine that two operations cannot possibly conflict.

Multiprocessor Dependence Chains

The key issue with respect to enforcing multiprocessor dependence chains is to disallow the reordering of operations related by program orders that make up the different types of chains. For virtually all of the specifications, these program orders are identified by the various flavors of the $\xrightarrow{spo}$ relation.[75] Therefore, we will use $\xrightarrow{spo}$ to generically refer to the variety of program order arcs (and their transitive closures) that make up multiprocessor dependence chains.

The restrictions are pretty straightforward for optimizations that lead to a simple reordering of memory operations. Given X $\xrightarrow{spo}$ Y, the compiler must avoid reordering the two operations; otherwise, reordering is safe. For example, for a model such as PL1, non-competing operations that appear in the interval between competing operations may be freely reordered with respect one another since they are not related by the $\xrightarrow{spo}$ relation.

The restrictions for optimizations such as register allocation that end up eliminating memory operations is slightly different. The conservative condition is to (a) disallow replacing a memory read R with a register read if the value of the register is set prior to the previous operation ordered before R by $\xrightarrow{spo}$, and (b) disallow replacing a memory write W with a register write if the write-back operation corresponding to W (or a later write in program order) does not appear in program order before the next operation ordered after W by $\xrightarrow{spo}$. Given a region of non-competing operations in a model such as PL1, the above conditions allow operations to be freely register allocated within the region.

The above restrictions can be relaxed to allow register allocation across $\xrightarrow{spo}$'s in some cases. For example, given X $\xrightarrow{spo}$ R, R may be replaced by a register read if the value that it reads belongs to a write that is also

[75] The only two exceptions are the aggressive specification for SC and the specification for Alpha where two reads to the same address appear in a category 1 chain but are not included in $\xrightarrow{spo}$.

register allocated and whose value is not written back to memory until after X in program order. Similarly, given W $\xrightarrow{spo}$ Y, W may be replaced by a register write as long as there is a write-back to memory (of the value written by W or by a later write to the same location) before Y in program order; the value that remains in the register may still be used to replace read references after Y as long as all the other conditions allow this.

One of the important requirements for the compiler is to preserve the conservative operation labels or explicit fence instructions that appear in the source program in order to ensure the information remains visible to the underlying system that will execute the compiler code. This requirement places a few more restrictions on optimizations, such as register allocation, that eliminate memory operations or instructions. For example, register allocating a read R should be disallowed if the value of the register is set by another read R' and the label for R is more conservative than the label for R'. Similarly, register allocating a write W should be disallowed if the label for W is more conservative than the label of the write-back memory operation (e.g., consider the case where W is a competing operation and the write-back memory operation is non-competing, assuming the PL1 model). In addition, if X $\xrightarrow{spo}$ Y $\xrightarrow{spo}$ Z implies X $\xrightarrow{spo}$ Z through transitivity, and register allocating Y (i.e., eliminating the operation) makes X $\xrightarrow{spo}$ Z no longer hold, we must conservatively disallow the register allocation.[76]

Much of the above complexity can be eliminated if the compiler simply uses the more conservative approach for register allocation. Consider the PL1 model, for example. The conservative yet simple approach described earlier limits the use of register allocation to within regions of instructions between competing operations; the compiler can safely register allocate any operations within such regions. The more complex conditions allow the compiler to sometimes extend the register allocation region past competing operations. Yet, these aggressive conditions may not lead to a significant performance gain especially if competing operations occur infrequently in a program.

Termination and Other Related Conditions

The termination condition plays an important role in disallowing incorrect compiler optimizations by requiring the compiler to ensure that, for certain writes, the write operation occurs in all executions of the compiled code. For example, if a memory write is converted to a register write, the compiler must guarantee that the register value will eventually get written to memory. For most models, the termination condition applies to all writes; for RCsc, RCpc, and the three PL specifications, the condition only applies to competing writes.

The termination condition also interacts closely with the condition on execution orders ($\xrightarrow{xo}$) that requires the number of memory sub-operations ordered before any given sub-operation to be finite (see Definition 4.11 in Chapter 4). Intuitively, the two requirements together guarantee that certain writes will not only appear in the execution, but will also eventually be seen by reads in other processors. For example, consider a read that may possibly be issued an infinite number of times, such as the read of the Flag location in Figure 5.39. If this read is register allocated within a potentially infinite loop with a value from outside the loop, the old value for the location may be used in every iteration of the loop and new writes to this location that appear in the execution order would not affect the value returned. The above would violate the requirement on the execution order (assuming the augmented version of the execution where register reads are transformed back to memory reads) since it will appear as if there are an infinite number of read sub-operations ordered before

[76]Consider the specification for RCsc, for example. Assume Wc $\xrightarrow{po}$ Rc_acq $\xrightarrow{po}$ W, which implies Wc $\xrightarrow{spo}$ W. This latter relation may not hold if the acquire is register allocated. A compiler may use implementation-dependent information about the target hardware to more aggressively determine whether eliminating the middle operation ends up allowing the reordering of the other two operations.

P1		P2	
a1:	A = 1;	*a2:*	B = 1;
b1:	while (B == 0);	*b2:*	while (A == 0);

Figure 5.40: Example to illustrate the termination condition.

the new write sub-operation to the same memory copy. This explains how the register allocation of Flag in the example in Figure 5.39 is disallowed by the specification.

The uniprocessor correctness condition (Condition 4.1 in Chapter 4) also interacts with the termination condition to ensure that certain write operations occur in all executions of the compiled code. The uniprocessor correctness condition requires that memory sub-operations in the execution for each processor should correspond to a "correct" uniprocessor execution of the process' program given the reads return the same values as the multiprocessor execution. The intuitive notion of correctness for a uniprocessor compiler is for most part sufficient for maintaining this condition. However, there are certain aspects that are specific to multiprocessors. For example, an initial write sub-operation W_{init} that would occur in the correct uniprocessor execution is required to also occur in the multiprocessor execution. Similarly, if an operation A is followed by an operation B where B can execute for an infinite number of times, then the compiler cannot reorder B before A since this can cause A to never execute. Or, if a write operation D is preceded by an operation C where C can execute for an infinite number of times, then D cannot be reordered before C. This is because D may execute in the "optimized" execution while it would not have executed in the corresponding "unoptimized" execution. Appendix G describes how the latter constraint may be relaxed.

We use the code segment in Figure 5.40 to illustrate various scenarios that can result in the violation of the termination condition or the related uniprocessor correctness condition and the "finite" requirement on execution orders. Assume a model that requires the termination condition to hold for P1's write of A and for P2's write of B. The uniprocessor correctness, value, and termination conditions together require both loops to terminate in any possible executions of this code. However, any of the following optimizations may violate this property: (1) if the while loop on each processor is moved above the write on that processor, (2) if the writes of A and B are done as register writes and the new values are not written back to memory, or (3) if location B is register allocated on P1 or location A is register allocated on P2. The first optimization is disallowed by the uniprocessor correctness condition, the second by the termination condition, and the third by the definition of execution order which allows only a finite number of sub-operations (reads in this case) to precede any given sub-operation (the write in this case) in the execution order.

Appendix H describes a more aggressive form of the termination condition for the PL models, along with its implementation implications.

Reach Condition

The constraints imposed on the compiler by the reach condition are analogous to those imposed on hardware as discussed in Section 5.3.6. Consider a shared-memory write instruction W that follows (in program order) a shared-memory read instruction R in the unoptimized source code. If an instance of R could be ordered before an instance of W by $\xrightarrow{rch}$ in any execution of the unoptimized code (assuming the source memory model), then the compiler cannot move W to before R in the optimized code.

5.10.5 Summary of Compiler Issues

The major flexibility required for doing compiler optimizations is to allow read and write memory operations to be arbitrarily reordered with respect to their original program order. However, relaxing just the write-read order, as is done by TSO or PC, or the write-write order, as is done by PSO, fails to provide sufficient flexibility. Therefore, such models end up disallowing virtually all interesting compiler optimizations on shared memory operations. The only models with sufficient flexibility are those that provide regions of instructions where read-read, read-write, write-read, or write-write program orders are relaxed among operations to different addresses. This includes the label-based models WO, RCsc, RCpc, and the three PL models, and the fence-based models Alpha, RMO, and PowerPC. For these models, the compiler can identify regions of instructions where it can apply many of the same optimizations that are used by a uniprocessor compiler.

The way the flexibility provided by relaxed models is exploited by compilers is somewhat different from the way hardware typically exploits this flexibility. For example, the aggressive specifications of multiprocessor dependence chains (that do not impose orders at every point in the chain) are difficult to exploit for the compiler. Similarly, relaxing multiple copy atomicity constraints do not provide any additional optimizations to the compiler. On the other hand, extra information such as the relationship between data and synchronization (e.g., information on data that is covered by a lock synchronization) can be more easily exploited by a compiler.

Finally, most current compilers are inadequate for supporting explicitly parallel programs simply because they are really uniprocessor compilers and can therefore violate some of the constraints that are required for correctly compiling for multiprocessors. To remedy this situation, compilers must become aware of the memory model assumed by the source program and must use information such as labels or fences provided by the program to guide optimizations.

5.11 Summary

This chapter covered a wide range of issues related to correctly and efficiently implementing various memory consistency models in a multiprocessor system. The scope of the techniques covered points to the wide variety of implementation options with differing trade-offs in performance and complexity or cost. The challenge for a designer is to choose the design that provides close to the full performance potential for a model while limiting the complexity and cost of the design. The performance results provided in the next chapter shed further light on this important trade-off.

Chapter 6

Performance Evaluation

Choosing a memory consistency model, and selecting the appropriate implementation techniques for supporting a model, requires designers to balance the trade-offs between programmability, design complexity, and performance. Performance data for various models and implementations play a key role in helping designers make an informed decision on the above trade-off.

This chapter presents a detailed evaluation of the performance gains from exploiting the optimizations enabled by relaxed memory models in the context of large-scale shared-memory architectures with hardware support for cache coherence. Our results are based on detailed simulation studies of several parallel applications. Section 6.2 characterizes the performance of different models in architectures with *blocking reads*, which is typical for most current commercial systems. We consider the interaction of relaxed models with other latency hiding techniques, such as prefetching and hardware support for multiple contexts, in Section 6.3. Finally, Section 6.4 considers the effect of supporting *non-blocking reads*, which is becoming prevalent in many new generation processor designs. Related work on performance evaluations is discussed in Section 6.5.

Overall, our results show that the overlapping and reordering optimizations enabled by relaxed models are extremely effective in hiding virtually the full latency of write operations in architectures with blocking reads. Furthermore, we show that these optimizations complement the gains from other latency hiding techniques by further enhancing system performance. Finally, we show that a substantial fraction of the read latency can also be hidden if the architecture supports non-blocking reads. In each of the above cases, we identify the key architectural enhancements that are necessary for achieving these gains by considering implementations with different degrees of aggressiveness.

6.1 Overview

The main focus of this chapter is to quantify the performance gains from optimizations enabled by relaxed memory models in the context of scalable shared-memory multiprocessors with hardware support for cache

(a)	(b)	(c)	
Write	Write	Read	Read
Read	Write	Read	Write

Figure 6.1: Categories of program reordering optimizations.

coherence. More specifically, we concentrate on designs with an invalidation-based cache coherence protocol, which is by far the most common choice. Examples of such machines include the Stanford DASH [LLG+90], the MIT Alewife [ALKK90], and the Convex Exemplar.

Instead of providing performance results for specific memory models, we primarily present results for the basic optimizations that are enabled by different groups of models. This approach isolates the effect of different optimizations and provides better intuition for the important sources of performance gain. Furthermore, it is simple to map these results into performance gains for specific models, based on whether the optimization is allowed by the model and whether a given implementation exploits this optimization.

The two key optimizations enabled by relaxed models are the relaxation of various program orders and the relaxation of multiple-copy atomicity constraints for writes. Our study primarily focuses on program reordering optimizations. Since we assume an invalidation-based protocol, it is simple to support the illusion of multiple copy atomicity for writes (see Section 5.3.5 in Chapter 5); furthermore, relaxing this constraint is unlikely to enhance the performance of such a protocol.

Figure 6.1 depicts the program reordering optimizations, divided into three categories. These are the same categories used in Chapter 2 to describe the various relaxed memory models. The first category allows the reordering of read operations with respect to previous write operations. The second category allows the reordering of write operations. Finally, the third category allows the reordering of read operations with respect to future operations. While the first two optimizations can be exploited by most architectures, the latter optimization requires more sophisticated processor support in the form of *non-blocking reads*. Correspondingly, this chapter presents two distinct set of results, one for architectures with blocking reads and another for those with non-blocking reads.

6.2 Architectures with Blocking Reads

This section presents the performance results for architectures with blocking reads. We describe the framework for our simulation experiments in Section 6.2.1. The performance results for implementations that aggressively exploit write-read and write-write program reordering are presented in Section 6.2.2. Finally, Section 6.2.3 studies the effect of less aggressive implementations, and the effect of varying the cache and cache line sizes.

The performance results presented in this section extend the results that were published in a prior paper [GGH91a] by considering a larger set of applications and varying more architectural parameters.

6.2.1 Experimental Framework

This section describes the architectural assumptions, simulation environment, and benchmark applications used in our performance study.

Architectural Assumptions

To provide meaningful performance comparisons for the optimizations enabled by relaxed models, we need to focus on a specific class of multiprocessor architectures. The reason is that trade-offs may vary depending on the architecture chosen. For example, the trade-offs for a small bus-based multiprocessor with broadcast capability and miss latencies in tens of cycles are quite different than those in a larger scale multiprocessor where broadcast is not possible and miss latencies are in hundreds of cycles.

For our study, we have chosen an architecture that resembles the DASH shared-memory multiprocessor [LLG+90], a large scale cache-coherent machine that has been built at Stanford. Figure 6.2 shows the high-level organization of the simulated architecture, which consists of several processing nodes connected through a low-latency scalable interconnection network. Physical memory is distributed among the nodes and cache coherence is maintained using a distributed directory-based protocol. For each memory block, the directory keeps track of remote nodes caching the block, and point-to-point messages are sent to invalidate remote copies. Acknowledgement messages are used to inform the originating processing node when an invalidation has been completed. Although the DASH prototype has four processors within each node, we simulate an architecture with only one processor per node. This allows us to isolate the performance effect of optimizations more clearly.

All our simulation results are based on a 16 processor configuration. The parameters used are loosely based on the DASH prototype; we have removed some of the limitations that were imposed on the DASH prototype due to design time constraints. The bus bandwidth of the node bus is assumed to be 133 Mbytes/sec, along with a peak network bandwidth of 120 Mbytes/sec into and 120 Mbytes/sec out of each node. Figure 6.2 also shows the organization of the processor environment. Each processing node in the system consists of a 33MHz MIPS R3000/R3010 processor connected to a 64 Kbyte write-through primary data cache. The write-through cache enables processors to do single cycle write operations. The first-level data cache interfaces to a 256 Kbyte second-level write-back cache. The interface includes read and write buffers. Both the first and second-level caches are direct-mapped and support 16 byte lines. For most of this study, we assume an aggressive implementation with a lockup-free secondary cache, a 16 word deep write buffer, and reads that bypass the write buffer; results for more conservative implementations are presented in Section 6.2.3.

The following describes some features of the simulated cache-coherence protocol (for more details, refer to the original paper on the DASH protocol [LLG+90]); many of the terms used below were discussed in Section 5.3.5 of the previous chapter. As we mentioned before, the protocol is invalidation-based. Invalidations are acknowledged as soon as they arrive at the target cache's incoming buffer (i.e., early acknowledgement), and acknowledgements are gathered at the requesting node. The protocol maintains the illusion of multiple-copy atomicity for writes by nacking operations from other processors while there are invalidations pending for the line. Write operations always generate a read-exclusive request even if the second level cache has a clean copy of the line (i.e., the exclusive request optimization is not used). For implementations that allow multiple outstanding writes, the protocol supports eager read-exclusive replies; otherwise, read-exclusive replies are delayed until all invalidations are acknowledged.

The latency of a memory access in the simulated architecture depends on where in the memory hierarchy the access is serviced. Table 6.1 shows the latency for servicing an access at different levels of the hierarchy, in the absence of contention (the simulation results include the effect of contention, however). The latency shown for writes is the time for acquiring exclusive ownership of the line, which does not necessarily include

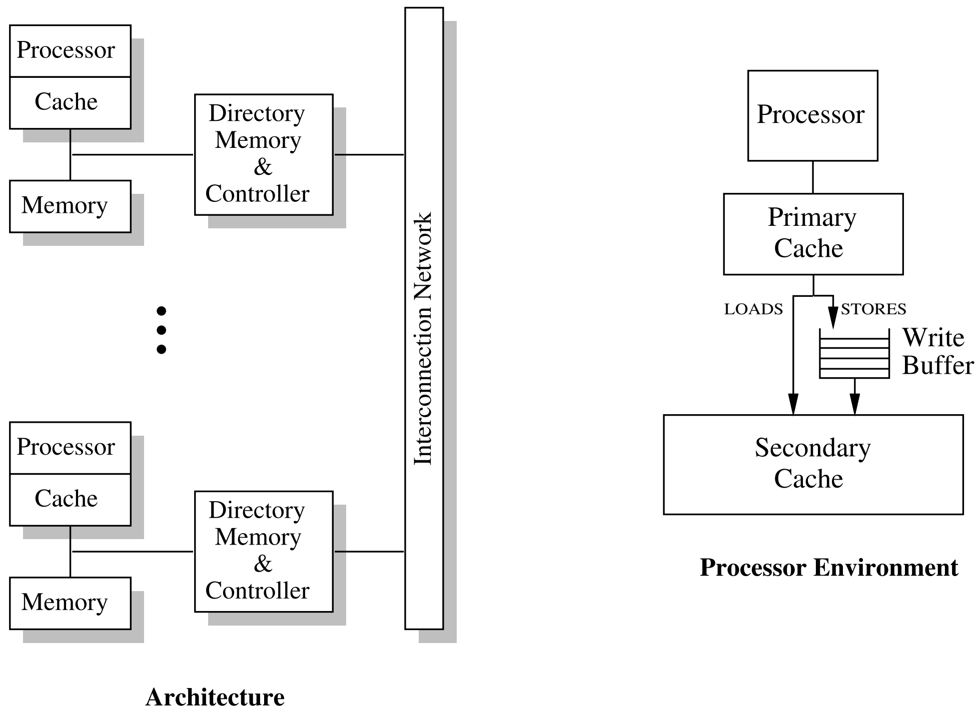


Figure 6.2: Simulated architecture and processor environment.

the time for receiving the acknowledgement messages for invalidations. The following naming convention is used for describing the memory hierarchy. The *local node* is the node that contains the processor originating a given request, while the *home node* is the node that contains the main memory and directory for the given physical memory address. A *remote node* is any other node.

Synchronization primitives are also modeled after DASH. The queue-based lock primitive [LLG$^+$90] is used for supporting locks. In general, locks are not cached except when a processor is spinning on a locked value. When the lock is released, if there are any waiting processors, one is chosen at random and is granted the lock using an update message. Acquiring a free lock takes approximately 20 and 100 processor cycles for local and remote locks, respectively. The total latency to perform a barrier for 16 processors, given all reach the barrier at the same time, is about 400 processor cycles; the typical simulated time for a barrier is much larger in practice since all processors do not reach the barrier at the same time.

Simulation Environment

We use an event-driven simulator that models the major components of the DASH architecture at a behavioral level. For example, the caches and the coherence protocol, contention, and arbitration for buses are all modeled in detail. The simulations are based on a 16 processor configuration. The architecture simulator is tightly coupled to the Tango-Lite reference generator, the light-weight thread-based version of Tango [GD90], to assure a correct interleaving of accesses. For example, a process doing a read operation is blocked until that read completes, where the latency of the read is determined by the architecture simulator. Instruction references are assumed to always hit in the instruction cache. Furthermore, operating system references are not modeled. Finally, main memory is distributed across all nodes and allocated using a round-robin scheme

Table 6.1: Latency for various memory system operations in processor clocks. Numbers are based on 33MHz processors.

Read Operations	
Hit in Primary Cache	1 pclock
Fill from Secondary Cache	15 pclock
Fill from Local Node	29 pclock
Fill from Remote Node	101 pclock
Fill from Dirty Remote, Remote Home	132 pclock
Write Operations	
Owned by Secondary Cache	2 pclock
Owned by Local Node	17 pclock
Owned in Remote Node	89 pclock
Owned in Dirty Remote, Remote Home	120 pclock

for the applications, unless specific directives are given by an application.

Given that our detailed simulator is significantly slower than the target multiprocessor that is being simulated, we can only afford to simulate smaller problem sizes relative to those that would be run on the actual machine. An important question that arises is how the simulated cache sizes must be scaled to approximate the behavior of larger problem sizes running on full-sized caches. We assume the full-sized caches (i.e., 64K first level and 256K second level) for most of our performance results. Since these cache sizes are large relative to the simulated problem sizes, most of the cache misses correspond to inherent communication misses. We later study the effect of smaller cache sizes in Section 6.2.3.

Benchmark Applications

The parallel applications we use consist of the entire SPLASH suite [SWG91] plus the LU-decomposition application that we used in earlier studies [GHG+91, MG91]. These applications are representative of algorithms used today in scientific and engineering computing environments. The applications use the synchronization primitives provided by the Argonne National Laboratory macro package [BBD+87]. All applications, except OCEAN, are written in C; OCEAN is written in Fortran. Table 6.2 provides a brief summary of the applications, along with their input data sets. We describe each application in more detail below.

OCEAN [SH92] models the role of eddy and boundary currents in influencing large-scale ocean movements. The simulation is performed for many time-steps until the eddies and mean ocean flow attain a mutual balance. The work in each time-step involves solving a set of spatial partial differential equations using Successive Over Relaxation (SOR). The principle data structure consists of a set of 25 two-dimensional arrays holding discretized values of the various values associated with the model's equations. For our experiments we simulated a 98-by-98 point square grid.

PTHOR [SG89] is a parallel distributed-time logic simulator based on the Chandy-Misra simulation algorithm. The primary data structures associated with the simulator are the logic elements (e.g., AND-gates, flip-flops), the nets (wires linking the elements), and the task queues which contain activated elements. Each processor executes the following loop. It removes an activated element from one of its task queues and determines the changes on that element's outputs. It then schedules the newly activated elements onto the

Table 6.2: Description of benchmark applications.

Application	Description	Input Data Set
OCEAN	simulates eddy currents in an ocean basin	98x98 grid
PTHOR	simulates digital circuit at the gate level	RISC (5 K elements), 5 steps
MP3D	simulates rarified hypersonic flow	100 K mols., cylinder.geom (6144 cells), 5 steps
CHOLESKY	sparse Cholesky factorization	bcsstk15
LU	dense LU decomposition with pivoting	200x200 matrix
LOCUS	routes wires for VLSI standard cell designs	Primary2 (25.8 K cells, 3817 wires)
BARNES	performs a hierarchical N-body gravitation simulation	8192 bodies, 3 steps
WATER	simulates water molecule interaction	512 mols., 2 steps

task queues. For our experiments we simulated five clock cycles of a small RISC processor consisting of about 5000 elements.

MP3D [MB88] is a 3-dimensional particle simulator used to study the pressure and temperature profiles created by an object flying at high speed through the upper atmosphere. The primary data objects in MP3D are the particles (representing the air molecules), and the space cells (representing the physical space, the boundary conditions, and the flying object). The overall computation of MP3D consists of evaluating the positions and velocities of molecules over a sequence of time steps. During each time step, the molecules are picked up one at a time and moved according to their velocity vectors. Collisions of molecules among themselves and with the object and the boundaries are all modeled. The simulator is well suited to parallelization because each molecule can be treated independently at each time step. The program is parallelized by statically dividing the particles equally among the processors. The main synchronization consists of barriers between each time step. Accesses to the space cells occur in an asynchronous manner within a time step; the algorithm is robust, however, due to the low frequency of particle collisions at a given space cell. For our experiments we ran MP3D with 100,000 particles in a 32x6x32 space array, and simulated 5 time steps.

CHOLESKY [RG90] performs sparse Cholesky factorization using a dynamic version of the supernodal fan-out method. The matrix is divided into supernodes (sets of columns with identical non-zero structures), which are further divided into conveniently-sized chunks called panels. A panel receives updates from other panels to its left, and is placed on a task queue after all updates have been received. Processors remove panels from this task queue to perform the associated modifications; this in turn causes other panels to be placed on the task queue. The principal data structure is the sparse matrix itself. The primary operation that is performed repeatedly is adding a multiple of one column to another column. Contention occurs for the task queue and the modified columns, which are protected by locks. For our experiments we ran `bcsstk15` which is a 3948-by-3948 matrix with 56,934 non-zeroes in the matrix and 647,274 non-zeroes in the factor.

LU performs LU-decomposition for dense matrices. The primary data structure in LU is the matrix being decomposed. Columns are statically assigned to the processors in an interleaved fashion. Each processor waits for the current pivot column, and then uses that column to modify all the columns that it owns. The processor that produces the current pivot column releases any processors waiting for that column. For our experiments we performed LU-decomposition on a 200x200 matrix.

LOCUS (abbreviation for "LocusRoute") [Ros88] is a high quality VLSI standard cell router. The program evaluates standard cell circuit placement with the objective of finding an efficient routing that minimizes area. The parallelism in LOCUS comes from routing multiple wires concurrently. Each processor continuously picks up a new wire from the task queue, explores alternative routes, and places the wire along the best route. The main data structure is a cost array that keeps track of the number of wires running through each routing cell of the circuit. Although the cost array is accessed and updated concurrently by several processors, it is not protected by locks since the resulting distortions are considered acceptable. For our experiments we used the largest circuit provided with the application, `Primary2.grin`, which contains 3817 wires and a 1290-by-20 cost array.

BARNES (abbreviation for "Barnes-Hut") [SHG92] is a hierarchical N-body gravitational simulation, with each body modeled as a point mass that exerts force on all other bodies in the system. For efficiency, a set of sufficiently far bodies are abstracted as a simple point mass. To facilitate this clustering, physical space is divided recursively to form an *oct-tree* until each cell contains at most one body. The octree is implemented as an array of bodies and an array of cells that are linked together to form a tree. Bodies are statically assigned to processors for the duration of a time-step. During each time-step, each processor calculates the forces of all bodies on its subset of bodies, and the bodies are then moved according to those forces. The tree is then regenerated for the next time step. A set of distributed locks provide exclusive access to the tree. For our experiments we ran BARNES with 8192 bodies through 3 time steps.

WATER is an N-body molecular dynamics application that evaluates forces and potentials in a system of water molecules in the liquid state. The main data structure is an array of molecules. Each processor is statically assigned a subset of the molecules. In each time step the program computes the potential of each molecule based on intra- and inter-molecular interactions. Due to symmetry, each processor only computes the interaction between a molecule it owns and half the other molecules. Synchronization in the program is maintained through locks on the molecules being modified, and through barriers that separate different phases in the program. We ran WATER with 512 molecules through 2 time steps.

Table 6.3 shows some general statistics for the eight applications. The numbers correspond to a 16 processor configuration, assuming the sequential consistency model (i.e., no program reordering optimizations). Table 6.4 presents more detailed statistics on shared data and synchronization access behavior. The read and write miss statistics correspond to misses past the second level cache. Except for LU, the hit rate for the combined caches is not substantially better than the rate achieved by the primary cache. The table also shows the frequency of accesses or misses relative to the total number of instructions executed.

6.2.2 Experimental Results

This section presents the performance results for architectures with blocking reads. We begin by presenting a base set of results for an implementation that preserves all program orders, as may be required by the sequential consistency model. We next study the effect of relaxing the write-read and write-write program orders. Finally, we consider the performance effect of minor ordering differences among relaxed models.

Table 6.3: General statistics on the applications. Numbers are aggregated for 16 processors.

Application	Instructions Executed *(millions)*	Shared Data References *(millions)*	Shared Data Size *(Kbytes)*
OCEAN	120	16.5	2972
PTHOR	86	15.8	2760
MP3D	209	22.4	4028
CHOLESKY	1302	217.2	6856
LU	50	8.2	640
LOCUS	897	130.3	5156
BARNES	337	44.9	1528
WATER	2165	195.3	484

Table 6.4: Statistics on shared data references and synchronization operations, aggregated for all 16 processors. Numbers in parentheses are rates given as references per thousand instructions.

Application	Reads *X1000*	Writes *X1000*	R/W Ratio	Read Misses *X1000*	Write Misses *X1000*	RMiss/WMiss Ratio	Locks	Barriers
OCEAN	12,280 (102)	4,255 (35)	2.9	1,438 (12)	1,810 (15)	0.8	352 (0.00)	2,400 (0.02)
PTHOR	14,516 (169)	1,316 (15)	11.0	550 (6.4)	237 (2.8)	2.3	105,414 (1.23)	3,984 (0.05)
MP3D	16,965 (81)	5,468 (26)	3.1	1,875 (9.0)	1,320 (6.3)	1.4	2,172 (0.01)	384 (0.00)
CHOLESKY	193,216 (148)	24,049 (19)	8.0	4,908 (3.8)	3,917 (3.0)	1.2	90,307 (0.07)	16 (0.00)
LU	5,478 (110)	2,727 (55)	2.0	273 (5.5)	123 (2.5)	2.2	3,184 (0.06)	29 (0.00)
LOCUS	117,440 (131)	12,847 (14)	9.1	2,721 (3.0)	1,845 (2.1)	1.5	51,016 (0.06)	16 (0.00)
BARNES	34,121 (101)	10,765 (32)	3.2	1,054 (3.1)	156 (0.5)	6.7	16,525 (0.05)	96 (0.00)
WATER	146,376 (68)	48,91 (23)	3.0	1,173 (0.5)	818 (0.4)	1.4	302,272 (0.14)	208 (0.00)

Preserving all Program Orders

This section evaluates the performance of implementations that preserve all program orders. The program order from a read to a following operation is inherently maintained because the processor stalls for the completion of each read (i.e., the blocking read restriction). To preserve the program order from writes to following operations, we assume the processor also stalls for each write operation to complete. The above depicts a straightforward implementation of the sequential consistency model.

Figure 6.3 shows the performance for the benchmark applications assuming all program orders are preserved. The figure presents a breakdown of the execution time for each application. The bottom component of each column represents the time spent executing instructions. This component provides a measure of processor utilization or efficiency. The remaining three components represent time spent stalling for read, write, and synchronization operations to complete. The synchronization component includes the time spent acquiring a lock, waiting for other processors to reach a barrier, or spinning on empty task queues.[1] We have

[1]The latter type of spinning is achieved through normal instructions. However, the applications are appropriately annotated to

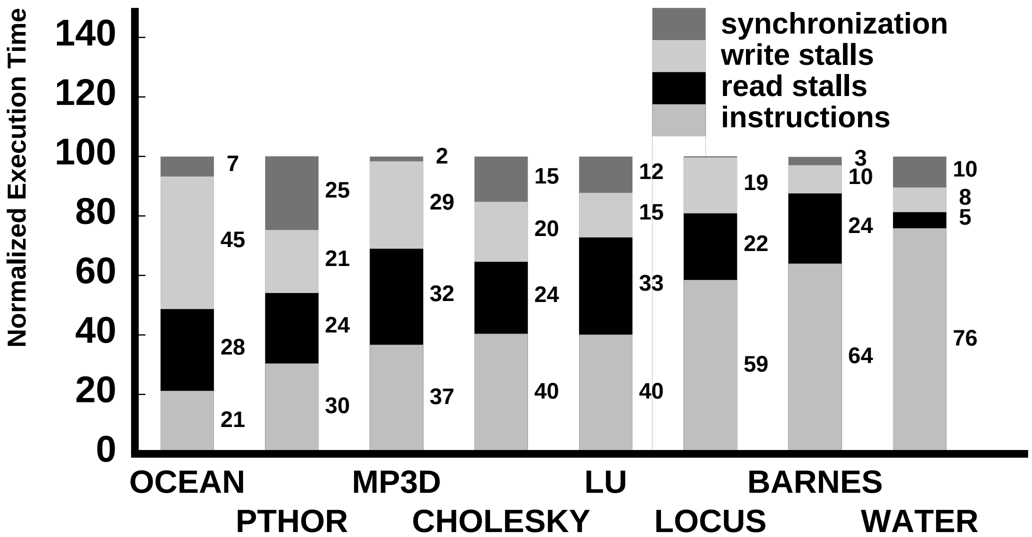


Figure 6.3: Performance of applications with all program orders preserved.

shown the applications in the order of increasing processor utilization. In general, the applications with lower processor utilizations benefit more from optimizations that hide the latency of memory operations.

As shown in Figure 6.3, many of the applications spend large portions of their execution time stalled for memory operations and synchronization. As a consequence, five out of eight applications achieve below 40% processor utilization. Referring back to Table 6.4, the time spent stalling for reads and writes is closely related to the frequency of misses for these operations. The correlation between synchronization operation frequency and the time spent on synchronization is much weaker because synchronization latency is often dominated by load balance issues. For example, even though LU and LOCUS have approximately the same frequency for locks and barriers, LU spends a much larger percentage of its execution time stalled for synchronization (mainly busy locks).

The program order from writes to following operations can be maintained in a slightly more efficient way. In the implementation described above (referred to as BASE), the processor stalls immediately after each write until the write completes. The alternative is to place the writes in the write buffer without stalling the processor. To maintain the order among write operations, the write buffer retires the write at the head only after the write is complete. Furthermore, the write-read program order is preserved by stalling a read until the write buffer is empty. This latter implementation (referred to as BASE') allows part of the write latency to be overlapped with computation up to the next read. In most cases, however, a read access occurs soon after a write miss, and most of the latency for completing the write miss will still be seen by the processor.

Figure 6.4 shows the performance of the alternative implementation described above. For each application, we show the breakdown of the execution time for both BASE and BASE' (labeled B and B', respectively), normalized to the execution time for BASE. The small variation in the instruction time for CHOLESKY is due to the slightly non-deterministic behavior of the application. We will later observe the same effect in some of the other applications. As expected, the performance of BASE' is within a few percent of BASE

delineate regions of code that are used for such synchronization purposes.

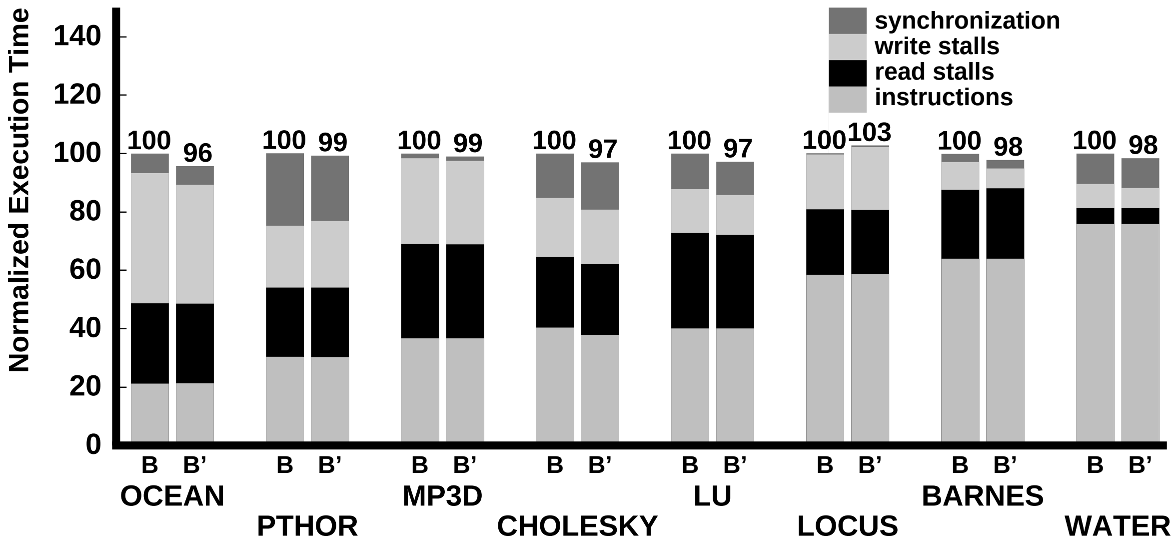


Figure 6.4: Effect of buffering writes while preserving program orders.

for all applications. BASE' performs better than BASE for all applications except LOCUS, with the gain mainly due to a slight decrease in the write stall time. LOCUS and PTHOR experience a higher write stall time, however. This latter effect is simply an artifact caused by the different interleaving of operations in the two executions. Overall, the performance difference between the two implementations is negligible. For this reason, we will continue to use the BASE implementation as a point of comparison.

Relaxing the Write-Read Program Ordering

The first reordering optimization we consider is relaxing the write-read program order, which is enabled by models such as TSO and PC. We begin by describing the implementation requirements for supporting this optimization. As before, the processor stalls for a read to complete. Writes, on the other hand, are simply sent to the write buffer. The only time the processor directly stalls on a write is if the write buffer is full. The program order among writes is maintained through the write buffer by retiring a write only after it completes. Relaxing the write-read program order is exploited by allowing reads to bypass pending writes in the write buffer; in case of an address match with a pending write, the value of the latest such write is forwarded to the read. To fully exploit the overlap of a read with previous writes, the secondary cache (refer to Figure 6.2) must be lockup-free. In this way, even if the cache is currently handling a write miss, it can service a later read (hit or miss) operation. This is a limited form of lockup-free behavior, however, since there can at most be a single write miss and a single read miss outstanding at any given time (due to write-write ordering and blocking read constraints, respectively).

The main effect of relaxing the write-read program ordering is to allow a read to complete without waiting for previous outstanding writes. Figure 6.5 presents the results of the above relaxation. For each application, we show the execution time for the write-read reodering optimization (labeled W-R) normalized to the time for the BASE implementation described in the previous section (labeled B). The visible variation in the instruction times for PTHOR, CHOLESKY, and WATER are due to the non-deterministic behavior

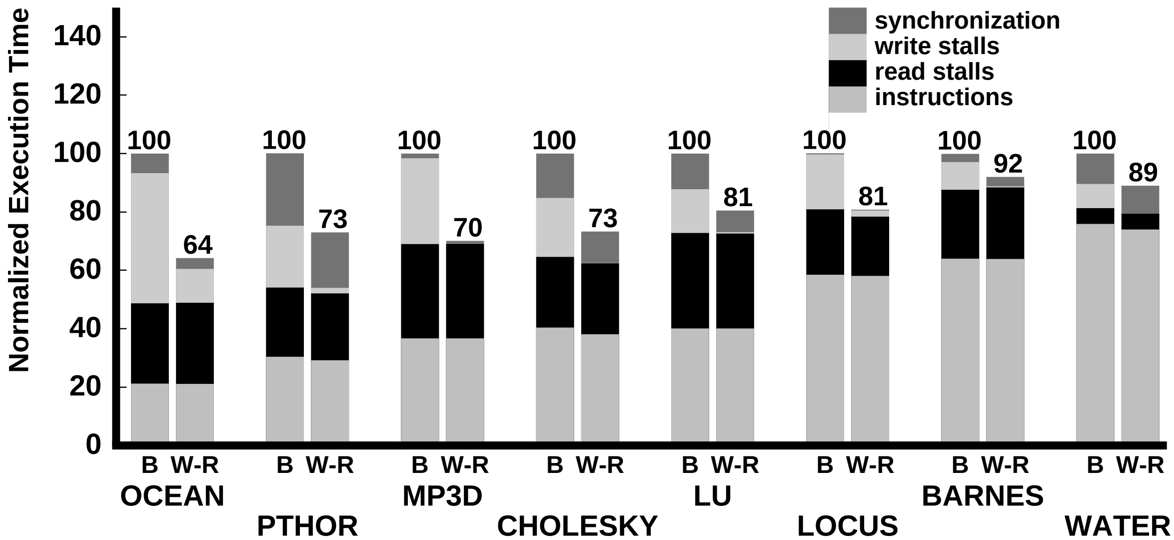


Figure 6.5: Effect of relaxing write-read program ordering.

mentioned in the previous section.

The performance results in the figure show a significant benefit from this relaxation, ranging from approximately 10% to 55% improvement in performance. This improvement arises mainly from a significant reduction in the write stall time; the write stall time is virtually eliminated in all the applications except OCEAN. Any leftover write stall time arises from the write buffer filling up which in turn stalls the processor on its next write. Referring back to Table 6.4, the magnitude of gains is highly correlated to the frequency of write misses in the application.

Figure 6.5 also depicts a number of secondary effects. One effect is that the read stall time increases in some of the applications (e.g., BARNES). The primary reason for this is that approximately the same number of read misses are executed in a shorter time compared to BASE, therefore increasing contention at the memory system which can lead to increased latencies. Consequently, some of the savings due to hiding write latencies are lost. On the other hand, we notice that synchronization times decrease in most applications relative to BASE, which in turn magnifies the gains from hiding write latencies. Empirically, we have noticed that fewer memory stalls often leads to fewer synchronization stalls. This effect may possibly be due to a decrease in variation of times for different processors to arrive at the synchronization points.

We now return to the issue of the write buffer becoming full. The results show that, except for OCEAN, the write buffer rarely fills up in any of the other applications. The reason for this is fairly intuitive: write misses are well interleaved with read misses rather than being clustered, and the number of read misses usually dominates. Since the processor has blocking reads, the stall duration for read misses provides sufficient time for write misses to be retired from the write buffer, thus preventing it from getting full. OCEAN is the only application where read misses are outnumbered by write misses (refer to Table 6.4). This makes the stall duration for reads insufficient for fully covering the service time for the write buffer; increasing the size of the write buffer does not help in this case. Overall, the write buffer depth of 16 in our simulated architecture seems more than sufficient for handling the clustering of writes present in the applications.

In summary, relaxing the write-read program ordering enables an implementation to successfully hide virtually all of the write latency in most applications.

Relaxing the Write-Write Program Ordering

We now consider the effect of relaxing write-write program ordering in addition to the write-read program ordering. This type of relaxation is allowed by a model like PSO. Furthermore, this relaxation characterizes the behavior of models such as RCpc and RMO in architectures with blocking reads; even though these latter models relax the program order from reads to later operations, this optimization cannot be exploited in the context of blocking reads.

We begin by describing the implementation requirements for exploiting the pipelining of writes. There are two key differences compared to the implementation described in the previous section. The *first* difference is that the write buffer can retire a write without requiring ownership to be obtained for the write. Some writes, such as writes following a STBAR in PSO or a competing write in RCpc, must still be ordered with respect to previous writes. To achieve this, the write buffer delays the issue and retiring of such writes (when they arrive at the head of the buffer) until all previous writes have completed. As discussed in the previous chapter, the implementation may use counters to keep track of the outstanding writes. The *second* difference is with respect to the secondary cache in terms of its lockup-free behavior. The cache must now allow multiple outstanding writes; the blocking read restriction still limits the number of outstanding reads to one, however. Our simulation limits each lockup-free cache to a maximum eight outstanding misses (i.e., eight write misses, or one read miss and seven write misses).

The effect of pipelining writes is that writes can be retired at a much faster rate than is possible in the implementation studied in the previous section. This can help performance in two ways. First, the chances of the write buffer filling up and stalling the processor are smaller. Second, if there is a write synchronization operation (e.g., unlock operation) behind several writes in the write buffer, then a remote processor trying to acquire the synchronization (e.g., lock on the same variable) can observe the write sooner, thus reducing the synchronization stall time. Even though the write synchronization operation is still delayed until all previous writes complete, this delay is often reduced because the previous writes are serviced in an overlapped manner. Therefore, synchronizations that are on the critical path may occur faster.

Figure 6.6 presents the results for the above relaxation. For each application, we show the execution time for the BASE implementation (labeled B), the write-read reordering optimization from the previous section (labeled W-R), and the write-write plus write-read reordering optimizations (labeled W-W). As before, the execution times are normalized to that of BASE for each application. Compared to the W-R case, W-W eliminates virtually any leftover write stall time. Even in OCEAN, the faster rate of service for writes effectively eliminates stalls due to a full write buffer. In addition, the synchronization time is reduced in some of the applications. Except for OCEAN, the performance of W-R and W-W are almost identical since W-R already eliminates almost all of the write stall time for most of the applications. The fact that CHOLESKY performs slightly worse under W-W is simply an artifact of the different interleavings in the two executions causing the synchronization time to increase by a small amount.

In summary, relaxing the write-write program ordering helps eliminate virtually all of the write latency in all applications, even applications such as OCEAN which exhibit more write misses than read misses. The performance improvement relative to the BASE implementation ranges from approximately 10% to 80%, with six out of eight applications gaining over 25%.

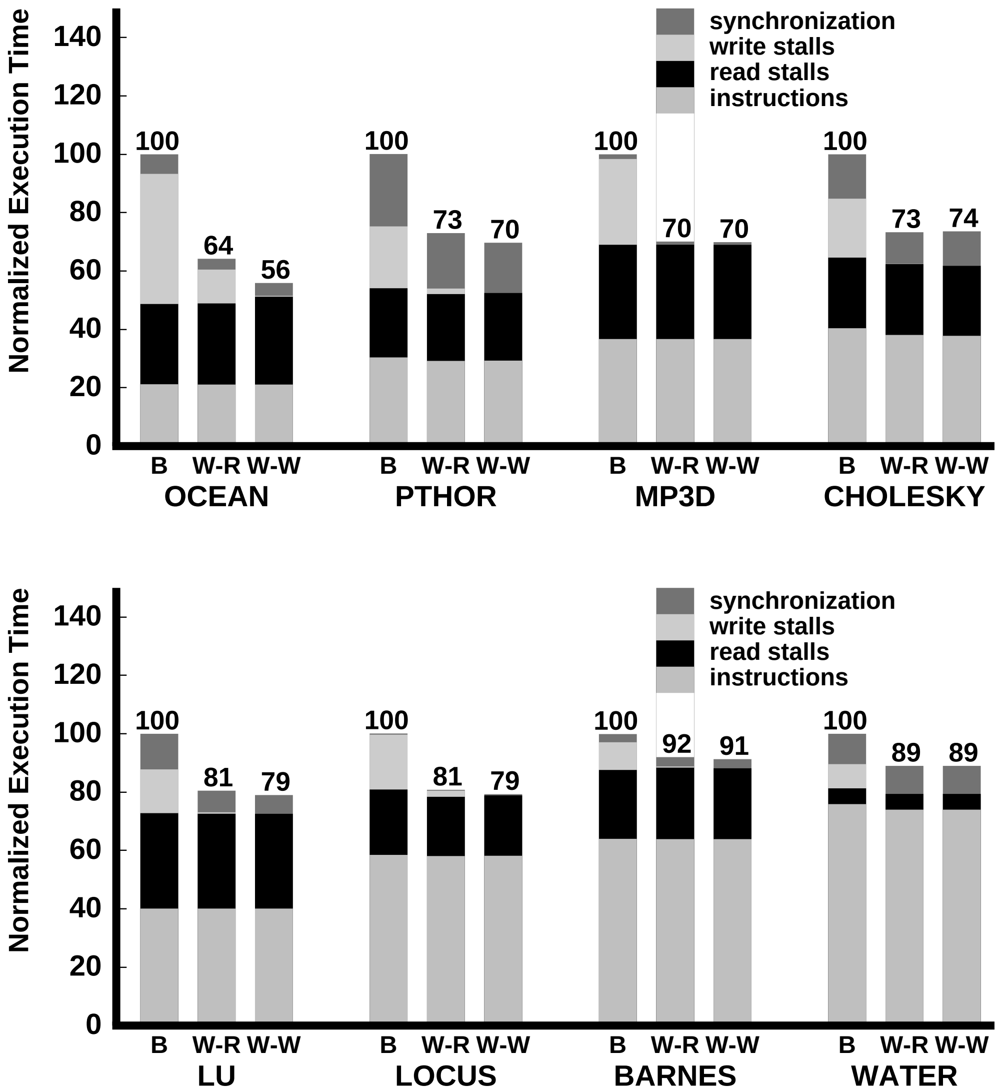


Figure 6.6: Effect of relaxing write-write and write-read program ordering.

Effect of Subtle Differences Among Models

In studying the effect of relaxing write-read program ordering, we have assumed implementations that allow a read to *always* bypass previous writes that do not conflict with the read. Models such as TSO, PC, PSO, and RCpc allow such an implementation. However, a model such as WO does not allow a read to be reordered with respect to a previous write if either the write or the read are labeled as synchronization. The Alpha, RMO, and PowerPC models also disallow such reordering if there is a fence in between the write and the read. Finally, the RCsc, PL1, PL2, and PL3 models all exhibit cases where the write-read order must be preserved when both operations are conservatively labeled (e.g., competing write followed by competing read in PL1).

Similarly, for the write-write program ordering, we have assumed that the only case when two writes to different addresses must be ordered is if the later write in program order has a conservative label or is preceded by a fence (depending on whether we are dealing with label-based or fence-based models). Except WO, all models discussed in this thesis that relax write-write reordering allow such an implementation; WO does not allow a synchronization write to be reordered with respect to either previous or future writes.

The above differences among models become significant only if the frequency of fences or conservative labels is high. In the context of our applications, this translates to a high synchronization rate. However, as the data in Table 6.4 shows, the frequency of synchronization operations is extremely low in most of applications in our study. The only application that shows a relatively high rate of synchronization is PTHOR, with an average of 1.2 lock (and 1.2 unlock) operations occurring every thousand instructions. Therefore, we do not expect these secondary differences to affect the performance of most of the applications.

Instead of attempting to show performance results for all the models, we try to place a bound on the performance effect of the types of differences discussed above. For this purpose, we have selected two models, WO and RCpc, that lie on the extremes with respect to such differences. The performance of other models (that allow similar optimizations) will lie somewhere in between these two bounds. The implementation assumed for RCpc is the same as the W-W implementation studied in the previous section. The WO implementation has a few extra constraints. In particular, the processor stalls for previous writes to complete before issuing a synchronization read (e.g., lock). In addition, the write buffer does not retire a synchronization write (e.g., unlock) until it completes. This latter constraint is less significant since it influences performance only indirectly by potentially increasing the chances of a full write buffer.

Figure 6.7 presents the performance of the WO and RCpc models for the eight applications. The results for RCpc are the same as those shown in Figure 6.6 for the W-W implementation. As before, the execution times are normalized to that of BASE; however, we no longer show the breakdown for the BASE case. As expected, the performance of the two models is identical for most applications. For PTHOR, the RCpc model performs significantly better than WO due to the high synchronization rate in this application. Surprisingly, the performance of WO for PTHOR is also worse than the performance of the W-R implementation (compare with normalized execution times in Figure 6.5), which is allowed by less relaxed models such as TSO and PC that do not allow write pipelining. Finally, WO performs slightly worse than RCpc for OCEAN and WATER.

To understand the performance of PTHOR better, we gathered data regarding the extra stalls introduced by WO over RCpc. We found that there is frequently a write miss or unlock within 20 cycles before a lock operation. Similarly, there is a read within a small number of cycles after an unlock. Therefore, at most 20 cycles of the latency of the write miss or unlock can be hidden and the rest of the latency is visible to the processor. More detailed data show that virtually all unlocks caused a delay for the next read and 30% of the

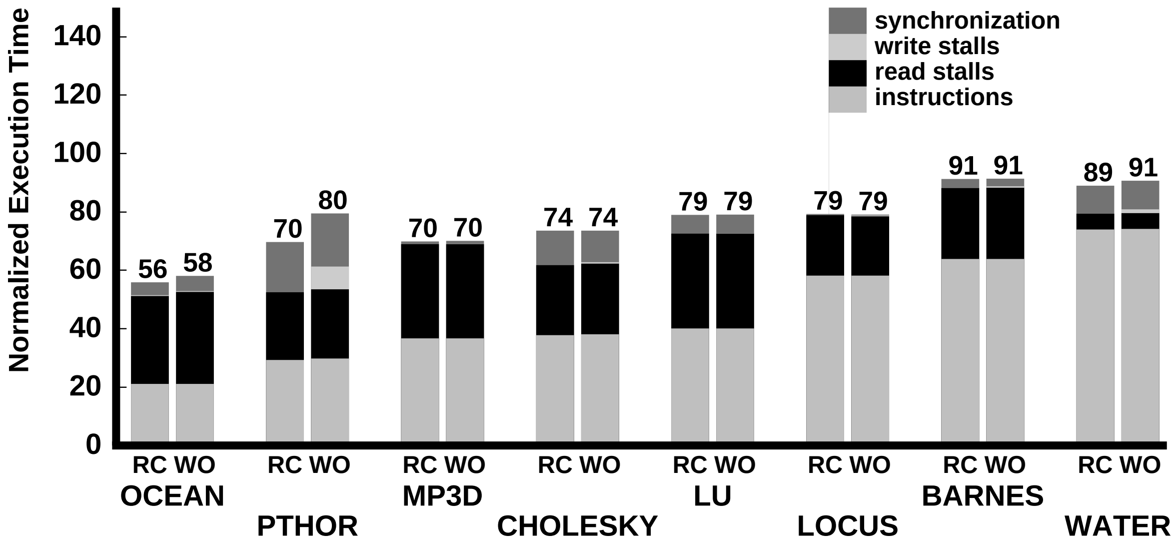


Figure 6.7: Effect of differences between the WO and RCpc models.

lock operations were delayed due to previous write misses. For the above reasons, WO fails to completely eliminate the write stall time, with RCpc performing 14% faster.

The relative performance of WO as compared to models such as TSO, PC, and RCpc may be affected by the fact that our simulated architecture does not cache locks, which increases the average latency for acquiring or releasing locks. Caching locks is beneficial when a processor acquires and releases a lock several times with no other processor accessing the lock in between. All models gain from a faster acquire if the lock is found in the cache. As for the reduced latency of the release, TSO, PC, and RCpc do not benefit since they already hide write and release latencies. WO can potentially benefit from this, however, which may reduce the performance difference observed for PTHOR.

In summary, the subtle differences among models do not have a significant affect on performance for most of the studied applications. The intuitive reason for this is that these subtle differences become significant only if the application exhibits a high frequency of synchronization.

6.2.3 Effect of Varying Architectural Assumptions

This section studies the effect of less aggressive implementations, and also considers the effect of different cache and line sizes.

Less Aggressive Implementations

In the previous section, we evaluated the performance of program reordering optimizations using an aggressive implementation with lockup-free caches. The goal was to minimize the influence of the implementation in the comparison of the optimizations. In this section, we explore less aggressive implementations and study the impact on the relative performance of the write-read and write-write reordering optimizations.

We will consider four version of the implementation: (i) LFB (lockup-free with bypass), the aggressive implementation studied in the previous section with a lockup-free secondary cache, a 16 word deep write buffer, and reads that bypass the write buffer; (ii) LFN (lockup-free with no bypass), which is the same as LFB, except reads are not allowed to bypass the write buffer; (iii) BLB (blocking cache with bypass), which is the same as LFB, except the secondary cache is blocking (i.e., not lockup-free); and (iv) BLN (blocking cache with no bypass), which is the same as BLB, except that reads are not allowed to bypass the write buffer. Recall that the lockup-free cache must only support a single outstanding read and a single outstanding write at any given time when we consider the write-read reordering optimization. The write-write reordering optimization requires the cache to allow multiple outstanding writes.

Effect on the Write-Read Reordering Optimization Figure 6.8 presents the performance results for the various implementations of the write-read reordering optimization. The LFB results are the same as the W-R results presented in Section 6.2.2. The execution time for each application is normalized to that of the base implementation (labeled B) that preserves all program orders. For several of the applications, there is a significant performance loss when we move from LFB to less aggressive implementations. This result shows that supporting both a lockup-free cache and read bypassing is essential for realizing the full potential of the write-read reordering optimization.

Comparing the LFB and LFN implementations, we see a large performance difference in several of the applications. The only difference between the two implementations is that LFB allows reads to bypass the write buffer. Because writes are serviced one at a time in both implementations, it is likely that there is one or more writes in the write buffer when a processor tries to issue a read to the secondary cache. In OCEAN, which has a very high write miss rate, almost 70% of the primary cache read misses run into at least one or more writes in the write buffer. Therefore, disallowing the read to bypass these writes can substantially increase the time required to service each read.

Comparing LFB and BLB, we also see a large performance difference for several applications. The only difference between these two is that LFB's secondary cache can have both a read miss and a write miss outstanding, while BLB can have either a read miss or a write miss outstanding but not both. Thus, in BLB, a read miss that follows closely behind a write miss that is already being serviced by the secondary cache will have to wait until the write completes, which can be tens of cycles. The detailed results for LFB executing OCEAN and PTHOR show that 73% and 22% (respectively) of the read misses are serviced while there was a write miss outstanding.

Comparing BLN, BLB, and LFN, we see less of a difference in performance. For many applications, the difference among these implementations is negligible. However, in OCEAN and PTHOR, BLB performs somewhat better than the other two. The obvious advantage of allowing bypassing is that the read miss does not have to wait for the write buffer to empty before being serviced. The results for OCEAN and PTHOR suggest that read bypassing is more important than a lockup-free cache if only one is to be supported.

Finally, the reason BLN still performs better than the base implementation (labeled B) is as follows. In B, both first-level cache read hits and read misses are delayed for the write buffer to empty out. In BLN, read hits in the first-level cache are not delayed for pending writes and only first-level read misses suffer that penalty. Therefore, BLN still exploits write-read reordering in a limited way.

In summary, implementations that exploit only write-read program reordering must support both a lockup-free cache and reads that bypass the write buffer to get the full benefit from this relaxation.

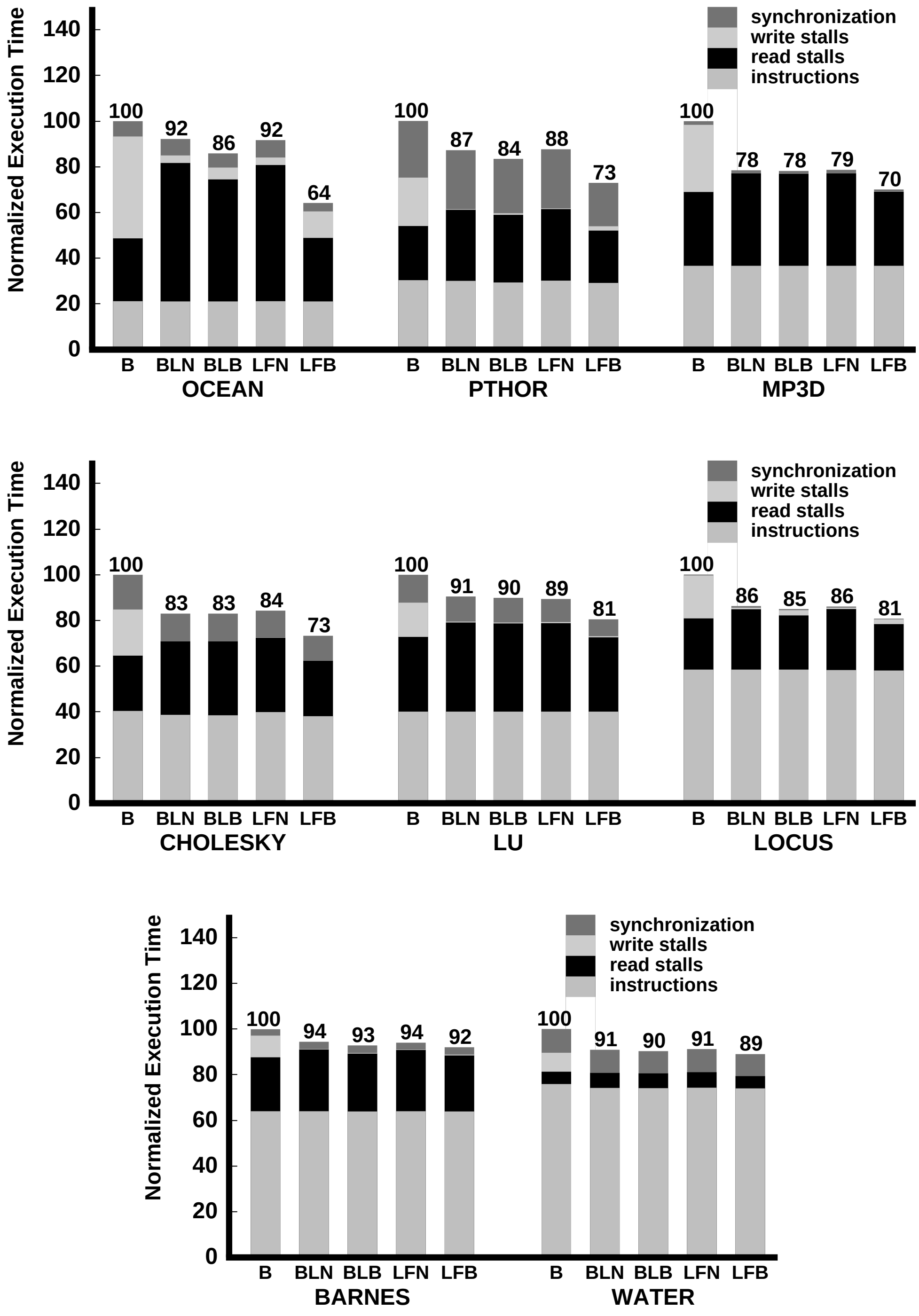


Figure 6.8: Write-read reordering with less aggressive implementations.

Effect on the Write-Write Reordering Optimization Figure 6.9 presents the performance results for implementations that exploit both the write-read and the write-write reordering optimizations. The LFB results are the same as the W-W results presented in Section 6.2.2. Again, the execution time for each application is normalized to that of the base implementation (labeled B) that preserves all program orders.

In contrast to the results in the previous section, we see that the performance of LFB and LFN are comparable. Exploiting write-write reordering allows a much faster service rate for the write buffer, especially when the implementation supports multiple outstanding writes with a lockup-free cache. This significantly decreases the chance that a primary read miss encounters a write in the write buffer. For example, over 99.9% of the primary read misses in OCEAN encounter an empty write buffer under LFN, while without write pipelining (i.e., the LFN implementation of the previous section), over 70% of the read misses encountered at least a single pending write. For this reason, disallowing reads to bypass the write buffer does not have a significant effect on performance. Similarly, a more shallow write buffer would suffice.

The significant performance loss occurs when the implementation no longer supports a lockup-free cache (i.e., going from LFB or LFN to BLN or BLB). As in the previous section, the lockup-free cache allows a read miss to be serviced right away regardless of whether a previous write miss is currently being serviced by the cache. Furthermore, as we mentioned above, write misses in the write buffer are retired at a much faster rate since the cache allows multiple outstanding writes.

The comparison of BLN and BLB is similar to the previous section. For most applications, allowing read bypassing with a blocking read does not provide a gain. However, read bypassing is useful in applications such as OCEAN and PTHOR which have high write miss rates. It is also interesting to compare the results for BLN or BLB with the results in the previous section. Even though the secondary cache is blocking, write-write reordering can still be exploited by using eager read-exclusive replies. The write buffer retires a writes as soon as the read-exclusive reply is back and potentially before the invalidations are acknowledged. This allows a slightly faster service rate for the write buffer. Furthermore, the blocking cache is kept busy for a shorter duration on each write miss since it also only waits for the read-exclusive reply and not the invalidation acknowledgements. Therefore, the BLN and BLB implementation in Figure 6.9 perform better than the BLN and BLB implementations in Figure 6.8 (which do not exploit write-write reordering).

In summary, implementations that exploit both write-read and write-write program reordering must support a lockup-free cache to get the full benefit from these relaxations. Allowing reads to bypass the write buffer does not provide much benefit once the design supports a lockup-free cache; the fast service rate for the writes also alleviates the need for a deep write buffer.

Cache Size Variations

This section studies the effects of varying the cache size. Varying the cache size typically changes the absolute number of misses, the ratio of read versus write misses, and the relative mixture of capacity/conflict versus coherence (or communication) misses. The results will allow us to observe the behavior of the applications and to evaluate the effectiveness of the reordering optimizations with respect to different cache sizes.

Figure 6.10 presents the results, with the performance of each application shown for three different sets of cache sizes. The rightmost pair of bars correspond to the full-sized 64K/256K caches in DASH that were used throughout the previous sections (same as results in Figure 6.6). The leftmost and middle pairs of bars correspond to 2K/8K and 8K/64K caches, respectively. For each application, the results are normalized to the

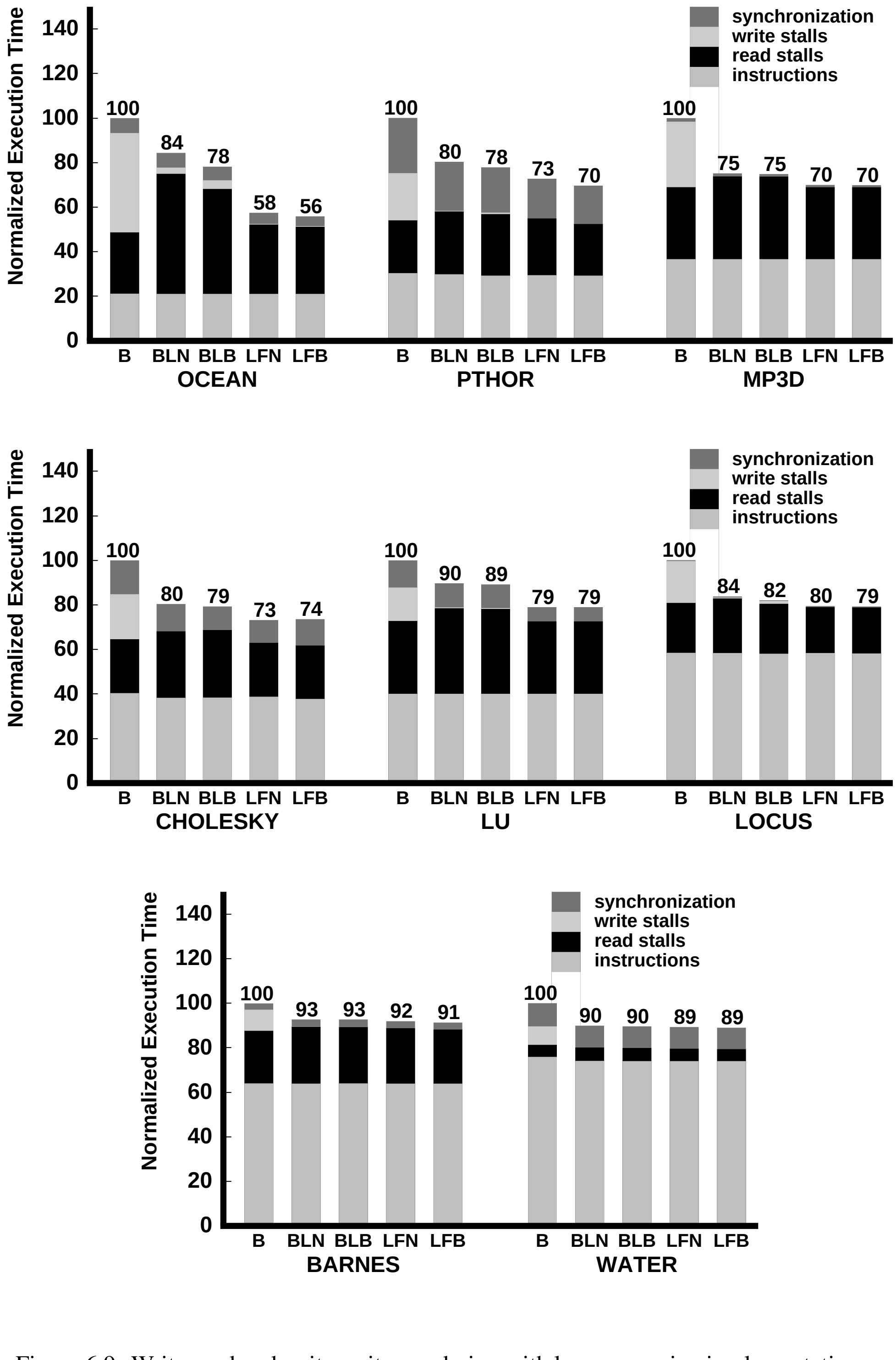


Figure 6.9: Write-read and write-write reordering with less aggressive implementations.

performance of the base implementation running with the full-sized caches. In addition to showing results for the base implementation with no reordering optimizations (labeled B), we also show results for the aggressive implementation with write-read and write-write reordering (labeled W-W) that we studied in Figure 6.6. The percent number beside each pair of bars shows the relative performance gain of W-W over B for each cache size.

We begin by considering the effect of varying the cache size on the performance of each application running on the base implementation. At one extreme, varying the cache size has virtually no impact on the performance of MP3D. The reason for this is that MP3D sweeps through a large data set on each time step (the particles for each processor are 200 Kbytes, and the space cell array is 288 Kbytes), and even the 64K/256K caches are not large enough to capture this data set. Real-life runs of MP3D would have even larger data sets, and therefore, our results are indicative of runs of this application on a real machine with even larger caches. At the other extreme, applications such as CHOLESKY, LU, LOCUS and BARNES, exhibit significant increase in performance once key data structures fit in the cache. For LU, the knee of the performance curve occurs between the 8K/64K and 64/256K cache sizes, corresponding to when the columns owned by a processor (approximately 20 Kbytes) fit in the primary cache. For CHOLESKY, LOCUS, and BARNES, the knee occurs between the 2K/8K and 8K/64K cache sizes. The other applications exhibit a substantially less dramatic change in performance as we vary the cache size.

We next consider the effect on each component of the execution time. The absolute time to execute instructions is effectively constant as we vary the cache size for each applications. The read and write stall times typically decrease as we move to larger caches due to a corresponding reduction in misses. The synchronization component varies a little bit more irregularly; for example, even though the read and write stall components decrease in WATER as we go from the small to the medium sized caches, the increase in synchronization time makes the performance of the medium sized cache worse.

The W-W implementation exhibits the same behavior we observed in Figure 6.6: it effectively eliminates the write stall component of the execution time at all cache sizes. The relative gain from the reordering optimizations is a bit harder to predict since it depends on the relative mix of read stall, write stall, and synchronization times as we vary the cache size. The percentage gain over the base implementation is shown in italics beside each pair of bars. For four of the applications (i.e., OCEAN, PTHOR, CHOLESKY, and LOCUS), the relative gain from hiding the write latency actually increases as we move to larger caches even though the absolute number of read and write misses actually decrease. As is especially evident in CHOLESKY, increasing the cache size end up reducing the absolute read stall time much more substantially compared to the write stall time. Therefore, the write stall component occupies a larger percentage of the execution time as we increase the cache size. The relative gain is unaffected in MP3D since the application is insensitive to the cache size variation. Finally, LU, BARNES, and WATER exhibit a slightly less regular behavior. LU is the only application where there is a substantial reduction in the gains from hiding write latency as we move from the small and medium sized caches to the larger cache; nevertheless, the gain at the larger cache size (27%) is still significant.

In summary, the write-read and write-write reordering optimizations are successful in hiding virtually all of the write latency irrespective of the cache size. One somewhat non-intuitive result is that the gain from hiding write latency actually increases with the cache size in four of the applications; the larger caches are more effective in reducing the read stall time, thus increasing the relative importance of the write stall time.

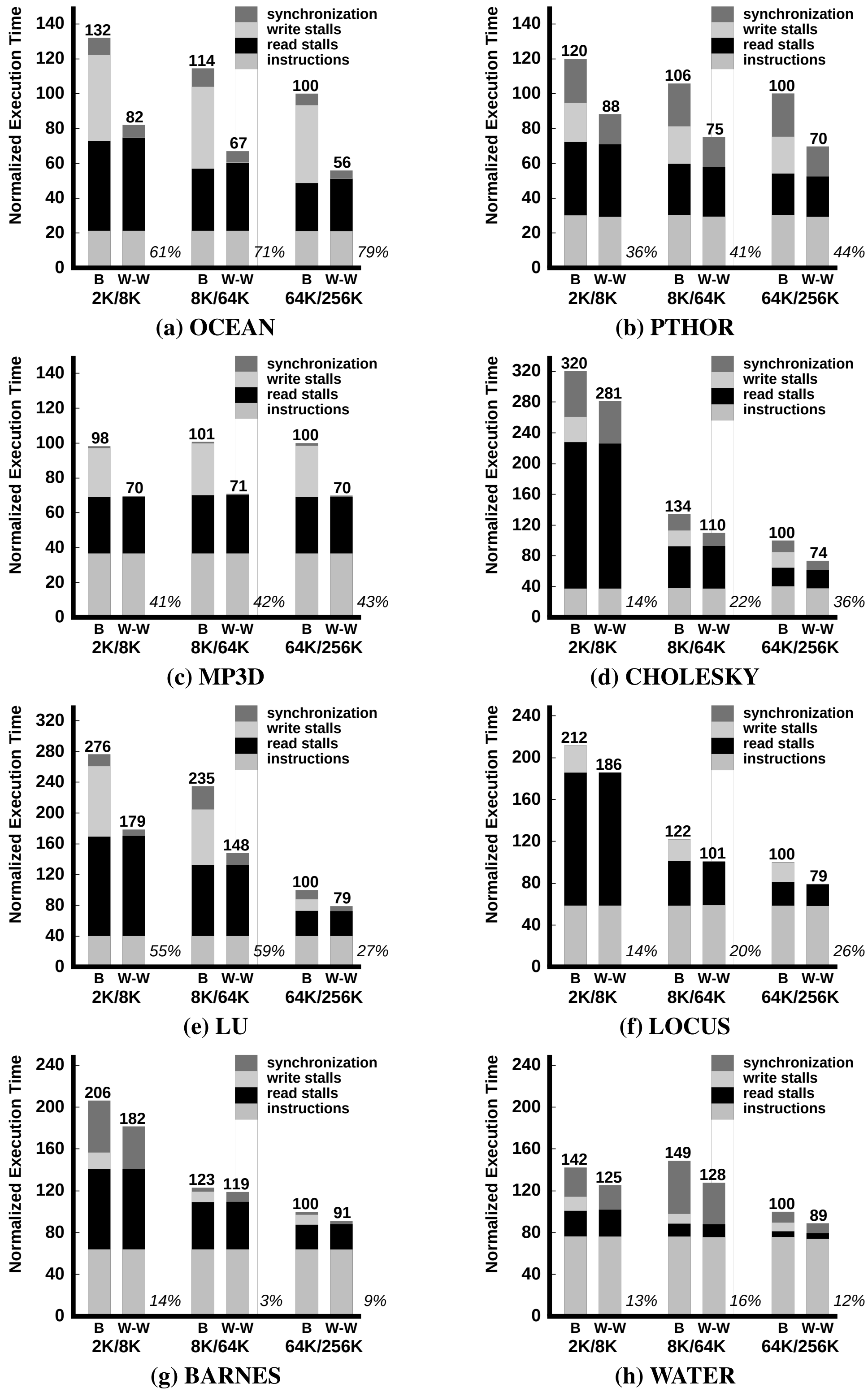


Figure 6.10: Effect of varying the cache sizes.

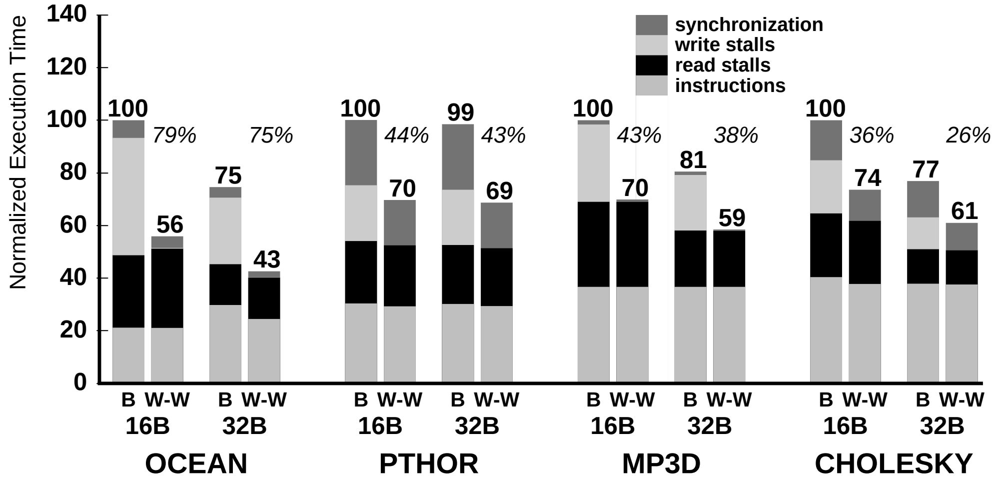


Figure 6.11: Effect of varying the cache line size.

Line Size Variations

Figure 6.11 presents results using two different cache line sizes for the four applications that benefited most from the write-read and write-write reordering optimizations. The leftmost pair of bars for each application are the same as the results in Figure 6.6 with a 16-byte line and 64K/256K caches. The rightmost pair uses a 32-byte line. The results for each application are normalized to the performance of the base implementation with a 16-byte line. The percent number above each pair of bars shows the relative performance gain of W-W over B for the given line size. Three of the applications benefit from the larger line size; the performance of PTHOR is effectively unchanged, however. OCEAN exhibits an anomalous increase in the number of executed instructions when we go to the larger line size; again, this is an artifact of a non-deterministic behavior.

The W-W implementation eliminates the write stall time even at the larger line size. The relative gain from this are slightly less at the larger line size for three of the applications. Nevertheless, the performance gains still remain significant (26% to 75%).

6.2.4 Summary of Blocking Read Results

This section characterized the performance gains from exploiting the program reordering optimizations in the context of architectures with blocking reads. Allowing reads to be overlapped with previous writes was shown to be extremely effective in hiding the latency of write operations; the combination of blocking reads, and the fact that most applications exhibit a larger number of read misses than write misses, allow writes to be buffered without stalling the processors. Overlapping and pipelining of write operations was shown to provide a significant benefit in only a single application (OCEAN) which exhibits a larger number of write misses than read misses; the faster service rate for writes eliminated any stalls caused by a full write buffer.

Overall, the performance gain from the above two optimizations ranged from roughly 10% to 80%, with six out of eight applications gaining over 25%. Lockup-free caches were shown to be essential for achieving the full potential from the above program order relaxations; implementations that do not exploit the write-write reordering optimization also benefit from deeper write buffers and allowing reads to bypass writes in the write buffer.

The blocking read constraint disallows the hardware from exploiting the reordering of reads with respect to later operations, which is enabled by the more aggressive relaxed models (i.e., WO, RCsc, RCpc, Alpha, RMO, PowerPC, and the three PL models). For this reason, relaxed models cannot hide the read latency in architectures with blocking reads. The following section considers the use of other latency hiding techniques to address the read latency in such architectures. We next study the effect of supporting non-blocking reads in Section 6.4.

6.3 Interaction with Other Latency Hiding Techniques

This section studies two other latency hiding techniques in the context of architectures with blocking reads. The two techniques we consider are software-controlled non-binding prefetching and hardware support for multiple contexts. We will show that exploiting the optimizations enabled by relaxed models in combination with either of the above techniques leads to a higher overall performance. The study presented in this section builds on our previous work in this area [GHG+91].

6.3.1 Interaction with Prefetching

This section studies the effect software-controlled non-binding prefetching. We briefly describe the implementation assumptions for prefetching before presenting the performance results.

In our simulation model, a prefetch instruction is similar to a write in that it does not block the processor. The primary cache is checked in the cycle the prefetch instruction is executed. If the line is already in the cache, the prefetch is discarded. Otherwise, the prefetch is sent to a prefetch issue buffer which keeps track of outstanding prefetches. The reason for having a separate prefetch buffer is to avoid delaying prefetch requests unnecessarily behind writes in the write buffer. We model a prefetch issue buffer that is 16 entries deep. The secondary cache is also checked before the prefetch goes to memory. When the prefetch response returns, it is placed in both the secondary and primary caches. Filling the primary cache is assumed to require 4 cycles, during which the processor is not allowed to execute any loads or stores. If the processor references a location for which a prefetch has already been issued, the latter reference is combined with the prefetch request and the reference completes as soon as the prefetch result returns. We assume support for both *read* and *read-exclusive* prefetches. A read prefetch brings data into the cache in a shared mode, while a read-exclusive prefetch acquires an exclusive copy, enabling a later write to that location to complete quickly.

The prefetch instructions for five out of the eight applications (i.e., OCEAN, MP3D, CHOLESKY, LU, LOCUS) were automatically generated using the compiler developed by Mowry [Mow94]. While this compiler generates efficient prefetch instructions for the above applications, it fails to do so for the other three applications in our study. For PTHOR and BARNES, we use a version of the application, also provided by Mowry [Mow94], with hand-inserted prefetch instructions. We did not have access to a version of WATER with hand-inserted prefetching, however.

The results of the prefetching experiments are shown in Figure 6.12. For each application, we show the result for the base implementation (labeled B), for prefetching (labeled PF), and for combination of prefetching with the write-read and write-write program reordering optimizations (labeled RO, for reordering optimizations). The percent number above each RO bar represents the performance gain from exploiting the

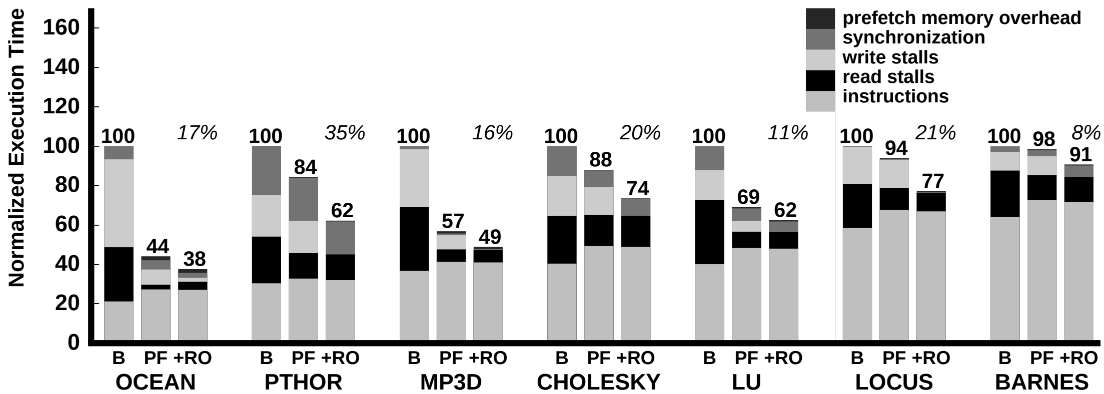


Figure 6.12: Effect of prefetching and relaxing program order.

program reordering optimizations relative to the prefetching case alone (i.e., as we go from PF to RO). The results for the program reordering optimizations without prefetching can be found in Figure 6.6 (compare with the W-W case). The instruction time component of the execution time includes the extra instructions used for prefetching. Furthermore, there is a new component in the execution time breakdown, shown as the *prefetch memory overhead* time. This time corresponds to two situations: (1) when the processor attempts to issue a prefetch but the prefetch issue buffer is full, and (2) when the processor attempts to execute a load or store during the time the cache is busy with a prefetch fill.

The results for prefetching (PF bars) show significant gain in many of the applications. Read and write stall times are reduced due to the use of read and read-exclusive prefetch instructions. The overhead for prefetch instructions clearly shows through in all cases as an increase in the instruction execution time; this negates some of the performance gains from prefetching. We also see that the prefetch memory overhead component is negligible in all applications. At one extreme, the performance improvement over base is 127% in OCEAN. MP3D and LU also have substantial gains from prefetching. However, the gains for BARNES, LOCUS, CHOLESKY, and PTHOR are much more modest, with BARNES gaining a mere 2%.

Referring back to results in Figure 6.6 for the W-W implementation, we see that the write-read and write-write reordering optimizations actually provide a higher performance gain than prefetching for four out of the seven applications (i.e., PTHOR, CHOLESKY, LOCUS, and BARNES). These are the same four applications mentioned above where the gains from prefetching are modest, partly because prefetching fails to significantly reduce the write stall times.

We now consider the results for combining prefetching with the reordering optimizations (labeled as RO). We see that the performance is enhanced for all applications as a result of the combination. Furthermore, comparing the results with those of Figure 6.6 for the W-W implementation, we see that the combination of the techniques performs better or at least as well as (for CHOLESKY and BARNES) the reordering optimizations alone. The main effect of the reordering optimizations is to eliminate virtually any write stall time that prefetching could not eliminate. For four of the applications (PTHOR, CHOLESKY, LOCUS, and BARNES), the incremental gain from exploiting the reordering optimizations beyond prefetching is larger than the gain prefetching provides over the base implementation; again, these are the applications where prefetching is less effective. The percent number shown in italics above each RO bar shows the performance gain from the reordering optimization over the performance of the prefetching case. Combining the two techniques provides an extra 35% performance increase in PTHOR, and an extra 15-20% performance increase for four other applications.

In summary, prefetching and reordering optimizations enabled by relaxed models are complementary techniques; the reordering optimizations eliminate write latency and prefetching reduces the remaining read latency.

6.3.2 Interaction with Multiple Contexts

This section evaluates the use of processors with support for multiple contexts. Each processor has several threads assigned to it, which are kept as hardware contexts. When the context that is currently running encounters a long-latency operation, it is switched out and another context starts executing. In this manner, multiple contexts can help tolerate latency by overlapping the latency of operations in one context with the computation and operations of other concurrent contexts.

The performance gain to be expected from multiple context processors depends on several factors. The first factor is the number of contexts. More contexts increase the chance that a processor will find a ready-to-run context on a switch, but the number of contexts is constrained by available parallelism in the application and the hardware cost to support the contexts. However, a handful of hardware contexts may be sufficient to hide most of the latency if the interval between long-latency operations such as cache misses is large. The second factor is the context switch overhead. A shorter context switch time leads to a more efficient overlap of contexts, but requires more complex hardware support. The third factor relates to application behavior; applications with clustered misses make it more difficult to overlap the latency of one context with the operations in another. Finally, multiple contexts on the same processor share the same cache and can therefore interfere with each other either constructively or destructively.

For our simulations, we use processors with four contexts. We do not consider more contexts per processor because 16 4-context processors require 64 parallel threads and some of our applications do not achieve good speedup with that many threads. In fact, for three of the applications (OCEAN, PTHOR, and CHOLESKY), we show results with only 2 contexts per processor (which translates to 32 threads) since the overall performance was better than the 4-context case. We assume an aggressive context switch overhead of 4 cycles, which corresponds to flushing/loading a short RISC pipeline when switching to the new instruction stream.

Figure 6.13 presents the results with multiple contexts. For each application, we show the result for the base implementation (labeled B), for multiple contexts (labeled MC; the number represents contexts per processor), and for the combination of multiple contexts with the program reordering optimizations (labeled RO). The percent number above each RO bar represents the performance gain from exploiting the program reordering optimizations relative to the multiple contexts case alone (i.e., as we go from MC to RO). The results for the program reordering optimizations without multiple contexts are in Figure 6.6 (the W-W case). Each bar in the graphs is broken down into the following components: time spent executing *instructions* (this includes instructions used to spin on empty task queues; our previous experiments included the time for these instructions under the synchronization component), time spent *switching* between contexts, the time when all contexts are idle waiting for their operations to complete (*all idle*), and the time when the current context is idle but cannot be switched out (*no switch*). Most of the latter idle time is due to the fact that the processor is locked out of the primary cache while fill operations of other contexts complete.

The results show that multiple contexts provide significant gain for most of the applications. Especially for MP3D and LOCUS (and even for BARNES), multiple contexts eliminate almost all overhead components, making the time to execute instructions the main component of the execution time. The two exceptions are PTHOR and CHOLESKY. These two applications do not exhibit sufficient parallelism at even 32 threads, and therefore the threads spend too much time spinning waiting for work which shows through as an increase in the instruction component of the execution time. The reason WATER exhibits small gains is that the application already runs efficiently without multiple contexts. The aggressive context switch time assumption makes the switching component of the execution time negligible in all applications. Similarly, the "no-switch" idle time is negligible in most cases.

Referring back to results in Figure 6.6 for the W-W implementation, we see that the reordering optimizations actually provide a higher performance gain than multiple contexts for four of these applications: OCEAN, PTHOR, CHOLESKY, and WATER. This difference is largest for PTHOR and CHOLESKY, where multiple contexts fail to hide much of the latency.

We now consider the combination of multiple contexts with the reordering optimizations (labeled as RO). With the reordering optimizations, write misses are no longer considered long latency operations from the processor's perspective since writes are simply put into the write buffer. Therefore, the processor no longer switches on write misses. The effect of this is to increase the distance between long latency operations, thus requiring fewer contexts to eliminate most of the remaining read miss latency. The results in Figure 6.13 show the performance is enhanced for all applications as a result of this combination. The percent number shown in italics above each RO bar shows the performance gain from the reordering optimization over the performance of the multiple contexts case. Combining the two techniques provides an extra 37% performance increase in PTHOR, and a higher than 25% increase for three other applications. Finally, comparing the results with those of Figure 6.6 for the W-W implementation, we see that the combination of the techniques also performs better than the reordering optimizations alone.

In summary, multiple contexts and reordering optimizations enabled by relaxed models are also complementary techniques; the latter optimizations eliminate writes as long latency operations, allowing the remaining read latency to be hidden with fewer contexts.

6.3.3 Summary of Other Latency Hiding Techniques

Since the techniques of relaxing program orders, prefetching, and using multiple contexts are alternative ways for hiding latency, it may seem that the use of one technique eliminates the need for the others. However, the results in this section show that there is synergy in exploiting program reordering optimizations in combination with either prefetching or multiple contexts. The reason for this is that prefetching and multiple contexts are sensitive to application behavior and fail to universally reduce or eliminate the write latency for every application. At the same time, allowing writes to overlap with following read and write operations fully eliminates the write latency in virtually any application. Supporting these techniques together provides synergy at the implementation level as well; this was discussed in Section 5.8 of the previous chapter.

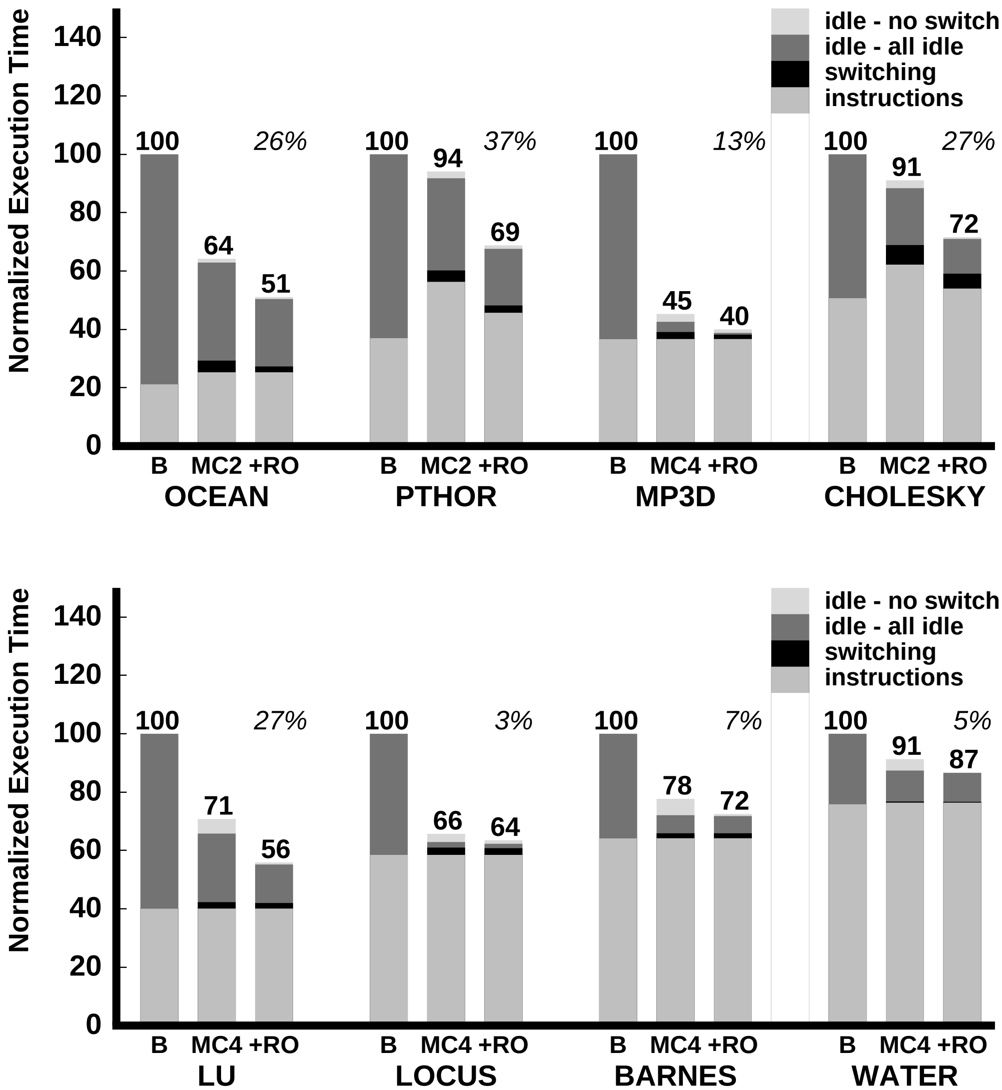


Figure 6.13: Effect of multiple contexts and relaxing program order.

6.4 Architectures with Non-Blocking Reads

The previous sections showed that the latency of write operations can be hidden by buffering writes and allowing reads to bypass previous pending writes. Hiding the latency of reads by exploiting the overlap allowed by relaxed models is inherently more difficult, however, simply because the computation following the read typically depends on the return value. To hide this latency, the processor needs to find other independent computation and memory accesses to process while awaiting the return value for the read. Achieving this through hardware implies an aggressive processor architecture with a decoupled decode and execution unit and capability for dynamic scheduling, branch prediction, and speculative execution. Given expected memory latencies in the range of tens to hundreds of processor cycles, there are serious concerns as to whether such a processor architecture allows for effective overlap of read latencies.

This section explores the use of dynamically scheduled processors to aggressively exploit the overlap of a read with operations that follow it in program order. Our results show that a substantial fraction of the read latency can be hidden by exploiting this technique. We describe the framework for our simulation experiments in Section 6.4.1. The performance results for overlapping reads with future operations is presented in Section 6.4.2.

The results presented in this section are directly derived from a previous study that has already been published [GGH92].

6.4.1 Experimental Framework

This section presents the different components used in our simulation environment. The first part discusses architectural details of the processor. The second part explains the coupling of the multiprocessor simulation with the processor simulator. Finally, we present the benchmark applications used in our study.

Simulated Processor Architecture

The simulated processor environment is based on an architecture proposed by Johnson [Joh91]. Only a brief description is provided here; the interested reader is referred to [Joh91] for more detail.

Figure 6.14 shows the overall structure of the processor. The processor consists of a number of functional units with associated *reservation stations* [Tom67]. Although not shown in the figure, there are also four floating point units for performing floating point add, multiply, divide, and conversion.[2] The reservation stations are instruction buffers that decouple instruction decoding from instruction execution and allow for dynamic scheduling of instructions. Decoded instructions and operands are placed into the appropriate reservation station by the decoder. The reservation station can issue an instruction as soon as the instruction has its data dependences satisfied and the functional unit is free.

The reorder buffer [SP85] shown in Figure 6.14 is responsible for several functions in this architecture. The first function is to eliminate storage conflicts through register renaming [Kel75]. Each instruction that is decoded is dynamically allocated a location in the reorder buffer and a tag is associated with its result register. The tag is updated to the actual result value once the instruction completes. Correct execution is achieved by providing the value or tag in the reorder buffer (instead of the value in the register file) to later instructions

[2] While Johnson assumes the floating point units reside on a coprocessor, we assume they are on the same chip as the integer units.

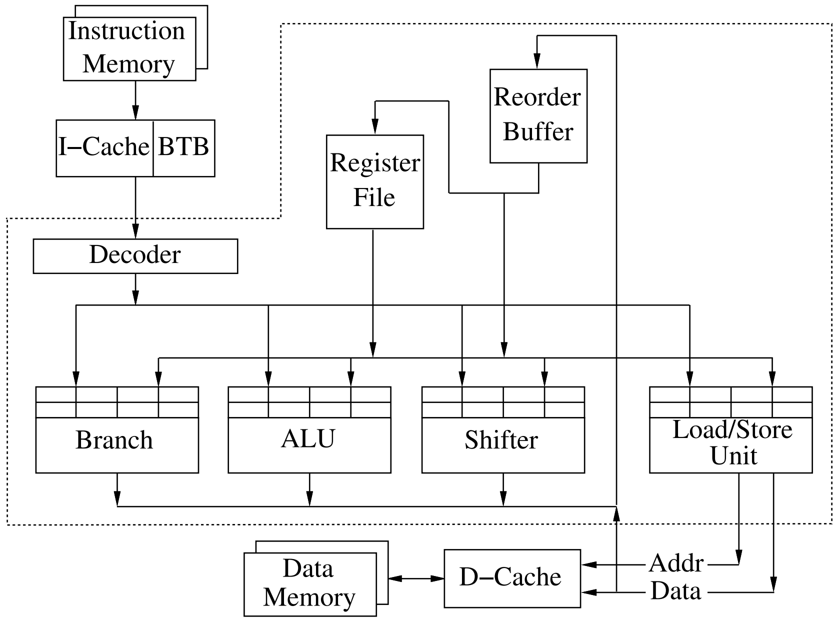


Figure 6.14: Overall structure of Johnson's dynamically scheduled processor.

that attempt to read this register. Unresolved operand tags in the reservation stations are updated when the corresponding instruction completes.

The second function for the reorder buffer is to provide the rollback mechanism necessary for supporting speculative execution past unresolved branches. The architecture uses a branch target buffer (BTB) [LS84] to provide dynamic branch prediction. The rollback mechanism for mispredicted branches works as follows. Decoded instructions enter the reorder buffer in program order. When an instruction at the head of the buffer completes, the entry belonging to it is deallocated and the result value is written to the register file. Since instructions enter the reorder buffer in program order and are retired in the same order, updates to the register file take place in program order. Therefore, instructions that depend on an unresolved branch are not committed to the register file until the branch completes. Similarly, memory stores are also held back (the reorder buffer controls the retiring of stores from the store buffer).[3] If a branch is mispredicted, all instructions past the branch are invalidated from the reorder buffer, the reservation stations, and the appropriate buffers, and decoding and execution is started from the correct branch target. The mechanism provided for handling mispredicted branches is also used to provide precise interrupts. This allows the processor to restart quickly without the need to save and restore complex state information.

For our study, we have assumed a single cycle latency for all functional units except the load/store unit. The latency for loads and stores is assumed to be one cycle in case of a cache hit. The scheduling of instructions within each functional unit is allowed to be out-of-order. This pertains to the load and store unit also as long as the memory consistency constraints allow it. Furthermore, load operations can bypass the store buffer and dependence checking is done on the store buffer to assure a correct return value for the load.

Regarding the caches, we simulate a lockup-free data cache [Kro81] that allows for multiple outstanding requests, including multiple reads. We ran our simulations assuming a single cache port, thus limiting the

[3] A store at the head of the reorder buffer is retired as soon as its address translation completes and the consistency constraints allow its issue. The store buffer is free to issue the store after this point.

number of loads and stores executed to at most one per cycle. The branch target buffer size was set to 2048 entries with a 4-way set-associative organization. For simplicity, all instructions are assumed to hit in the instruction cache.

The next section describes how our processor simulator is coupled to the multiprocessor simulation.

Multiprocessor Simulation

Our multiprocessor simulation is done using Tango Lite, the same reference generator used to do our studies for blocking reads. The simulation assumes a simple multiprocessor architecture. The architecture consists of 16 processors, each with a 64 Kbyte lockup-free data cache that is kept coherent using an invalidation-based scheme. The caches are direct mapped, write-back caches with a line size of 16 bytes. The simulated processors are simple in-order issue processors with blocking reads. Writes are placed in a write buffer and are pipelined, and reads are allowed to bypass the write buffer. The latency of memory is assumed to be 1 cycle for cache hits and a fixed number of cycles for cache misses. In our experiments, we use a 50 cycle penalty for cache misses (results for a 100 cycle penalty are provided in a technical report [GGH93a]). Queueing and contention effects in the interconnection network are not modeled.

The above simulation generates a dynamic instruction trace for each of the simulated processes. The generated trace is augmented with other dynamic information including the effective address for load and store instructions and the effective latency for each memory and synchronization operation.

To simulate the effects of dynamic scheduling, we choose the dynamic instruction trace for one of the processes from the multiprocessor simulation and feed it through our processor simulator (described in the previous section). Since the pertinent information about the multiprocessor cache simulation and the synchronization behavior is already contained in the trace, the processor simulator simply uses this information to determine the latency of memory accesses and synchronization.

The use of trace-driven simulation in our study may introduce some inaccuracies in our results [KEL91]. This is because the trace is generated assuming a different processor architecture than the processor we eventually simulate. Thus, although the exact global interleaving of shared accesses may be different given the two processor architectures, we use the same ordering for both. The extent to which this affects results depends both on the application behavior and on the characteristics being studied. For our study, the program characteristics that determine how well dynamically scheduled processors hide memory latency are the cache miss behavior, the data dependence characteristics between memory accesses and computation, and the predictability of branches. We do not expect these characteristics to vary greatly due to slightly different global interleaving of the accesses.

One characteristic that can vary greatly due to different interleavings is the synchronization latency and load balance in an execution; this issue was also witnessed in the results presented for blocking reads. Thus, synchronization times need to be treated cautiously in trace-driven simulations. In this study, we mainly focus on the fraction of read latency that can be hidden and the overhead of synchronization does not fundamentally affect the ability of dynamic scheduling techniques in hiding this latency. Synchronization times are only reported to provide an estimate for the fraction of execution time that corresponds to memory latency. Overall, we believe that our results are only minimally affected by the use of traces in our simulations.

Table 6.5: Statistics on branch behavior.

Program	*Percentage of Instructions*	*Avg. Distance bet. Branches (in instructions)*	*Percentage Correctly Predicted*	*Avg. Distance bet. Mispredictions (in instructions)*
OCEAN	6.0%	16.6	97.9%	778.9
PTHOR	15.3%	6.5	81.2%	34.7
MP3D	6.1%	16.4	90.8%	176.9
LU	8.0%	12.5	98.0%	618.1
LOCUS	15.6%	6.4	92.1%	81.6

Benchmark Applications

We present results for five out of the eight applications that were studied in the previous section: OCEAN, PTHOR, MP3D, LU, and LOCUS. These are the applications that provide a higher potential for gain from reducing the read latencies. The descriptions for these applications can be found in Section 6.2.1. There are a few differences compared to the blocking read study. For two of the above applications, we use inputs different from those used in the blocking read study. For MP3D, we ran 10,000 particles in a 64x8x8 space array for 5 time steps. For LOCUS, we used the `Primary1.grin` input, which contains 1266 wires and a 481-by-18 cost array. Furthermore, because we are simulating the detailed data dependences among instructions, both shared and non-shared data are simulated, and all libraries (e.g., math libraries) are annotated to generate the appropriate instruction and data references.

Because branch behavior is an important factor in determining the performance of dynamically scheduled processors, Table 6.5 provides some statistics on branches for the five applications. This data is from the Tango Lite multiprocessor simulation assuming 16 processors and a memory latency of 50 cycles.

6.4.2 Experimental Results

This section studies the effect of relaxing program orders in conjunction with dynamically scheduled processors. We present simulation results comparing statically and dynamically scheduled processors. The degree to which dynamic scheduling succeeds in hiding the latency of reads is determined by the size of the lookahead buffer from which the processor chooses instructions to execute, the predictability of branches, and finally the data dependence characteristics of the program, which determine the degree of independence among instructions. Our experimental results will shed more light on the effect of each of these factors on performance. The results assume a 16 processor system with single instruction issue processors and a uniform 50 cycle cache miss latency. We briefly consider the effect of higher latency and using multiple instruction issue capability in the latter part of this section.

While the results presented in this section show similar trends to the results in the blocking read study (e.g., for the base implementation with no reordering optimization or for the statically scheduled processor), *these two sets of results are not directly comparable*. There are several reasons for this. The most important reason is that we are simulating a much simpler memory system architecture in this section; e.g., there is only a single level 64 Kbyte cache, there is a single latency of 50 cycles for all cache misses, and contention at the memory system or network is not modeled. Another reason is that we are simulating both shared and non-shared data, in addition to all library references; furthermore, two of the applications use a slightly different input set.

Figure 6.15 shows the simulation results comparing statically and dynamically scheduled processors. The left-most bar in each graph shows the breakdown of the execution time for the base implementation (labeled B) which completes each operation before initiating the next one (i.e., no overlap in execution of instructions and memory operations). The order in which we show the components of execution time is slightly different from the previous sections. The bottom section still represents the time to execute the instructions. The section above it now represents the time that the processor is stalled waiting for synchronization (e.g., locks, barriers). The two sections on top show the read and write stall times. The penalty for cache misses is assumed to be 50 cycles in these simulations.

The rest of the bars in each graph correspond to different implementations of a relaxed memory model such as RCpc that enables the reordering of read and write operations with respect to one another. The first bar corresponds to a simple statically scheduled processor with blocking reads (SSBR); this implementation cannot exploit the reordering of reads with following operations. The second bar is a statically scheduled processor with non-blocking reads (SS). This processor is allowed to execute past read misses and the stall is delayed up to the first use of the return value. We assume a 16 word deep write buffer for the above processors. The SS processor also has a 16 word deep read buffer. Finally, we evaluate the dynamically scheduled processor (DS) described in Section 6.4.1. To allow us to better isolate the effects on hiding memory latency, we have limited the decode and issue rate for the DS processor to a maximum of 1 instruction per cycle. Another important assumption in this study is that the processor cycle time is the same for SSBR, SS, and DS (i.e., dynamic scheduling does not increase the cycle time). The size of the reorder buffer (or lookahead window) used is denoted at the bottom of each column. This size corresponds to the maximum number of instructions that can reside in the window at any given time. The window size is varied from 16 to 256 instructions. All simulations assume an aggressive memory system with lockup-free caches, write buffers that allow reads to bypass writes, and the ability to issue one access per cycle from each node. Of course, the extent to which these features are exploited depends the processor architecture.

Some general observations that can be made from the simulation results are: (i) the statically scheduled processor with blocking reads (SSBR) successfully hides all of the write latency, but fails to hide any read latency, (ii) the statically scheduled processor with nonblocking reads also fails to hide much of the read latency, and (iii) dynamic scheduling can substantially reduce the read latency given processors with large window sizes. The first section below briefly describes the results for static scheduling. The next two sections analyze the results for dynamic scheduling in greater detail.

Static Scheduling Results

We briefly discuss the results for the two statically scheduled processors (SSBR and SS) below. We first concentrate on the results for the SSBR processor. The effect of relaxing the write-read and write-write program order on statically scheduled processors was studied in the previous section. The results here show the same trend: these two optimizations hide the write latency completely in all applications.

We now consider the results for the SS processor. The difference between SS and SSBR is that SS does not block on read accesses and delays this stall until the first use of the return value. Therefore, there is the potential for hiding the latency of reads in the region between the read access and its first use. However,

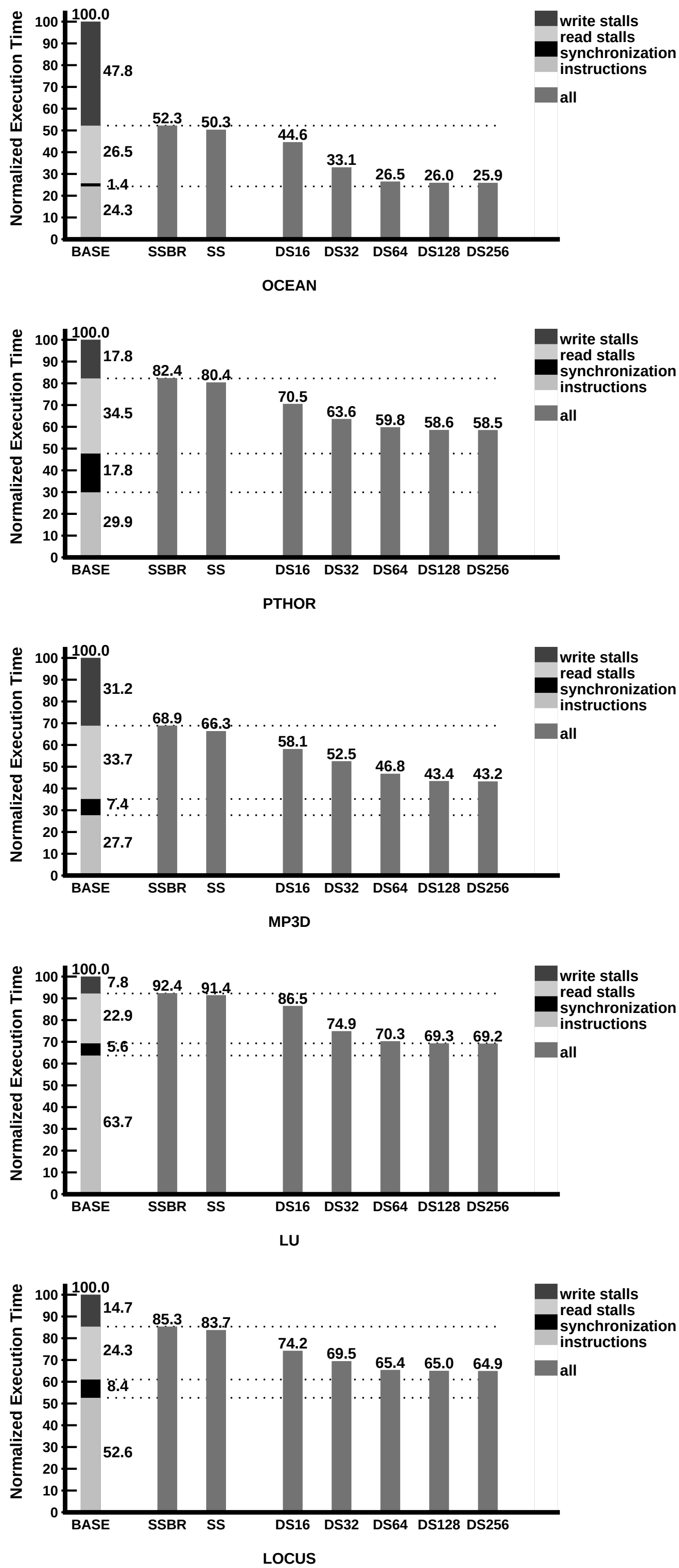


Figure 6.15: Results for dynamically scheduled processors (memory latency of 50 cycles).

as the results indicate, the improvement over SSBR is minimal. This is mainly because the object code has not been rescheduled to exploit non-blocking reads (by moving the load access further away from the use of the return value). Thus, the processor is effectively stalled a short distance away from a read miss since the use of the return value is normally within a few instructions away. It is still interesting to study the effect of compiler rescheduling to exploit non-blocking reads as an alternative to using dynamic scheduling techniques in hardware.

Dynamic Scheduling Results

We begin by building some intuition for the different factors that affect the performance of dynamically scheduled processors, specifically in their ability to hide read latencies. We then analyze the results based on this intuition. The next section presents more simulation results to further isolate the factors that influence performance.

Three factors determine the effectiveness of dynamic scheduling in hiding memory latency. Two of these factors relate to the characteristics of the application: (i) data dependence behavior and (ii) control (or branch) behavior. The third factor is the size of the lookahead window (or reorder buffer) provided by the architecture.

The data dependence characteristics of an application is the most fundamental factor influencing performance. Dynamic scheduling depends on the presence of independent accesses and computation for hiding memory latency. For the applications we are considering, independent operations are present at different granularities. Typically, the applications exploit parallelism at the task or loop iteration level and each processor is assigned a set of tasks or loop iterations. One source of independent operations is within the task or loop iteration boundary. Another source of independence is operations from two different tasks or loop iterations that have been assigned to the same processor. The availability of such independent operations and the distance between them determines the feasibility of a hardware technique in finding and overlapping them.

The behavior of branches in an application is also a key factor that affects performance. The frequency of branches and their predictability determine whether it is feasible for hardware to find distant independent operations.

Finally, the size of the lookahead window determines the maximum distance between independent operations that can be overlapped by the hardware. To overlap two independent accesses, the size of the lookahead window needs to be at least as large as the distance between the two accesses. Furthermore, to fully overlap the latency of memory accesses with computation requires enough independent instructions to keep the processor busy for the duration of the latency. This requires the window size to be at least as large as the latency of memory. For example, with the latency of 50 cycles, we require a window size of at least 50 instructions. Smaller window sizes overlap only a fraction of this latency (proportional to the size).

We now consider the bound on the gain achievable by dynamic scheduling given that read and write accesses can be overlapped between synchronization points. Since we limit issue to a maximum of one instruction per cycle, the time due to the computation (busy/useful time) cannot be diminished. In addition, acquire synchronization overhead arising from load imbalance or contention for synchronization variables is also impossible to hide with the techniques we are considering. On the other hand, the fraction of the synchronization overhead arising from the memory latency to access synchronization variables can be hidden in the same way that normal memory latency is hidden. For example, the latency to access a free lock can be hidden by overlapping this time with the computation prior to it. Relating this to the applications, our detailed simulation results show that virtually all of the acquire overhead in MP3D, LU, LOCUS, and OCEAN arises from load imbalance and contention. Therefore, the best dynamic scheduling can do for these applications is to fully hide the read and write (including release) latency. In PTHOR, however, approximately 30% of the acquire overhead is due to latency for accessing free locks. Therefore, for PTHOR, it is theoretically possible to hide a fraction of the acquire overhead as well.

Referring to the results for DS in Figure 6.15, we notice that there is a gradual increase in performance as we move from window size of 16 to window size of 256, with the performance increasing more rapidly for the small window sizes and leveling off at the larger windows. For LU and OCEAN, dynamic scheduling hides virtually all memory latency at window size of 64. For LOCUS, almost all latency, except for 16% of the read latency, is hidden at the larger window sizes. For MP3D, 24% of the read latency remains even at the larger window sizes. Similarly, PTHOR has 31% of its read latency remaining.

Data dependence and small window size effects are most likely the two factors that influence performance at the small window sizes (16 and 32). Based on the average distance between mispredicted branches shown in Table 6.5, branch behavior is not expected to be a determining factor for the small windows, except possibly in PTHOR. The factors shift at the larger window sizes. In LU and OCEAN, performance levels off at larger windows simply because the memory latency is virtually all hidden at the smaller window sizes. On the other hand, the leveling off of performance in MP3D, PTHOR, and LOCUS is most likely due to the fact that the branch prediction falls short of effectively using such large window sizes.

Detailed Analysis of Dynamic Scheduling

This section provides simulation results to further isolate the effects of different factors influencing the performance of dynamically scheduled processors. Figure 6.16 presents these results for all five applications. The left-most column in each graph repeats the data presented in Figure 6.15 for the BASE processor. The next five columns in each graph provide performance results for the different window sizes assuming perfect branch prediction. The last five columns in each graph present the results given for the case where branch prediction is perfect and data dependences are being ignored.[4]

To isolate the effect of branches on performance, we can compare the performance for each window size with and without perfect branch prediction (left side of Figure 6.16 and right side of Figure 6.15). For LU and OCEAN, the branch prediction is already so good that we see virtually no gain from perfect branch prediction even at the largest window size of 256. For LOCUS, perfect branch prediction performs slightly better at the large window sizes of 128 and 256. For MP3D, noticeable gain from perfect branch prediction starts at window size of 64. PTHOR, which has the worst branch behavior of all the applications, seems to noticeably gain from perfect branch prediction even at the small window sizes of 16 and 32. Thus, except for PTHOR, the branch behavior of the applications does not seem to affect the performance until we reach the 128 and 256 window sizes.

To isolate the effect of data dependences, we can look at the improvement in performance at each window size when we move from perfect branch prediction to additionally ignoring data dependences (left and right side in Figure 6.16). For LU and OCEAN, there is little or no gain from ignoring data dependences, pointing

[4]Dependences arising from consistency constraints, for example from an acquire synchronization to the following access, are still respected.

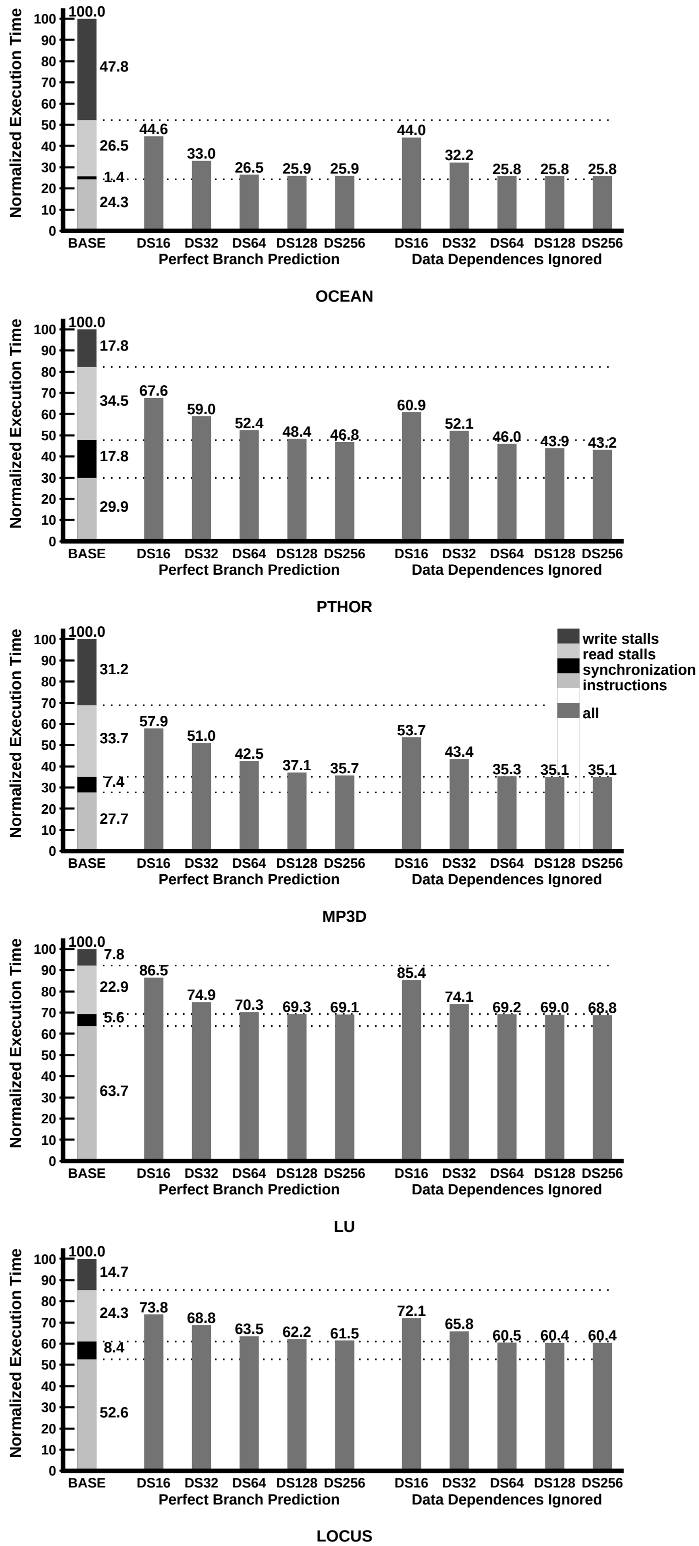


Figure 6.16: Effect of perfect branch prediction and ignoring data dependences for dynamic scheduling results.

to the fact that data dependences do not hinder performance in these two applications (this is as expected since it was possible to hide all memory latency for these two applications). For MP3D, PTHOR, and LOCUS, we observe that ignoring data dependences increases the performance more at the small window sizes. In fact, at window size of 256, the performance with and without ignoring data dependences is virtually the same in these three applications. This points to the fact that there are some data dependences at short distances (within a task or loop iteration), however, by looking ahead at substantially larger distances, it is possible to find more independent operations (across tasks or loop iterations). Although this is an interesting observation, from a practical point of view, it is not possible to exploit such independence at large distances with hardware techniques alone simply because the branch prediction falls short of providing such lookahead distances.

Our detailed simulation results also confirm the presence of data dependences at short distances. One such result measures the delay of each read miss from the time the instruction is decoded and placed in the reorder buffer to the time the read is issued to memory. Below, we consider this measure for the window size of 64 given perfect branch prediction. In LU and OCEAN, we find that read misses are rarely delayed more than 10 cycles. This clearly points to the fact that read misses are independent from one another in these two applications. Results for MP3D show that about 15% of the read misses are delayed over 40 cycles. Similarly, in LOCUS, more than 20% of read misses are delayed over 40 cycles. This indicates the presence of read misses that are data dependent on one another, with one read miss affecting the address of the next read miss. Data dependences have the most serious effect in PTHOR. For PTHOR, around 50% of the read misses are delayed over 50 cycles. This is indicative of dependence chains formed by multiple misses. One effect of such dependence chains is that there are fewer independent accesses to overlap. Furthermore, it is more difficult to fully hide the latency of read misses in such a chain by overlap with independent computation, simply because the misses in the chain behave like a single read miss with double or triple the effective memory latency. This is the main reason why it takes window sizes of up to 256 to fully hide the latency of read misses in MP3D and PTHOR even with perfect branch prediction.

Finally, we can isolate the effect of window size by further analyzing the results for when both branch prediction is perfect and data dependences are ignored (right hand side in Figure 6.16). In this case, the only remaining factor that can affect performance is the window size. Looking at the results, we notice that window sizes of 16 and 32 do not fully hide the latency of reads. As explained in the previous section, the reasons for this are (i) small window sizes do not find independent operations that are farther apart than the window size, and (ii) to fully overlap latency with computation, the window size (in instructions) needs to be at least as large as the latency of access (in cycles). Relating to (i), our detailed simulation data for LU show that 90% of the read misses are a distance of 20-30 instructions apart. Thus, the window size of 16 performs relatively poorly compared to other applications. Similarly, in OCEAN, about 55% of the read misses are 16 to 20 instructions apart. Again, the reason for the poor performance at window size of 16 is that the distance between independent accesses exceeds the window size. Regarding reason (ii), we observe from the results that once we reach a window size of 64 (which exceeds the memory latency), the latency can be fully overlapped with computation given the assumed lack of data dependences.[5]

In summary, data dependence and branch predictability do not hinder performance in LU and OCEAN.

[5] Given perfect branch prediction and the assumed lack of dependence, the execution times for MP3D, LU, LOCUS, and OCEAN asymptotically approach the time for the computation plus the acquire overhead. For PTHOR, however, some of the acquire overhead is also being hidden. As explained in the previous section, the reason for this is that about 30% of the acquire overhead in PTHOR arises from latency for accessing free locks.

The performance of LOCUS and MP3D are primarily affected by data dependence at small window sizes and by branch predictability at large window sizes. Finally, PTHOR is affected by both data dependence and branch behavior at all window sizes.

Effect of Higher Latency and Multiple Instruction Issue

We briefly discuss the effect of higher memory latency and multiple instruction issue in this section. The simulation results for these experiments are presented in an earlier technical report [GGH93a].

We repeated the simulations assuming a higher memory latency of 100 cycles. The trends were similar to what we observed for the 50 cycle memory latency. The most obvious difference was that the performance levels off at the window size of 128 instead of 64. This arises from the fact that the window size needs to exceed the latency in order to fully overlap memory latency with computation. Another observation was that the relative gain in performance from hiding memory latency is consistently larger for the higher latency of 100 cycles.

We also did a preliminary study of the effect of multiple instruction issue. As with the higher latency results, the performance still increases when we moved from window size of 64 to 128, while without multiple issue, the performance virtually levels off at window size of 64 (see Figure 6.15). This arises from the fact that multiple issue speeds up the computation, while memory latency remains at 50 cycles. Thus, a larger window size is needed to fully overlap the memory latency with computation time.

6.4.3 Discussion of Non-Blocking Read Results

The goal of this section has been to evaluate the effect of overlapping reads with later operations. We used a dynamically scheduled processor to exploit this type of overlap.

The choice of the application domain clearly influences our results. The scientific and engineering parallel programs used in this study typically exhibit sufficient instruction-level parallelism that can be exploited to hide memory latency. Furthermore, the branch predictability in these applications is usually better than for most uniprocessor programs.

Our architectural assumptions also influence the results. Our results are somewhat optimistic since we assume a high bandwidth memory system. In addition, we do not model the effect of contention in the network or at memory modules. However, the results are pessimistic in that we did not consider the effect of compiler help to schedule code to allow the processor to use the lookahead window more effectively. In addition, the FIFO retirement of instructions from the lookahead window (to provide precise interrupts) is a conservative way of using the window since instructions that have already been executed may remain in the window, disallowing new instruction from being considered for overlapped execution. Other techniques for providing precise interrupts may allow for better use of the window space. More aggressive branch prediction strategies may also allow higher performance for the applications with poor branch prediction.

The most important concern about dynamically scheduled processors is the extra hardware complexity and its effect on cycle time. These factors have to be considered in determining the actual gain from dynamic scheduling. In addition, other techniques for hiding latency need to be considered. For example, the overlap of memory accesses allowed by relaxed models can also be exploited by the compiler for scheduling read misses to mask their latency on a statically scheduled processor with non-blocking reads or to decrease window size requirements in a dynamically scheduled processor.

6.4.4 Summary of Non-Blocking Read Results

This section explored the overlap of reads with future read and write operations for hiding read latency using dynamically scheduled processors. Assuming a memory latency of 50 cycles, the average percentage of read latency that was hidden across the five applications was 33% for window size of 16, 63% for window size of 32, and 81% for window size of 64. For two of the applications, LU and OCEAN, read latency was fully hidden at window size of 64. In general, larger window sizes did not increase the performance substantially. The trends for a memory latency of 100 cycles are similar, except that larger windows were needed to fully hide the read latency. In isolating the different factors that affect performance, we identified data dependences at short distances and small lookahead as the limiting factors for small windows and branch prediction as the limiting factor for large windows.

These results show that a substantial fraction of the memory latency can be hidden by overlapping operations as allowed by relaxed models. Most next generation commercial processors will be able to benefit from these gains since they use dynamic scheduling techniques. However, it is not yet clear whether large window sizes can be efficiently implemented in such processors. Another interesting area for research is to evaluate compiler techniques that exploit relaxed models to schedule reads early. Such compiler rescheduling may allow dynamic processors with smaller windows or statically scheduled processors with non-blocking reads to effectively hide read latency with simpler hardware.

6.5 Related Work

This section describes related work on evaluating the performance of relaxed memory models. The latter part of the section briefly mentions some remaining areas for investigation.

6.5.1 Related Work on Blocking Reads

Our original study in this area [GGH91a] provided the first comprehensive set of results comparing the performance of various relaxed memory models on real applications. The work presented in this thesis extends our previous results to a larger set of applications and a wider set of architectural assumptions.

A later study by Zucker and Baer [ZB92, Zuc92] provides performance results for a "dance-hall" architecture with uniform memory latency across four applications. The overall trends observed by this study are qualitatively similar to those in our original study. However, some major architectural differences make a more detailed comparison difficult. Specifically, the low cache miss latency of 18-20 processor cycles assumed severely limits the performance gains from relaxed models; real multiprocessor systems exhibit significantly higher latencies.

Torrellas and Hennessy [TH90] present a detailed analytical model of a multiprocessor architecture and estimate the effects of relaxing the consistency model on performance. A maximum gain of 20% in performance is predicted for using weak consistency over sequential consistency. This prediction is lower than the results in our study mainly due to the low memory latencies and limited bus bandwidth that were assumed.

There have also been a number of studies evaluating the performance gain from exploiting relaxed models in the context of software-supported shared memory systems [CBZ91, KCZ92, DKCZ93]. The sources of performance gain for relaxed models are quite different in these systems compared to a hardware-supported cache coherent system studied in this chapter.

6.5.2 Related Work on Interaction with Other Techniques

Our earlier paper on the interaction of relaxed memory models with prefetching and multiple contexts was the first to provide a performance study of combining these techniques [GHG+91]. Mowry [Mow94] and Laudon [Lau94] also present results for the interaction with prefetching and multiple contexts, respectively, across a larger set of applications. The trends observed in all these studies are quite similar.

Another relevant technique is a cache coherence protocol optimization that can reduce write latency by detecting and specially treating migratory data. This idea was concurrently proposed by Cox and Fowler [CF93] and Stenstrom et al. [SBS93]. The protocol detects a migratory pattern if it sees a read/write miss sequence by one processor followed by a similar sequence by another processor. Once a line is classified as migratory, read requests are provided with ownership for the line, effectively prefetching the line for the next write; the effect is quite similar to issuing a read-exclusive prefetch for certain reads. This technique can substantially reduce the number of write misses experienced by a processor especially for applications that exhibit a high percentage of migratory data. Therefore, it can be used to decrease the write stall time in a sequentially consistent system. Relaxed models provide a more general and robust way of eliminating write stall time since they work for both migratory and non-migratory data. Nevertheless, the migratory optimization may be beneficial in systems with low bandwidth interconnects since it eliminates messages associated with some write misses as opposed to only hiding the latency of the miss. This optimization can also be used in conjunction with relaxed models to reduce the total number of write misses; performance is enhanced only if the application's bandwidth requirements are high relative to the system's network.

6.5.3 Related Work on Non-Blocking Reads

There have been several studies on the effect of non-blocking reads in both uniprocessor and multiprocessor systems. We mention a few of the studies that are relevant to our work.

Melvin and Patt [MP91] have observed that dynamically scheduled processors are less sensitive to higher memory latency than statically scheduled processors. Their study was done in the context of multiple issue of instructions in uniprocessors and the largest cache miss penalty used was 10 cycles. In addition, the behavior (e.g., cache miss, data dependence, and branch behavior) of the uniprocessor applications they studied is substantially different from the parallel applications used in our study.

Farkas and Jouppi [FJ94] study the effect of non-blocking reads and lockup-free caches in the context of statically scheduled uniprocessors. They use an aggressive compiler to schedule loads early for up to a 20 cycle load penalty. Their results show substantial improvements across many of the SPEC92 benchmarks.

In the context of multiprocessors, Scheurich and Dubois [DSB86, SD88] provided simple analytical models to estimate the benefits arising from relaxed consistency models and lockup-free caches. The gains predicted by these models are large, sometimes close to an order of magnitude gain in performance. These results are over optimistic partly due to the fact that effects such as data dependence, branch prediction, or look-ahead capability are not carefully considered.

Finally, our study of non-blocking reads in conjunction with dynamically scheduled processors [GGH92] provided evidence that these techniques can also work well in the context of higher latencies that are typical of multiprocessor systems. The results presented in this chapter are completely derived from this earlier work.

6.5.4 Areas for Further Investigation

The study presented in this chapter focuses on the performance of parallel scientific and engineering applications on cache-coherent shared memory systems. Below, we briefly describe some areas where the performance of relaxed models remain to be fully explored.

The choice of the application domain clearly influences any performance study. Other areas that are interesting to explore include commercial and data base applications and the effect on operating system code, especially in the context of non-blocking read architectures.[6]

From a processor perspective, we have not fully explored the effect of multiple instruction issue per cycle. Because multiple issue processors decrease the time required to execute instructions, it is clear that they would increase the importance of reducing and hiding memory latency. Therefore, we expect the gains from relaxed models to increase. However, understanding the detailed interaction between instruction parallelism and latency hiding demands a comprehensive study. Furthermore, it is interesting to study the detailed architectural choices in building dynamically scheduled processors from the perspective of exploiting the overlap of reads.

From a cache coherence protocol perspective, we focused on an invalidation-based directory protocol similar to the one used by DASH [LLG+90]. The trade-offs for other protocols may be different. For example, update-based protocols are interesting to study because they trade off an increase in write misses and write latency for a decrease in read misses. Since relaxed models are extremely effective in hiding write latency, and since read latency is harder to hide in general, there is a potentially nice match between the two techniques. Nevertheless, since simple update protocols fail to perform well in large scale systems, more research is required to develop robust hybrid protocols that selectively use updates. Studying the effect of relaxed models in the context of other protocol designs, such as the IEEE Scalable Coherent Interface (SCI) protocol [JLGS90], is also interesting. Typical implementations of the SCI protocol can exhibit large write latencies since cache copies are invalidated in a serialized fashion, making the latency proportional to the number of sharers. Therefore, the gains from hiding the write latency can be more substantial.

Chapter 5 described a couple of techniques (i.e., speculative reads and hardware prefetching for writes described in Section 5.4) for boosting the performance of sequentially consistent systems in hardware. These techniques enable many of the reordering optimizations that are typically allowed by more relaxed models to be safely used while maintaining the illusion of the stricter model. The degree to which these techniques boost the performance of models such as sequential consistency remains to be fully studied. While the success of these techniques may de-emphasize the correctness aspect of overlapping memory accesses in multiprocessors at the hardware level, the underlying architectural mechanisms (e.g., lockup-free caches, non-blocking reads, and other techniques studied in this chapter) that enable and exploit such overlap still remain important.

Chapter 5 also explored a large number of implementation alternatives. In this chapter, we mainly concentrated on the important design choices. While other design choices are likely to have a more minor

[6] For blocking read architectures, it is predictable that relaxed models will eliminate the write latency as in scientific applications.

effect on performance, it is still interesting to consider their performance effects in more detail.

Finally, the effect of relaxed models on compiler optimizations (and vice versa) remain to be fully studied. As we have discussed before, strict models such as sequential consistency disallow many of the compiler optimizations that are applied to uniprocessor code. Therefore, the gains from relaxed models on the compiler side may potentially exceed the gains on the hardware side. Similarly, as we mentioned in Section 6.4, compiler algorithms for scheduling non-blocking reads are likely to play an important role in exploiting the overlap of reads in the context of both statically and dynamically scheduled processors.

6.6 Summary

This chapter characterized the performance gains from exploiting the program reordering optimizations enabled by relaxed memory consistency models. We showed that these optimizations lead to significant gains in the context of architectures with blocking reads by fully hiding the write latency. The gains ranged from 10% to 80%, with six out of eight applications gaining over 25%. Furthermore, we showed relaxed models are complementary to other latency hiding techniques such as prefetching and supporting multiple contexts. Combining the techniques provided gains of higher than 15% over using prefetching alone for five applications, and gains higher than 25% over using multiple context alone for four of the applications. Finally, we showed that architectures that support non-blocking reads can exploit relaxed models to also hide a substantial fraction of the read latency, leading to a larger overall performance benefit from relaxed models.

Chapter 7

Conclusions

The memory consistency model for a shared-memory multiprocessor specifies how memory behaves with respect to read and write operations from multiple processors. As such, the memory consistency model influences many aspects of system design, including the design of programming languages, compilers, and the underlying architecture and hardware. The choice of the model can have significant impact on *performance*, *programming ease*, and software *portability* for a given system. Models that impose fewer constraints offer the potential for higher performance by providing flexibility to hardware and software to overlap and reorder memory operations. At the same time, fewer ordering guarantees can compromise programmability and portability.

Coming up with a new memory consistency model is quite trivial. After all, it is just a matter of picking and choosing the set of orders that must be maintained among shared memory operations. The real design challenge lies in providing a *balanced* solution with respect to the fundamental trade-offs discussed above. Choosing a relaxed model, or designing a new relaxed model, requires considering questions such as:

- What is the target environment, including the types of programmers, applications, and architectures or systems that are targeted?
- Are the semantics of the model defined precisely?
- How difficult is it to reason and program with the model? How restrictive is the model with respect to different programming styles? Is it possible to provide a simple representation to programmers? How easy is it to port programs to and from the model?
- What are the practical implementation optimizations that motivate the model? How difficult is it to efficiently implement the model at both the architecture and compiler level?
- What are the performance gains from using this model relative to alternative models? Does the evaluation include both the compiler and the hardware? And most importantly, do the performance gains justify the additional programming and implementation complexity?

Addressing the above issues is non-trivial since it involves considering complex trade-offs and interactions, in addition to dealing with subtle correctness issues (which are easy to underestimate). Unfortunately, failure to properly address such issues has led to an abundance of alternative proposals and a lack of clear methodologies for choosing among them. This situation is exacerbated by the prevalence of non-uniform terminologies and informal specifications, making it extremely difficult to compare the different alternatives.

7.1 Thesis Summary

The primary goal of this thesis has been to shed light on the complex trade-offs that arise in choosing an appropriate memory consistency model. The contributions lie in four areas:

- abstractions that enhance programmability with relaxed models,
- precise specification of the ordering constraints imposed by a model,
- efficient and practical techniques for implementing a memory model, and
- detailed quantitative analysis of the performance benefits of various models.

The abstraction presented by properly-labeled programs captures the types of optimizations exploited by previously proposed relaxed models within a simple and easy-to-use programming style. To enable safe optimizations, the programmer is required to supply high-level information about the behavior of memory operations, such as whether an operation is involved in a race with other operations to the same address. This approach alleviates the need for reasoning with low-level reordering optimizations that are exposed by a typical relaxed model. Another important benefit of the programmer-supplied information is automatic and efficient portability across a wide range of implementations.

To enable correct and efficient implementations, the thesis presents a formal framework for specifying the ordering constraints imposed by a model. Our specification methodology inherently exposes aggressive optimizations by imposing as few constraints as possible. Specifying various models within this uniform framework also enables easy comparison of the semantics and optimizations supported by the different models. Finally, the specifications are useful for determining correct and efficient mechanisms for porting programs across different models.

The implementation techniques and performance results primarily focus on multiprocessor systems with hardware-support for cache coherence. On the implementation side, our work represents the most comprehensive set of techniques for efficiently supporting various memory models. In addition to presenting fundamental techniques, the thesis exposes subtle implementation issues that are often not covered in the literature. Our work on evaluating the performance of different models also represents the most comprehensive set of quantitative results currently available. Together, the above two contributions provide a solid understanding of the trade-off between design complexity and performance.

7.2 Future Directions

This section presents a brief overview of promising areas for future research. Many of the ideas related to extending this thesis and pursuing new directions have already been discussed in each of the previous four chapters.

The least explored areas with respect to relaxed memory models are programming languages, programming environments, and compilers. The abstraction presented to the programmer and the information provided by the programmer are tightly linked to the programming language; one important area is with respect to easier expression of information by programmers. The programming environment, including tools that help detect races and incorrect programmer-supplied information, is also critical in making relaxed models simpler to use. Finally, there is much research to be done in the compiler area including implementations that correctly and efficiently support a given model and performance evaluations to determine the gains from relaxed models on the compiler side.

Other important areas to study include verification tools for checking the correctness of specifications and implementations, better understanding and characterization of programs with asynchronous data operations, and finally exploring other ways to exploit the flexibility provided by relaxed memory models. As an example of the latter point, exploiting relaxed models for improving the performance of software distributed shared memory systems has been an active area of research.

An exciting area to pursue is more efficient implementation techniques for supporting strict models such as sequential consistency. Our work on the hardware side has led to the surprising observation that many of the optimizations that motivate relaxed memory models are indeed possible to exploit under sequential consistency (see Section 5.4 in Chapter 5 and our previous paper [GGH91b]). More recently, Krishnamurthy and Yelick [KY94, KY95] have presented preliminary results on applying safe compiler optimizations to explicitly parallel programs under sequential consistency, based on an aggressive analysis that attempts to determine a minimal set of program orders that are required for correctness. Further developments, especially on the compiler side, can significantly change the balance in the choice of memory models for systems that can exploit the above optimizations.

It is also instructive to observe the trends in industry. Companies such as Digital, IBM, and Sun have already opted for supporting relaxed memory models in their commercial multiprocessors and have successfully developed fully functional operating systems, compilers, and other software that exploit these models. Fortunately, all three companies have attempted to provide formal specifications of their models; this is in contrast to the ambiguous descriptions provided for older architectures such as the IBM-370 and VAX. Other companies such as Hewlett-Packard (HP), Intel, and Silicon Graphics (SGI) have chosen to support sequential consistency in their current architectures. The next generation processors from these companies exploit aggressive implementation techniques in conjunction with dynamically scheduled processors to efficiently support sequential consistency (mentioned in the previous paragraph). However, there is still a dependence on relaxed models for enabling compiler optimizations for explicitly parallel programs. Furthermore, companies such as Intel still maintain the option to move to relaxed models by recommending that programmers use special serializing and locking operations for future compatibility.

Overall, there is still a lot of room for interesting research and solid contributions in the area of memory consistency models for shared-memory multiprocessors. Our hope is that the framework and intuition presented here pave the way for future developments in this area.

Appendix A

Alternative Definition for Ordering Chain

This appendix presents an alternative definition for the ordering chain defined by Definition 3.1 in Chapter 3. The definition given below captures all ordering chains defined by Definition 3.1. The subtle difference is that the new definition also includes chains that start with a conflict order or end with a conflict order even if all operations in the chain are not to the same location.

Definition A.1: Ordering Chain
Given two conflicting operations u and v in an execution, an *ordering chain* exists from operation u to operation v if and only if
(a) $u \xrightarrow{po} v$, or
(b) $u \xrightarrow{po} w_1 \xrightarrow{co} r_1 \xrightarrow{po} w_2 \xrightarrow{co} r_2 \xrightarrow{po} w_3 \ldots \xrightarrow{co} r_n \xrightarrow{po} v$, where $n \geq 1$, w_i is a write access, r_j is a read access, and w_i and r_j are to the same location if $i = j$. u may be the same as w_1, and v may be the same as r_n, as long as there is at least one $\xrightarrow{po}$ arc in the chain.

Assume we substitute the above definition (instead of Definition 3.1) in Definition 3.2 for competing operations under the PL1 model. Intuitively, fewer operations will be considered competing since the chains captured by Definition A.1 are a superset of those in Definition 3.1. Consider the program segment in Figure A.1, for example. With the original definition of ordering chain, all memory instructions in this example must be labeled as competing to yield a PL1 program. However, with the new definition, the read of A on P3 can be labeled as non-competing. The latter interpretation of competing is closer to the true notion of a race; the read of A on P3 is always ordered after the write of A on P1 in every SC execution and is therefore not involved in a race. Nevertheless, we expect that the resulting labels based on the two different definitions (i.e., Definition 3.1 and Definition A.1) will be identical for most realistic programs.

The main reason we chose Definition 3.1 over Definition A.1 is that Definition A.1 is not appropriate for the PL3 model defined in Chapter 3 (e.g., it would disallow the optimization of using non-atomic write operations to support loop writes), and we wanted to maintain the notion of ordering chains consistent across the three PL models. The alternative is to use the new definition for ordering chains by appropriately substituting it

P1		P2		P3	
a1:	**A = 1;**	*a2:*	**while (A == 0);**		
		b2:	**B = 1;**	*a3:*	**while (B == 0);**
				b3:	**u = A;**

Figure A.1: Canonical 3 processor example.

in Definition 3.2 for PL1 and also in Definition 3.6 for PL2. One caveat in doing this is that the notion of competing will be different for the new PL1 and PL2 models compared to the PL3 model; some operations that may be considered non-competing under PL1 and PL2 will be considered as competing under PL3 (e.g., read of A on P3 in the example discussed above). Therefore, transforming PL1 and PL2 programs to PL3 programs can become complicated.

We refer back to Figure A.1 to illustrate the difference between Definition 3.1 and Definition A.1 in PL3. Every SC execution must have the result (u=1) for this example. Assume Definition 3.1 as the definition for an ordering chain. The following labeling of the program yields a PL3 program: (a1) as a non-loop write, (a2) and (a3) as loop reads, (b2) as a loop write, and (b3) as a non-loop read. Note that (b3) clearly does not qualify as a loop read, and since (a1) competes with (b3), (a1) does not qualify as a loop write. This is important because we allow loop writes to be non-atomic for PL3 and if (a1) was labeled as a loop write, this optimization could lead to a non-SC execution with the result (u=0). Now consider substituting Definition A.1 for Definition 3.1 as the definition of an ordering chain. The write (a1) would no longer be considered as competing with the read (b3). Therefore, the read (b3) could be labeled as non-competing and the write (a1) would now qualify as a loop write. This example shows that allowing loop writes to be non-atomic would no longer be a safe optimization under PL3 if we used Definition A.1 instead of Definition 3.1.

Appendix C describes a modification to the PL3 model based on Definition A.1 for ordering chains; the modified version inevitably forgoes some of the optimizations exploited by the original version.

Appendix B

General Definition for Synchronization Loop Constructs

This appendix provides the more general definition for a synchronization loop construct defined by Definition 3.10 in Chapter 3. The conditions below are identical to the general conditions provided in the original paper on PLpc [GAG+92], with a few corrections from a later technical report [AGG+93]. The intuitive idea behind the conditions is to ensure that the number of times the loop executes or the values returned by the unsuccessful reads cannot be practically detected. This allows us to conceptually replace the loop construct with only the successful operation(s) that cause the loop to terminate.

Definition B.1: Loop Construct
A *loop construct* is a sequence of instructions in a program that would be repeatedly executed until a specific read in the sequence (the *exit read*) reads a specific location (the *exit location*) and returns one of certain values (the *exit read values*). If the exit read is part of a read-modify-write, then the write of the read-modify-write is called the *exit write* and the value it writes is called the *exit write value*.

Definition B.2: Synchronization Loop Construct

A loop construct in a program is a *synchronization loop construct* if and only if it always terminates in every SC execution of the program and the following conditions hold. Consider a modification of the program so that it executes beginning at the loop construct. Add another process to the program that randomly changes the data memory. Consider every SC execution with every possible initial state of the data memory and processor registers. At the beginning of every such SC execution, the exit read, exit location, and exit read values should only be a function of the initial state of memory and registers and of the program text. The exit write value can additionally be a function of the value that the exit read returns. Then, for every such SC execution,
(i) except for the final exit write, loop instructions should not change the value of any shared memory location,
(ii) the values of registers or private memory changed by any loop instruction cannot be accessed by any instruction not in the loop construct,
(iii) a loop instruction cannot modify the exit read, exit location, exit read values, or the exit write values corresponding to a particular exit read value,
(iv) the loop terminates only when the exit read returns one of the exit read values from the exit location and the exit write stores the exit write value corresponding to the exit read value returned,
(v) if exit read values persist in the exit location, then the loop eventually exits,
(vi) the first instruction instance program ordered after the loop is the same in every other SC execution that begins with the same initial state of data memory and processor registers, and
(vii) the only shared-memory operations in the iteration that terminates the loop should be the exit read and exit write (if the exit read is part of a read-modify-write).

When analyzing an SC execution, the accesses of a synchronization loop construct can be replaced by the final successful exit read and exit write (if any). The unsuccessful accesses can be labeled non-competing.

The above definition is fairly general, which partly explains its complexity. For example, it allows implementations of locks using a test&test&set [RS84] or back-off [MCS91] techniques to be considered as synchronization loop constructs.

Appendix C

Subtleties in the PL3 Model

This appendix presents a subtle example that illustrates the need for some of the complexity in the definitions of loop read (Definition 3.11) and loop write (Definition 3.12) in the PL3 model. In addition, we describe a modification to the PL3 model that removes some of this complexity at the cost of forgoing some optimizations.

C.1 Illustrative Example for Loop Read and Loop Write

Figure C.1 shows an example that provides some intuition for the presence of complicated conditions such as condition (d) in Definition 3.11 for loop reads and condition (b) in Definition 3.12 for loop writes. All four while loops qualify as synchronization loop constructs; therefore, all unsuccessful reads from each loop are assumed to be discarded in any execution. In every SC execution, the successful read of (b3) executes after the write (d1). In addition, the only sequentially consistent result is (u,v,w,x)=(1,1,0,1). Memory instructions shown in bold generate a competing operation in at least one SC execution. To yield a PL3 program, all instructions shown in bold must be considered as sync operations to ensure sufficient synchronization ordering chains among the non-competing operations. The operations generated by the (c1), (b2), and (a3) loops trivially satisfy the conditions for loop read. Similarly, the write to B on P2 trivially satisfies the conditions for loop write. In contrast, the successful read operation generated by the (b3) loop satisfies all conditions for a loop read except for condition (d) in Definition 3.11 (in this case, W' in condition (d) corresponds to the hypothetical write that initializes location A). Therefore, this instruction must be labeled as non-loop. As a result, neither of the two writes to A on P1 qualify as loop writes; (d1) does not qualify as a loop write because it competes with (b3) which is a non-loop read, and (b1) does not qualify because of condition (b) in Definition 3.12.

One of the optimizations we exploit in the PL3 model is to allow loop writes to be non-atomic with respect to multiple copies. This optimization is safe with the labeling described above. The write that especially matters is (b1), since if this write behaves non-atomically, then it is possible for the loop (b3) on P3 to observe the initial value of A and terminate before either (b1) or (d1) complete with respect to this processor, which

P1		P2		P3	
a1:	C = 1;	*a2:*	D = 1;		
b1:	**A = 1;**	*b2:*	**while (A == 0);**		
c1:	**while (B == 0);**	*c2:*	**B = 1;**	*a3:*	**while (B == 0);**
d1:	**A = 0;**	*d2:*	v = C;	*b3:*	**while (A == 1);**
e1:	u = D;			*c3:*	w = A;
				d3:	x = D;

Figure C.1: Program segment illustrating subtleties of loop read and loop write.

could result in the non-SC result of (u,v,w,x)=(1,1,1,1).

Condition (d) of Definition 3.11 and condition (b) of Definition 3.12 play an important role in disallowing the write (b1) from being labeled as a loop write. Consider removing condition (d) of Definition 3.11; the read in (b3) would qualify as a loop read, which would in turn allow the writes in (b1) and (d1) to be labeled as loop writes. Similarly, consider removing condition (b) of Definition 3.12; even though the write (d1) would not qualify as a loop write, the write in (b1) would still qualify as a loop write.

The above example is admittedly contrived and the likelihood of observing such behavior in real programs is low. Nevertheless, the conditions for PL3 must be applicable across all possible programs.

C.2 Simplification of the PL3 Model

It is possible to simplify the PL3 model by forgoing the optimization that allows loop writes to be non-atomic. Therefore, the only major optimization that would be exploited relative to PL2 is the relaxation of program order between a competing write followed by a competing read if either is identified with the loop label.

Forgoing the optimization that allows loop writes to be non-atomic simplifies a number of the condition for the PL3 model. First, condition (d) of Definition 3.11 and condition (b) of Definition 3.12 could be eliminated, which would simplify the conditions for loop read and loop write. Second, we could now use Definition A.1 instead of Definition 3.1 for the definition of an ordering chain. Of course, forgoing this optimization requires the sufficient conditions for PL3 to be strengthened. In particular, the following multiprocessor dependence chains should be added to the conditions presented in Figure 4.16 in Chapter 4: Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ {Wc $\xrightarrow{sco}$ Rc $\xrightarrow{spo'}$ }* {Wc $\xrightarrow{sco}$ Rc ($\xrightarrow{spo}$ | $\xrightarrow{spo'}$)} RW. Similarly, the conditions for porting PL3 programs to PC and RCpc (Tables 4.3 and 4.4) must be changed to require every Rc to be part of a RMW (with W mapped to non-sync or release for RCpc), and the conditions for porting to PC+ and RCpc+ (Table 4.6) should be changed to require every Wc to be mapped to an atomic write.

Appendix D

Detecting Incorrect Labels and Violations of Sequential Consistency

This appendix describes the two detection techniques, mentioned in Section 3.8.4, for helping programmers use programmer-centric models.

D.1 Detecting Incorrect Labels

By providing memory operation labels for a program, the programmer assumes that executions of the program will obey sequential consistency. However, this assumption may be violated if any of the labels are erroneous. Therefore, tools that help the programmer identify potentially incorrect labels can play an important role in further simplifying the use of programmer-centric models.

One of the fundamental pieces of information conveyed through labels in the PL framework is whether a memory operation is involved in a race (i.e., competes with another operation). Identifying races in parallel programs is in itself an important problem; even with sequentially consistency, programmers opt to avoid data races to simplify the task of reasoning about programs. In fact, the presence of races often signals a programming error. Therefore, much research has been done on detecting data races. Fortunately, many of the proposed techniques can be easily adapted for identifying erroneous labels for competing operations in the context of programmer-centric models.

The proposed techniques for detecting data races for programs written for sequential consistency can be classified as *static* or *dynamic*. Static techniques use compile-time analysis to detect potential data races that could occur in any possible execution of a program [BK89, Tay83b]. In contrast, dynamic techniques analyze individual executions of the program and determine whether a particular execution exhibits a data race [AP87, DS90a, HKMC90, NM89, NM91]. The following discusses the trade-offs between these two techniques in more detail.

The advantage of static techniques is that all data races that may potentially occur in *any* possible SC execution of a program are detected. Therefore, the programmer is assured that the program will not exhibit any data races if no data races are reported by the detection technique. However, the problem of exactly determining whether a program is free of data races or not is known to be undecidable [Ber66]. Exact analysis has been shown to be NP-complete even for restricted classes of programs and synchronization primitives [NM90, Tay83a]. Therefore, practical algorithms for static detection are often extremely conservative. This limits the usefulness of static techniques since many programs that actually exhibit no data races may conservatively be reported as having data races.

Dynamic techniques have the advantage of providing *exact* information on whether a specific execution of the program exhibits data races. These techniques often use a trace gathering mechanism to monitor the order among memory accesses in an execution and analyze this information to determine whether the execution exhibits any data races. The tracing and analysis can be either done *on-the-fly* (e.g., [DS90a]) or in a *post-mortem* fashion (e.g., [NM89]). The on-the-fly techniques buffer trace information in memory and detect data races as they occur. In contrast, post-mortem techniques generate trace files containing information on the order of accesses and the information is analyzed after the execution completes.

Like static techniques, dynamic techniques have their own set of limitations. *First*, the gathering and analysis of memory traces can adversely affect the execution time for the program. A two to five times increase in execution time has been reported for on-the-fly techniques [DS90a]. Even though more efficient on-the-fly techniques have been proposed by limiting the class of programs [MC91], the overhead is still expected to be unacceptable for normal executions of the program. Thus, dynamic techniques are limited for use during debugging only. The *second* problem with the dynamic techniques arises from the fact that exact information is only provided for a single execution of the program, with no information provided about other possible executions. This is further complicated by the fact that the extra overhead of tracing can affect the critical timing in a program, resulting in possibly different executions than would have normally occurred. Therefore, there is no guarantee that a program will not exhibit any data races even if all executions during debugging are determined to be free of data races.

The dynamic techniques described above are designed for use with architectures that are sequentially consistent. Fortunately, most systems that support relaxed models also provide mechanisms for enforcing sequential consistency (e.g., by simply disallowing some of the reordering optimizations). Therefore, it is possible to employ dynamic race detection techniques by configuring such systems to be sequentially consistent during program debugging mode. More recently, an extension has been proposed to allow the use of dynamic race detection even when the system is configured with a relaxed model during debugging (certain system restrictions apply) [AHMN91]. This approach extends dynamic detection to systems that do not support sequential consistency in any form. Furthermore, there is a potential performance advantage during debugging since the system can operate under a relaxed model. However, this performance advantage is expected to be small in most systems since the execution time during debugging will be dominated by the overhead of tracing and analysis which effectively washes out the performance difference between sequential consistency versus a more relaxed model.

We now briefly describe how the dynamic race detection techniques can be applied to detecting incorrect labels in programs written for the PL models. Since dynamic techniques capture a trace of the execution, it is conceptually possible to add additional steps in the analysis phase to verify the validity of labels by checking for various conditions among the memory operations in the trace. For PL1, the only information provided by the programmer is whether an operation is competing or not. Since the notion of race and competing are closely related, the techniques discussed above can be easily adapted to identify competing operations in an execution and to check that none of the competing operations are incorrectly identified as non-competing by the labels. By adding a little more analysis to the race detection techniques, it is possible to extend the applicability of this technique to the PL2 model by detecting whether enough operations are identified as sync and non-sync. The PL3 model provides a more challenging problem, however, since (i) the conditions for distinguishing loop and non-loop operations are quite complex, and (ii) some operations (i.e., all but the final memory operation in each synchronization loop) have to be ignored during the analysis phase. Although it may conceptually be possible to extend the race detection techniques to detect incorrect labels with the PL3 model, there needs to be further study to determine the practicality of the approach; for example, the more complex analysis that is required compared to simple race detection may potentially increase the execution time overhead above the acceptable threshold for even debugging.

In summary, dynamic detection of races provides a practical method for helping the programmer identify incorrect labels during the development and debugging phase of a program. However, race detection techniques do not provide a complete solution since races that do not exhibit themselves during debugging can unexpectedly occur during normal executions without being detected. In systems that support a relaxed model, the occurrence of such races can lead to violations of sequential consistency. Therefore, to ensure sequentially consistent executions, the programmer is still ultimately responsible for guaranteeing the correctness of labels.

D.2 Detecting Violations of Sequential Consistency

Given the difficulty in verifying the correctness of the labels, an alternative approach for helping the programmer use programmer-centric models is to directly detect whether sequential consistency is upheld by executions of a program on a given system. After all, the ultimate reason for checking the correctness of labels is to ensure that all executions of the program will be sequentially consistent. We proposed and studied the viability of this approach in an earlier paper [GG91]; the following covers the important highlights from this work.

The main result of our work is a new implementation technique, applicable to cache coherent systems, that dynamically detects possible violations of sequential consistency [GG91]. Unfortunately, exactly determining whether or not an execution is sequentially consistent turns out to be a difficult problem. Therefore, we settle for a conservative detection for violations of sequential consistency. Below, we describe the constraints we impose on the accuracy of the detection. For the detection to be *conservative*, we have to guarantee that a violation is detected if the execution is not sequentially consistent. More formally, if execution E is not sequentially consistent, the detection mechanism will detect a violation in E. However, this condition can be trivially satisfied by *always* detecting a violation for every execution. Such a detection scheme is clearly not useful. Therefore, we need to place a bound on the conservatism of the detection. We know that PL programs are guaranteed to produce sequentially consistent executions on systems that support PL models. Therefore, a useful detection scheme is one that is conservative, but does not conservatively detect a violation when the program is properly labeled. More formally, the *bound on conservative detection* is as follows: If program P is a PL program, then the detection mechanism is guaranteed not to detect a violation in *any* execution of P. Therefore, if no violation is detected for an execution, then the execution is sequentially consistent (due to conservative detection). And if a violation is detected, then the program is known to have incorrect labels (due to bound on conservative detection).

The detection technique discussed above can provide useful feedback to programmers on architectures with relaxed consistency models. In contrast to the work on data race detection, our technique does not exactly determine whether or not the program or execution have data races (or incorrect labels). Instead, the technique detects whether sequential consistency may be violated in an execution of the program. An advantage of this approach is that the programmer is provided with exact information either about the execution or about the program. For every execution, the proposed implementation conclusively determines *either* that the execution is sequentially consistent *or* that the labels provided by the programmer are incorrect. In the first case, the programmer is assured that the correctness of the execution was not affected by the fact that the architecture supports a relaxed model. In the second case, the programmer knows that the program has incorrect labels and can result in sequentially inconsistent results on architectures supporting relaxed models. Furthermore, the implementation we proposed provides the functionality for detection with minor additional hardware and with virtually no affect on the performance of the system [GG91]. Therefore, the detection technique is efficient enough to be used during *all* executions of the program (as opposed to monitoring executions only during debugging).

The intuition behind the detection mechanism is simple. Assume u and v are two accesses in program order. Sequential consistency is guaranteed if the completion of v is delayed until u completes (Condition 2.2 in Chapter 2). Multiprocessors that support a relaxed model may allow u and v to be reordered to achieve higher performance. The purpose of the detection mechanism is to determine whether such reordering may result in a sequentially inconsistent execution. A violation may arise if v completes before u. First consider the case where v is a read access. Assume the read completes (its return value is bound) before u completes. This does not necessarily lead to a violation of sequential consistency, however. If at the time u completes, the return value for v is the same as the current value of the location accessed by v, then any computation based on the access is correct since even if v was delayed until u completed, the value the access would return would be the same. However, if the current value is different from the value returned by access v, then sequential consistency may have been violated and the detection scheme conservatively detects a violation. Now consider the case where v is a write access. Assume the write completes before u completes. Even so, sequential consistency is guaranteed if no other processor attempts to access the location touched by v until u completes since this is *as if* v was delayed until u completed. However, the detection scheme conservatively detects a violation in case there is an attempt to access the location touched by v while u is pending.

Cache coherent architectures provide an efficient base for the above detection mechanism. Each processor can maintain the state of its outstanding memory operations in a per-processor *detection buffer* and can monitor coherence transactions destined to its cache to determine whether a violation of SC is likely. Let us refer back to accesses u and v. If v is a read access, an invalidation or update for the location accessed by v before u has completed indicates that the value bound by access v is old and may result in a violation of SC. In addition, the lack of invalidation or update messages indicates that the value bound by v is current. Similarly, if v is a write access, a read, invalidation (or ownership request), or update message to the location accessed by v before u has completed indicates a possible violation of SC. The original paper discusses an example implementation based on the above intuition and describes some restrictions on the system to ensure the desired bound on conservative detection [GG91]. The implementation techniques described are general, and are compatible with either an invalidation-based or update-based coherence mechanism and with cache line sizes of greater than one word (unlike some proposals for race detection [MC91]).

The detection mechanism discussed above is applicable to the PL1 and PL2 models. As we mentioned above, a few extra restrictions are imposed on the system to ensure the detection is not overly conservative. However, these restrictions are either already satisfied or can be easily incorporated in most systems with virtually no effect on performance [GG91]. As with the race detection techniques, extending the approach to PL3 seems challenging because (i) some operations (i.e., all but the final memory operation in each synchronization loop) have to be ignored by the detection mechanism and thus need to be identifiable, and (ii) systems that support PL3 typically allow optimizations such as allowing some competing writes to be non-atomic which further complicates detection. Therefore, whether the approach can be extended to PL3 remains to be explored.

D.3 Summary of Detection Techniques

Compared to a sequentially consistent system, the extra effort required to program a system that supports a programmer-centric model arises from the need to provide correct labels for all memory operations. We discussed two techniques that can help the programmer in ensuring correct labels. The first technique involved an extension to the debugging environment that helps the programmer identify incorrect labels while a program is being developed. While this technique is useful during the debugging phase, it fails to protect the programmer from incorrect labels during normal program runs. The second technique we discussed remedies this problem by employing a simple hardware mechanism that can efficiently monitor normal executions of the program and notify the programmer if sequential consistency may be compromised due to the presence of incorrect labels.

Work in this area is still in its infancy and there are many opportunities for improvement. Specifically considering the two techniques discussed above, both inherently depend on the original program order among memory references in determining whether an execution exhibits incorrect labels or sequentially inconsistent behavior. Therefore, they are not readily applicable to programs already optimized by a compiler that exploits the reordering of memory operations enabled by relaxed models. Furthermore, extensions of these techniques to deal with more complex programmer-centric models such as PL3 remains to be explored.

Appendix E

Alternative Definition for Return Value of Reads

The abstraction for memory operations presented in Chapter 4 uses the R_{init}(i) and W_{init}(i) sub-operations to capture the effect of optimizations such as read forwarding. These sub-operations play an important role in the return value condition for reads (Condition 4.6) and the initiation condition (Condition 4.4), which together allow a processor's read to occur before its own write to the same location and yet ensure that the read returns the value of that write. We chose to introduce the R_{init}(i) and W_{init}(i) sub-operations because they lead to a more flexible and intuitive abstraction for modeling optimizations such as read forwarding.

As an alternative, it is possible to eliminate the R_{init}(i) and W_{init}(i) sub-operations and the initiation condition altogether if we define the return value condition (currently defined by Condition 4.6) as follows:

Condition E.1: Alternative Condition for Return Value for Read Sub-Operations
A read sub-operation R(i) by P_i returns a value that satisfies the following conditions. If there is a write operation W by P_i to the same location as R(i) such that W $\xrightarrow{po}$ R and R(i) $\xrightarrow{xo}$ W(i), then R(i) returns the value of the last such W in $\xrightarrow{xo}$. Otherwise, R(i) returns the value of W'(i) (from any processor) such that W'(i) is the last write sub-operation to the same location that is ordered before R(i) by $\xrightarrow{xo}$. If there are no writes that satisfy either of the above two categories, then R(i) returns the initial value of the location.

The original specification of TSO and PSO [SFC91] use a similar value condition to model the effect of optimizations such as read forwarding. The more informal descriptions in Chapter 2 (Section 2.4) also use a similar value condition that does not require the R_{init}(i) and W_{init}(i) sub-operations. The slightly non-intuitive aspect of Condition E.1 is that a read can return the value of a write even though *all* sub-operations associated with the write occur after the read.

Appendix F

Reach Relation

This appendix defines the reach relation ($\xrightarrow{rch}$) discussed in Section 4.2 for the properly labeled models. The reach relation presented below is a straightforward adaptation of the $\xrightarrow{rch}$ relation developed for the PLpc model [AGG+93].

Defining the reach relation requires considering the set of dynamic instruction instances, I, in an execution. The following presents an abstraction for the instruction instances. We classify instructions into three types: *computation*, *memory*, and *control*. Computation instructions read a set of registers (*register read set*) and map the read values into new values that are written to another (possibly same) set of registers (*register write set*). Memory instructions are used to read and write memory locations (both private and shared). A memory read instruction reads the address to be read from a register, reads the specified memory location, and writes the return value into a register. A memory write instruction reads the address and value to be written from registers and writes the value into the specified memory location. A memory read-modify-write instruction is both a memory read and a memory write instruction that reads and writes the same location. For a read instruction, the register read set comprises of the address register and the register write set is the destination register. For a write instruction, the register read set comprises of the address and value registers and there is no register write set. Finally, control instructions (e.g., branch instructions) change the control flow by appropriately adjusting the program counter. The register read set for a control instruction is a set of registers whose values determine the change in the control flow. If an instance of a control instruction results in the program counter only being incremented, then we say the instruction instance preserves program control flow; if the program counter changes in any other way, we say the instruction instance results in changing the program control flow. Note that the above description of instructions is merely an abstraction and can be adapted to most architectures and languages.

We now define the notion of local data dependence ($\xrightarrow{dd}$) and control dependence ($\xrightarrow{cd}$) that will be used to develop the $\xrightarrow{rch}$ relation. For two instruction instances A and B in an execution, A $\xrightarrow{dd}$ B if B reads a value that A writes into its register write set or a private memory location of its processor. (The interaction due to shared-memory data dependence is handled later.) Our notion of control dependence is borrowed from Ferrante et al. [FOW87]. Control dependence is defined in terms of a control flow graph and dominators [ASU86]. Let Prog be the program under consideration, and let E be an execution of program Prog. Consider the control flow graph of any process in program Prog, with the final node in the graph denoted by EXIT. Let C' and D' be two instructions in the control flow graph. C' is *post-dominated* by D' if D' occurs on every path in the control flow graph from C' to EXIT. Let A and B be two instruction instances of processor P in execution E and let A' and B' be the instructions corresponding to A and B in the program text. Instruction instance B is *control dependent* on instruction instance A if the following conditions hold: (i) A $\xrightarrow{po}$ B in execution E, *and* (ii) A' is not post-dominated by B', *and* (iii) there is a path between A' and B' in the control flow graph of processor P such that all the nodes on this path (excluding A',B') are post-dominated by B'. Note that if A $\xrightarrow{cd}$ B, then A' is a control instruction.

To allow for possibly non-terminating executions, we need to augment the control flow graph and the resulting $\xrightarrow{cd}$ relation with additional arcs. Informally, consider a loop in the program that does not terminate in some SC execution. Then, for any instruction instance *i* that is program ordered after an instance of the loop, we require $\xrightarrow{cd}+$ to order *i* after instances of the control instructions of the loop that change the program flow for this loop and cause infinite execution. More formally, let C' be any control instruction that could be executed an infinite number of times in some SC execution E. Suppose an infinite number of successive instances of C' change the control flow of the program in E. Add an auxiliary edge from every such instruction C' to the EXIT node. This ensures that any such control instruction C' is not post-dominated by any of the instructions that follow it in the control flow graph, and so instances of C' are ordered before all instances of all subsequent instructions by $\xrightarrow{cd}+$. The modification described above is not necessary if all SC executions of the program will terminate, or if there are no memory operations that are ordered after possibly non-terminating loops in the control flow graph.

The $\xrightarrow{cd}$ and $\xrightarrow{dd}$ relations are used below to define two other relations, the uniprocessor reach dependence ($\xrightarrow{Udep}$) and the multiprocessor reach dependence ($\xrightarrow{Mdep}$) relations. The $\xrightarrow{rch}$ relation is defined in terms of these two new relations.

Definition F.1: Uniprocessor Reach Dependence

Let X and Y be instruction instances in an execution E of program Prog. X $\xrightarrow{Udep}$ Y in E iff X $\xrightarrow{po}$ Y, and either
(a) X $\xrightarrow{cd}$ Y in E, or
(b) X $\xrightarrow{dd}$ Y in E, or
(c) X and Y occur in another possible SC execution E' of the program Prog, X $\xrightarrow{cd}+$ Z $\xrightarrow{dd}$ Y in E', and Z does not occur in E.

Definition F.2: Multiprocessor Reach Dependence

Let X and Y be instruction instances in an execution E of program Prog. Let Y be an instruction instance that accesses shared-memory. X $\xrightarrow{Mdep}$ Y in E iff any of the following are true.
(a) X $\xrightarrow{po}$ Y in E and X and Y occur in another possible SC execution E' of Prog, where X $\xrightarrow{Udep}+$ Z $\xrightarrow{rpo}$ Y in E', Z is an instance of a shared-memory instruction, for any A $\xrightarrow{dd}$ B constituting the $\xrightarrow{Udep}+$ path from X to Z in E', B also occurs in E, and either Z does not occur in E, or Z occurs in E but is to a different address in E and E', or Z is a write to the same address in E and E' but writes a different value in E and E'.
(b) X $\xrightarrow{po}$ Y in E and X $\xrightarrow{Udep}$ Z $\xrightarrow{rpo}$ Y in E.
(c) X $\xrightarrow{po}$ Y in E and X $\xrightarrow{Mdep}$ Z $\xrightarrow{rpo}$ Y in E.
(d) X $\xrightarrow{po}$ Y or X is the same as Y in E. Further, X generates a competing read R within a loop (a synchronization loop construct for PL3) and Y generates an operation O (different from R) such that R $\xrightarrow{rpo}+$ O.

For PL3, Definition F.2 above assumes an additional constraint on synchronization loop constructs (either the simple definition in Section 3.2.3 or the general definition in Appendix B). The additional constraint is as follows. Consider any read R of a synchronization loop construct in execution E1. If the instruction instance corresponding to R occurs in any other execution E2, then the values that can terminate the instance of the synchronization loop construct corresponding to R are the same in E1 and E2. Without the above constraint, the following additional case must be added to the cases where $\xrightarrow{Mdep}$ holds: X $\xrightarrow{po}$ Y in E, Y generates an exit read of a synchronization loop construct and X $\xrightarrow{Udep}$+ Z, where Z is a branch instruction in the loop.

The $\xrightarrow{rpo}$ relation used as part of the $\xrightarrow{Mdep}$ definition is defined as follows. The $\xrightarrow{spo}$ relation and other conditions mentioned below are derived from the sufficient conditions presented in Chapter 4 for each of the three PL models.[1]

Definition F.3: $\xrightarrow{rpo}$ Relation
Let X and Y be instances of shared-memory instructions in an execution E, and let X' and Y' be the memory operations corresponding to X and Y respectively in E. X $\xrightarrow{rpo}$ Y in E iff either
(a) X' $\xrightarrow{spo}$ Y' or X' $\xrightarrow{spo'}$ Y' in E, or X' $\xrightarrow{spo''}$ Y' in E for PL3, or
(b) X' $\xrightarrow{po}$ Y' in E and X' $\xrightarrow{po}$ Y' is used in the uniprocessor dependence condition to impose an order between the sub-operations of X' and Y', or
(c) X'=W $\xrightarrow{po}$ Y'=R in E and the initiation condition requires that W_{init}(i) $\xrightarrow{xo}$ R_{init}(i) in E.

Finally, the following defines the reach relation. The $\xrightarrow{rch'}$ relation is also used in the next appendix to relax the uniprocessor correctness condition.

Definition F.4: Reach' Relation
Given an execution E and instruction instances X and Y in E (where X may or may not be the same as Y), X $\xrightarrow{rch'}$ Y in E iff X and Y are instances of memory instructions, X generates a shared-memory read, Y generates a shared-memory write, and X $\{\xrightarrow{Udep} \mid \xrightarrow{Mdep}\}+$ Y in E. For two different memory operations, X' and Y', from instruction instances X and Y respectively, X' $\xrightarrow{rch'}$ Y' iff X' is a read, Y' is a write, X' $\xrightarrow{po}$ Y' and X $\xrightarrow{rch'}$ Y.

Definition F.5: Reach Relation
Given an execution E and instruction instances X and Y in E, X $\xrightarrow{rch}$ Y in E iff X $\xrightarrow{rch'}$ Y and X generates a memory read that reads the value of another processor's write. The $\xrightarrow{rch}$ relation among memory operations is defined in an analogous manner to $\xrightarrow{rch'}$.

The reach relation is a transitive closure of the uniprocessor reach dependence ($\xrightarrow{Udep}$) and the multiprocessor reach dependence ($\xrightarrow{Mdep}$) relations. The uniprocessor component corresponds to uniprocessor data and control dependence, while the multiprocessor component corresponds to dependences that are present due to the memory consistency model. The components that make up $\xrightarrow{Udep}$ and $\xrightarrow{Mdep}$ are defined for a given execution E. Both relations also require considering other sequentially consistent executions of the program, and determining if an instruction in one execution occurs in the other execution.[2] For an instruction instance from one execution to occur in another, we do not require that locations accessed or the values read and written by the corresponding instruction instances in the two executions be the same; we are only concerned with whether the specific instances appear in the execution. In the absence of constructs such as loops and recursion, it is straightforward to determine if an instance that appears in one execution also appears in another. In the presence of constructs such as loops and recursion, care has to be taken to match consistent pairs of instruction instances. Instruction instances between two executions are matched consistently if the set

[1] We allow the relation X $\xrightarrow{spo}$ Y to be dependent on the addresses of X and Y. In this case, if the same pair of memory instructions accesses different addresses in two executions, it is possible for the pair to be ordered by $\xrightarrow{spo}$ in one execution but not in the other.

[2] It may be simpler to be conservative and consider all possible executions of the program and not just SC executions (for E') in the definitions of $\xrightarrow{Udep}$ and $\xrightarrow{Mdep}$.

```
a1:  u = A;
b1:  if (u == 1) {
c1:      v = B;
d1:  }
e1:  C = v;
```

(a)

```
a1:  u = D;
b1:  if (u == 1) {
c1:      v = Flag;
d1:  }
e1:  E = 1;
```

(b)

Figure F.1: Examples to illustrate the reach condition.

of instances that are considered to appear in both executions have the same program order relation between them in both executions, and are the maximal such sets. (A set S with property P is a maximal set satisfying property P if there is no other set that is a superset of S and also satisfies property P.)

To illustrate the above concepts, consider the example in Figure F.1(a). Assume an execution E where the read of A returns the value of 0. Thus, u is 0 and the read of B and assignment to v do not occur in execution E. Therefore, in E, there is no instruction instance ordered before the write to C by either $\xrightarrow{cd}$ or by $\xrightarrow{dd}$. Thus, parts (a) and (b) of the definition of $\xrightarrow{Udep}$ do not order any instruction instance before (e1) by $\xrightarrow{Udep}$. However, if we consider a possible SC execution E' where the return value of A is 1 (assume it is the value of another processor's write), then we have (b1) $\xrightarrow{cd}$ (c1) $\xrightarrow{dd}$ (e1) in E', (c1) does not occur in E, and so (b1) $\xrightarrow{Udep}$ (e1) in E by part (c) of the definition of $\xrightarrow{Udep}$. Further, (a1) $\xrightarrow{dd}$ (b1) in E and so (a1) $\xrightarrow{Udep}$ + (e1) in E and so (a1) $\xrightarrow{rch}$ (e1) in E.

Figure F.1(b) shows an example that uses $\xrightarrow{Mdep}$. Assume the PL1 model, with the read of Flag being the only competing operation. Consider an execution E where the return value for the read of D is 0. Therefore, (c1) does not occur in E. Consider a possible SC execution E' where the read of D returns 1 (assume the value is from another processor's write), causing (c1) to be executed. Under PL1, Rc $\xrightarrow{po}$ W constitutes Rc $\xrightarrow{rpo}$ W. Therefore, in E', (c1) $\xrightarrow{rpo}$ (e1). Since we have (b1) $\xrightarrow{cd}$ (c1) $\xrightarrow{rpo}$ (e1) in E' and (c1) does not occur in E, we have (b1) $\xrightarrow{Mdep}$ (e1) in E by Definition F.2(a). Further, (a1) $\xrightarrow{dd}$ (b1) in E and so (a1) $\xrightarrow{Udep}$ (b1) $\xrightarrow{Mdep}$ (e1) in E and so (a1) $\xrightarrow{rch}$ (e1) in E. Note that in this example, $\xrightarrow{Udep}$ does not hold between (b1) and (e1).

The proofs for the sufficient conditions of models such as PLpc have shown that the reach relation as formalized above is adequate [AGG+93]. It is likely that these conditions can be formalized even more aggressively without violating the semantics of the various PL models.

Appendix G

Aggressive Form of the Uniprocessor Correctness Condition

This appendix formalizes the relaxation of the uniprocessor correctness condition (Condition 4.1) discussed in Section 4.2. The formalism presented below is the same as that developed for the PLpc model [AGG+93]. In contrast to Condition 4.1, the extension presented here allows certain operations to execute before a preceding loop may terminate. Definition G.1 below formalizes the notion of a preceding loop. Condition G.1 formalizes the conditions that determine whether an operation can execute before its preceding loop terminates.

Definition G.1: Loop
A *loop* in a control flow graph refers to a cycle in the control flow graph. A loop L *does not terminate* in an execution iff the number of instances of instructions from loop L in the execution is infinite.

Condition G.1: Infinite Execution Condition
Consider an execution E of program Prog that contains instruction instance j of instruction j', and j' is a write instruction.
(a) If *j'* follows loop L that does not terminate in some SC execution, then the number of instances of instructions in E that are from loop L and that are ordered by program order before *j* is finite.
(b) The number of instruction instances that are ordered before *j* by $\xrightarrow{rch'}$ in E is finite.

With the above conditions, a processor can execute a read operation before it is known whether the previous loops in program order will terminate. For a write operation, the processor is allowed to execute the write before a previous loop as long as the loop is known to terminate in every SC execution and as long as no memory operations from the loop are ordered before the write by $\xrightarrow{rch'}$ (defined in the previous appendix). Referring back to the examples in Section 4.2, the above conditions allow the optimizations discussed for the example in Figure 4.12, while appropriately disallowing the optimizations discussed for the two examples in Figure 4.13 (Condition G.1(a) and (b) handle the examples in Figure 4.13(a) and (b), respectively).

Most programs are written so that either they will terminate in all SC executions, or there are no shared-memory operations that follow a potentially non-terminating loop. In addition, the information about whether a loop will always terminate in an SC execution is often known to the programmer and can be easily obtained. Thus, the above relaxation of the uniprocessor correctness condition is applicable to a large class of programs.

Appendix H

Aggressive Form of the Termination Condition for Writes

This appendix describes a relaxation of the termination condition for writes specifically for more aggressively supporting the properly-labeled models.

H.1 Relaxation of the Termination Condition

Condition 4.5 in Chapter 4 specifies when a write sub-operation must appear in the execution. This condition can be further relaxed for the PL models. Condition H.1 below specifies this relaxation.

> **Condition H.1: Aggressive Termination Condition for Writes for Properly-Labeled Models**
> Suppose a write sub-operation W_{init}(i) (corresponding to operation W) by P_i appears in the execution. Consider an operation O on P_j that conflicts with W. The termination condition requires W(j) (i.e., the sub-operation with respect to P_j) to appear in the execution if both W and O are labeled as competing (includes sync and non-sync labels for PL2, and loop, non-loop, and non-sync for PL3).

The relaxed condition turns out to be sufficient for supporting the three properly-labeled models. The proof for the above is based on the general framework developed in Adve's thesis [Adv93]; this is the same framework that was used to prove the sufficient conditions for the PLpc model [GAG+92]. For the three properly-labeled models, Condition H.1 above trivially satisfies the constraints on termination assumed by Adve's framework.

Figure H.1 shows a simple example to illustrate the difference between Condition H.1 and Condition 4.5. Assume the PL1 model, and assume the operations on P1 and P2 are labeled as competing (shown in bold) while the operations on P3 are labeled as non-competing. The sufficient conditions for PL1 shown in Figure 4.14 along with Condition 4.5 require that the while loops on both P2 and P3 terminate in every execution. However, if we substitute Condition H.1 for Condition 4.5, then the specification allows the while loop on P3 to not terminate in some executions. This is because the write of A on P1 is not required to complete with respect to P3 based on Condition H.1. Note that the program in Figure H.1 is not properly-labeled.

P1		P2		P3	
a1:	**A = 1;**	*a2:*	**while (A == 0);**	*a3:*	while (A == 0);

Figure H.1: Example to illustrate the more aggressive termination condition.

P1		P2		P3	
a1:	**A = 1;**	*a2:*	**while (A == 0);**		
		b2:	**B = 1;**	*a3:*	**while (B == 0);**
				b3:	**while (A == 0);**

Figure H.2: Example of the aggressive termination condition for the PL3 model.

In fact, the distinction between Conditions 4.5 and H.1 arises only for non-PL programs; the relaxation in Condition H.1 simply exploits the fact that the properly-labeled models are not required to provide sequentially consistent results to non-PL programs.

Figure H.2 shows an example specific to the PL3 model. Assume all operations are labeled as loop writes and reads, which satisfies the labeling requirements for PL3. None of the system requirements for PL3 (see Figure 4.16) impose an execution order between the sub-operation of the write of A on P1 with respect to P3 and the read sub-operation of A on P3. Therefore, if we remove the termination condition (either Condition 4.5 or Condition H.1), the sufficient system conditions for PL3 would not require the second while loop on P3 to terminate in every execution. Note that with Condition H.1, it is important for the read of A on P3 be labeled either a loop, non-loop, or non-sync to guarantee that the while loop terminates. In contrast to the PL3 specification, the same example does not depend on the termination condition (either version) for correctness if we assume either the PL1 or PL2 sufficient conditions (still assuming all operations are labeled as competing). This is because the multiprocessor dependence chain from the write of A on P1 to the read of A on P2 is upheld by the sufficient conditions for either PL1 and PL2, thus already requiring the write of A on P1 to complete with respect to P3 before the read of A on that processor.

H.2 Implications of the Relaxed Termination Condition on Implementations

As discussed in Chapter 5, most hardware implementations end up naturally ensuring the termination for *all* writes and write sub-operations. Therefore, Condition H.1 would not benefit most of these implementations. However, there are a few hardware implementations that could exploit the more relaxed condition. For example, consider implementations that aggressively exploit the early acknowledgement optimization described in Section 5.4.1 whereby incoming invalidations and updates are *only* serviced when the processor enforces a program order (delayed consistency implementations [DWB$^+$91] use a similar optimization). With this optimization, it is possible to build an implementation that supports the PL1 model, for example, and still behaves aggressively for examples such as the one in Figure H.1 by allowing executions where the loop on P3 does not terminate.

The performance gains from exploiting the more aggressive termination condition in a hardware implementation are likely to be small, however. The main reason is that the less aggressive condition (Condition 4.5) can be supported quite efficiently. For example, even in an implementation such as the one described above, the stricter semantics can be satisfied by guaranteeing that the incoming invalidation and update requests will be periodically serviced (e.g., by flushing the incoming queue once in a while), even if the period is chosen to be extremely long and non-deterministic. The semantics of the stricter condition may also be more desirable for programs with non-synchronized accesses to data that are not properly-labeled for performance reasons (see Section 3.6).

The relaxed semantics can be more beneficial for compiler optimizations, specifically optimizations such as register allocation. With the conservative semantics, the compiler must treat non-competing reads conservatively if it cannot determine that the read will only be executed a finite number of times. For example, the read on P3 in the example from Figure H.1 cannot be register allocated with the conservative termination condition (see Section 5.10.4). The more aggressive termination condition allows the compiler to always register allocate non-competing reads even if there is potential for the read to execute an infinite number of times.[1]

Finally, due to a subtle reason, most software distributed shared memory systems (see Section 5.6) would not benefit from the relaxed termination condition. This is because the restrictions that are typically placed on programs by such systems make it impossible to write a program that can distinguish between Condition 4.5 and Condition H.1.[2] This observation allows the implementation to exploit optimizations enabled by Condition H.1, yet provide behavior that is indistinguishable from that of Condition 4.5 for all the possible programs.

[1] Analogous to hardware implementations, it is possible to satisfy the conservative condition even when a location is register allocated by periodically loading the value from memory into the register. However, generating the appropriate compiled code to achieve this can be challenging.

[2] In other words, it is not possible to write a program with a competing label for some writes and a non-competing label for other reads (or writes) to the same location.

Appendix I

Aggressive Specifications for Various System-Centric Models

This appendix presents the specification for various system-centric models using the framework described in Chapter 4. In all cases, the set of conditions presented here are more aggressive than the original specification for each model because we impose execution order constraints among competing sub-operations only. However, these aggressive conditions are semantically equivalent to the original specifications of these models; i.e., an implementation that obeys the aggressive conditions *appears as if* it obeys the original conditions and vice versa. Consequently, for the programmer, the two sets of conditions are equivalent; however, for the system designer, our conditions expose more aggressive implementations.

Chapter 4 already provides the specifications for SC, RCsc, and RCpc. This appendix provides specifications for the rest of the models: IBM-370, TSO, PC, PSO, WO, Alpha, RMO, and PowerPC. In addition, we provide the specification for the extended versions of the models described in Section 4.4: TSO+, PC+, PSO+, RCpc+, PowerPC+. Section I.2 describes how the reach relation is adapted to the system-centric models; this condition is only relevant to the models that relax the read-write program order (i.e., WO, RCsc, RCpc, Alpha, RMO, PowerPC, RCpc+, and PowerPC+). Finally, Section I.3 describes how the aggressive uniprocessor correctness condition is adapted. The aggressive specifications for TSO, PC, PSO, and WO, along with proofs of their equivalence to the original specifications have appeared in a previous technical report [GAG+93] using a slightly different specification format; the proofs of equivalence for the other specifications are similar in nature.

I.1 Aggressive Specification of the Models

Figures I.1-I.13 present the various aggressive specifications. The notation used to identify different types of operations (e.g., synchronization and data operations in WO) is consistent with the notation used in Chapters 2 and 4.

The original specification of the models are based on the following sources: IBM-370 [IBM83], TSO and PSO [SFC91, SUN91], PC [GLL$^+$90, GGH93b], WO [DSB86, Sch89], Alpha [Sit92], RMO [WG94], and PowerPC [CSB93, MSSW94]. For simplicity, we only specify the behavior of read and write operations to a single data granularity. Therefore, we do not capture the behavior of other operations such as instruction fetches, operations to different data granularities (e.g, byte, word), or I/O operations specified by the original specifications of some of the commercial models (i.e., Alpha, RMO, and PowerPC). In addition, we have made the following assumptions since certain behaviors are not precisely defined by some of the original specifications. The first assumption relates to the reach relation. The WO, Alpha, and PowerPC models do not specify the reach condition and are therefore underspecified. The RMO model, on the other hand, defines a conservative version of the reach condition; for example, all writes past a conditional branch are delayed until the branch is resolved. For these models, we adopt the more formal and aggressive reach relation that is described in Section I.2. For WO, we also assume the following: the cache coherence condition holds for all writes, and the same location can be read and written by both synchronization and data operations.

define $\xrightarrow{spo}$,$\xrightarrow{spo'}$**:**

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

R $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ W
Ws $\xrightarrow{po}$ R
W $\xrightarrow{po}$ Rs
W $\xrightarrow{po}$ FENCE $\xrightarrow{po}$ R
W $\xrightarrow{po}$ R where W and R conflict

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{spo'}\}+$ Y

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y
R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.
Condition 4.5: termination condition for writes; applies to all write sub-operations.
Condition 4.6: return value for read sub-operations.
Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ RW
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW
RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ RW
W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.1: Aggressive conditions for IBM-370.

define $\xrightarrow{spo}$**:** X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

R $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ W
W (in RMW) $\xrightarrow{po}$ R
W $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y
R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:
Condition 4.4: initiation condition for reads and writes.
Condition 4.5: termination condition for writes; applies to all write sub-operations.
Condition 4.6: return value for read sub-operations.
Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of
uniprocessor dependence: RW $\xrightarrow{po}$ W
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW
RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ RW
W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ R
then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.2: Aggressive conditions for TSO.

define $\xrightarrow{spo}$**:** X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

R $\xrightarrow{po}$ RW

W $\xrightarrow{po}$ W

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X $\xrightarrow{co}$ Y

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to all write sub-operations.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ RW

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.3: Aggressive conditions for PC.

define $\xrightarrow{spo}$**:** X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

R $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ W
W (in RMW) $\xrightarrow{po}$ RW
W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y
R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:
Condition 4.4: initiation condition for reads and writes.
Condition 4.5: termination condition for writes; applies to all write sub-operations.
Condition 4.6: return value for read sub-operations.
Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of
uniprocessor dependence: RW $\xrightarrow{po}$ W
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW
RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}$+ RW
W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}$+ R
then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.4: Aggressive conditions for PSO.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$, $\xrightarrow{spo''}$:

X $\xrightarrow{spo''}$ Y if X and Y are the first and last operations in one of

RWs $\xrightarrow{po}$ RWs

RW $\xrightarrow{po}$ RWs

RWs $\xrightarrow{po}$ RW

X $\xrightarrow{spo'}$ Y if X $\{\xrightarrow{spo''}\}$ $\{\xrightarrow{rch} \mid \xrightarrow{spo''}\}*$ Y

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} \mid \xrightarrow{spo''}\}+$ Y

define $\xrightarrow{sco}$, $\xrightarrow{sco'}$:

X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on the same processor

X $\xrightarrow{sco'}$ Y if X and Y are the first and last operations in R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on different processors

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to all write sub-operations.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W (or RW $\xrightarrow{po}$ RW)

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ $\{(A \xrightarrow{sco} B \xrightarrow{spo}) \mid (A \xrightarrow{sco'} B \xrightarrow{spo'})\}+$ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo'}$ $\{(A \xrightarrow{sco} B \xrightarrow{spo}) \mid (A \xrightarrow{sco'} B \xrightarrow{spo'})\}+$ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.5: Aggressive conditions for WO.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$:

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{po}$ MB $\xrightarrow{po}$ Y

W $\xrightarrow{po}$ WMB $\xrightarrow{po}$ W

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} \mid \xrightarrow{spo'}\}+$ Y

define $\xrightarrow{sco}$: X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

Conditions on $\xrightarrow{xo}$:

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to write sub-operations for *all writes*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W (or RW $\xrightarrow{po}$ RW)

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{po}$ RW

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.6: Aggressive conditions for Alpha.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$, $\xrightarrow{spo''}$**:**

X $\xrightarrow{spo''}$ Y if X and Y are the first and last operations in one of

R $\xrightarrow{po}$ MEMBAR(RR) $\xrightarrow{po}$ R

R $\xrightarrow{po}$ MEMBAR(RW) $\xrightarrow{po}$ W

W $\xrightarrow{po}$ MEMBAR(WR) $\xrightarrow{po}$ R

W $\xrightarrow{po}$ MEMBAR(WW) $\xrightarrow{po}$ W

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

W (in RMW) $\xrightarrow{po}$ MEMBAR(RR) $\xrightarrow{po}$ R

W (in RMW) $\xrightarrow{po}$ MEMBAR(RW) $\xrightarrow{po}$ W

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} | \xrightarrow{spo'} | \xrightarrow{spo''}\}+$ Y

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to all write sub-operations.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.7: Aggressive conditions for RMO.

define $\xrightarrow{spo}$**:** X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of
X $\xrightarrow{po}$ SYNC $\xrightarrow{po}$ Y

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X $\xrightarrow{co}$ Y

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:
Condition 4.4: initiation condition for reads and writes.
Condition 4.5: termination condition for writes; applies to all write sub-operations.
Condition 4.6: return value for read sub-operations.
Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of
uniprocessor dependence: RW $\xrightarrow{po}$ W (or RW $\xrightarrow{po}$ RW)
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ RW
reach: R $\xrightarrow{rch}$ $\{$W $\xrightarrow{co'}$ R $\xrightarrow{rch}$ $\}+$ W
then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.8: Aggressive conditions for PowerPC.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$, $\xrightarrow{spo''}$**:**

X $\xrightarrow{spo''}$ Y if X,Y are the first and last operations in one of

R $\xrightarrow{po}$ RW

W $\xrightarrow{po}$ W

W $\xrightarrow{po}$ MEMBAR(WR) $\xrightarrow{po}$ R

X $\xrightarrow{spo'}$ Y if X,Y are the first and last operations in one of

W (in RMW) $\xrightarrow{po}$ R

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{spo'} | \xrightarrow{spo''}\}+$ Y

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to all write sub-operations.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.9: Aggressive conditions for TSO+.

define $\xrightarrow{spo}$**:** X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of

R $\xrightarrow{po}$ RW

W $\xrightarrow{po}$ W

W $\xrightarrow{po}$ Fence $\xrightarrow{po}$ R

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R $\xrightarrow{co}$ W_atomic $\xrightarrow{co}$ R

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to all write sub-operations.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ RW

W_atomic $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}+$ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.10: Aggressive conditions for PC+.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$, $\xrightarrow{spo''}$**:**

X $\xrightarrow{spo''}$ Y if X,Y are the first and last operations in one of

R $\xrightarrow{po}$ RW

W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ W

W $\xrightarrow{po}$ MEMBAR(WR) $\xrightarrow{po}$ R

X $\xrightarrow{spo'}$ Y if X,Y are the first and last operations in one of

W (in RMW) $\xrightarrow{po}$ RW

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{spo'} \mid \xrightarrow{spo''}\}+$ Y

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R $\xrightarrow{co}$ W $\xrightarrow{co}$ R

Conditions on $\xrightarrow{xo}$**:**

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to all write sub-operations.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ RW

W $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.11: Aggressive conditions for PSO+.

define $\xrightarrow{spo}$, $\xrightarrow{spo'}$:

X $\xrightarrow{spo'}$ Y if X and Y are the first and last operations in one of

Rc $\xrightarrow{po}$ RWc

Wc $\xrightarrow{po}$ Wc

Wc $\xrightarrow{po}$ Fence $\xrightarrow{po}$ Rc

Rc_acq $\xrightarrow{po}$ RW

RW $\xrightarrow{po}$ Wc_rel

X $\xrightarrow{spo}$ Y if X $\{\xrightarrow{rch} \mid \xrightarrow{spo'}\}+$ Y

define $\xrightarrow{sco}$, $\xrightarrow{sco'}$: X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

R1 $\xrightarrow{co}$ W $\xrightarrow{co}$ R2 where R1,R2 are on the same processor

R $\xrightarrow{co}$ Wc_atomic $\xrightarrow{co}$ R

Conditions on $\xrightarrow{xo}$:

(a) the following conditions must be obeyed:

Condition 4.4: initiation condition for reads and writes.

Condition 4.5: termination condition for writes; applies to write sub-operations for *all competing writes*.

Condition 4.6: return value for read sub-operations.

Condition 4.7: atomicity of read-modify-write operations.

(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of

uniprocessor dependence: RW $\xrightarrow{po}$ W

coherence: W $\xrightarrow{co}$ W

multiprocessor dependence chain: one of

W $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW

RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ RW

Wc_atomic $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}\}+$ R

then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.12: Aggressive conditions for RCpc+.

define $\xrightarrow{spo}$**:** X $\xrightarrow{spo}$ Y if X and Y are the first and last operations in one of
X $\xrightarrow{po}$ SYNC $\xrightarrow{po}$ Y

define $\xrightarrow{sco}$**:** X $\xrightarrow{sco}$ Y if X and Y are the first and last operations in one of
X $\xrightarrow{co}$ Y
R $\xrightarrow{co}$ W_atomic $\xrightarrow{co}$ R

Conditions on $\xrightarrow{xo}$**:**
(a) the following conditions must be obeyed:
Condition 4.4: initiation condition for reads and writes.
Condition 4.5: termination condition for writes; applies to all write sub-operations.
Condition 4.6: return value for read sub-operations.
Condition 4.7: atomicity of read-modify-write operations.
(b) given memory operations X and Y, if X and Y conflict and X,Y are the first and last operations in one of
uniprocessor dependence: RW $\xrightarrow{po}$ W (or RW $\xrightarrow{po}$ RW)
coherence: W $\xrightarrow{co}$ W
multiprocessor dependence chain: one of
RW $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}$+ RW
W_atomic $\xrightarrow{co}$ R $\xrightarrow{spo}$ RW
W_atomic $\xrightarrow{sco}$ R $\xrightarrow{spo}$ $\{$A $\xrightarrow{sco}$ B $\xrightarrow{spo}$ $\}$+ R
reach: R $\xrightarrow{rch}$ $\{$W $\xrightarrow{co'}$ R $\xrightarrow{rch}$ $\}$+ W
then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Figure I.13: Aggressive conditions for PowerPC+.

I.2 Reach Relation for System-Centric Models

The reach relation used for the system-centric models is virtually identical to the relation formalized in Appendix F for the PL models. The only differences are in Definition F.2(d) and Definition F.3(a) which must be customized for the system-centric models.

Definition F.2(d) can be customized as follows: "X $\xrightarrow{po}$ Y or X is the same as Y in E. Further, X generates a read R within a loop and Y generates an operation O (different from R) such that R $\xrightarrow{rpo}$+ O. The read R must be a synchronization read in WO, a competing read in RCsc or RCpc (or RCpc+), and any read in Alpha, RMO, or PowerPC (or PowerPC+)."

The definition of the $\xrightarrow{rpo}$ relation (Definition F.3) must also be customized for each model. Definition F.3(a) can be customized as follows: "X' $\xrightarrow{spo}$- Y' in E". The $\xrightarrow{spo}$- relation represents the $\xrightarrow{spo}$ relation for a given model *minus* the reach relation in the case of WO, RCsc, RCpc (and RCpc+), Alpha, and RMO.[1] For example, for RCpc, the $\xrightarrow{rpo}$ relation is as follows. Let X and Y be instruction instances and let X' and Y' be memory operations corresponding to X and Y respectively. X $\xrightarrow{rpo}$ Y if X' $\{\xrightarrow{spo'}\}+$ Y' (i.e., $\xrightarrow{spo'}$ as defined in Figure 4.20 in Chapter 4), or if X' and Y' conflict and X' $\xrightarrow{po}$ Y'.

I.3 Aggressive Uniprocessor Correctness Condition for System-Centric Models

The original specifications of the system-centric models do not explicitly state whether they assume the conservative or aggressive uniprocessor correctness condition. We believe that *except* for SC, the intent of the remaining models can be captured with the more aggressive uniprocessor correctness condition if one additional condition is enforced. The additional condition should ensure the following two properties:

1. For certain classes of well-behaved programs (e.g., properly-labeled programs), the model should appear sequentially consistent.

2. Consider any pair of shared-memory instructions X' and Y' from the same processor where X' is before Y' in the control flow graph of the processor. If the original specification of the model requires sub-operations of instances of X' to execute before those of Y', then the new condition should ensure that only a finite number of instances of X' are ordered by program order before any instance of Y'.

This condition is formalized below as Condition I.1 and is called the infinite execution condition. Parts (a), (b), and (c) of the condition cover property (1) above and part (c) also covers property (2) above. Definition I.1 is the same as Definition G.1 in Appendix G. Definition I.2 is particular to system-centric models and is used in Condition I.1(c) to satisfy property (2) above.

Definition I.1: Loop
A *loop* in a control flow graph refers to a cycle in the control flow graph. A loop L *does not terminate* in an execution iff the number of instances of instructions from loop L in the execution is infinite.

[1] Removing the reach relation is important since otherwise there would be a circularity in the specification since we use $\xrightarrow{spo}$ to define $\xrightarrow{rch}$ through $\xrightarrow{rpo}$ and $\xrightarrow{spo}$ itself uses $\xrightarrow{rch}$. The reach condition is specified in a more aggressive way in PowerPC and is therefore not merged with the $\xrightarrow{spo}$ relation.

P1	P2	P3
A = 1;	while (B==0);	if (P==1)
B = 1;	if (A==0) {	D = 1;
	while (1) { C = 1;}	
	P = 1;	
	}	

Figure I.14: Example illustrating the infinite execution condition.

Definition I.2: Infinite Execution Order ($\xrightarrow{ieo}$)

Let X and Y be two shared-memory instruction instances in an execution E, with X' and Y' being the corresponding memory operations.

If E is an execution on an IBM-370 system, then X $\xrightarrow{ieo}$ Y iff X' $\xrightarrow{spo}$ Y'.

If E is an execution on a TSO system, then X $\xrightarrow{ieo}$ Y iff X'=R $\xrightarrow{po}$ Y'=RW or X'=W $\xrightarrow{po}$ Y'=W.

If E is an execution on a PC system, then X $\xrightarrow{ieo}$ Y iff X' $\xrightarrow{spo}$ Y'.

If E is an execution on a PSO system, then X $\xrightarrow{ieo}$ Y iff X'=R $\xrightarrow{po}$ Y'=RW or X'=W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ Y'=W.

If E is an execution on a WO system, then X $\xrightarrow{ieo}$ Y iff X' $\{\xrightarrow{spo''}\}+$ Y'.

If E is an execution on a RCsc system, then X $\xrightarrow{ieo}$ Y iff X' $\{\xrightarrow{spo''} \mid \xrightarrow{spo'''}\}+$ Y'.

If E is an execution on a RCpc or RCpc+ system, then X $\xrightarrow{ieo}$ Y iff X' $\{\xrightarrow{spo'}\}+$ Y'.

If E is an execution on an Alpha system, then X $\xrightarrow{ieo}$ Y iff X' $\{\xrightarrow{spo'}\}+$ Y'.

If E is an execution on a RMO system, then X $\xrightarrow{ieo}$ Y iff X' $\{\xrightarrow{spo''}\}+$ Y'.

If E is an execution on a PC+ system, then X $\xrightarrow{ieo}$ Y iff X' $\{\xrightarrow{spo}\}+$ Y'.

If E is an execution on a TSO+ or PSO+ system, then X $\xrightarrow{ieo}$ Y iff X' $\{\xrightarrow{spo''}\}+$ Y'.

Condition I.1: Infinite Execution Condition

Consider an execution E of program Prog that contains instruction instance j of instruction j'.

(a) If loop L does not terminate in some SC execution, then the number of instances of instructions in E that are from loop L and that are ordered by program order before *j* is finite if j is a write instruction instance.

(b) The number of instruction instances that are ordered before *j* by $\xrightarrow{rch'}$ in E is finite (i.e., for models that have $\xrightarrow{rch'}$ defined: WO, RCsc, RCpc, Alpha, RMO, PowerPC, RCpc+, and PowerPC+). By definition, j is a write instruction instance.

(c) The number of instruction instances that are ordered before *j* by $\xrightarrow{ieo}$ in E is finite. This clause does not apply to the PowerPC and PowerPC+ models.

Figure I.14 shows an example to illustrate the use of Condition I.1(c) which is particular to the system-centric models. Assume the WO model with only the write of C and write of P on P2 labeled as synchronization operations. Note that more operations would have been labeled as synchronization for a port of a properly-labeled version of this program to WO. P3 would never execute the write to D on P3 in any WO execution according to the original specification of WO. However, this write may get executed if we allow the write to P to occur on P2 before its preceding while loop ends. This is disallowed by Condition I.1(c), but would have been allowed by Conditions I.1(a) and (b) alone.

Appendix J

Extensions to Our Abstraction and Specification Framework

We begin by clarifying some of the simplifying assumptions we made in Chapter 4 about the notion of result for an execution. Section J.2 considers the issues that arise in modeling external I/O devices. Section J.3 describes other event types, such as instruction fetches, that may also need to be modeled. We describe the steps that are required to incorporate extra events into our framework in Section J.4. Finally, Section J.5 describes a potentially more realistic notion of result that attempts to incorporate the larger set of events. Chapter 5 and Appendix Q further describe implementation issues with respect to I/O operations, instruction fetches, and multiple granularity data operations.

J.1 Assumptions about the Result of an Execution

The definition for result should capture sufficient behavior for an execution to include the aspects that a programmer would consider as the outcome of a program. Yet, we want to exclude other behavior so as to allow flexibility in specifications and implementations. The simple notion of result introduced by Definition 4.6 in Section 4.1.1 is useful for isolating the behavior of shared memory operations. Nevertheless, it fails to capture the behavior of events such as I/O operations. Below, we describe how making additional aspects of an execution visible to the programmer fundamentally changes the basic notion of result and assumptions about equivalence between specifications. Many of the issues discussed apply to uniprocessor systems as well.

Our simple notion of result consists of the values returned by the read operations that occur in a given execution. This notion provides us with substantial freedom in reordering memory operations, and as long as the reads in the execution return the same values as with a more conservative system, the aggressive and conservative systems are considered to be equivalent. One example of this is the aggressive conditions for SC, shown in Figure 4.7, whereby orders are imposed among conflicting operations only. A more dramatic

(a)		(b)	
a1:	A = 1;	*a1:*	B = 1;
b1:	B = 1;	*b1:*	u = B;
c1:	u = B;	*c1:*	A = 1;
d1:	v = A;	*d1:*	v = A;

Figure J.1: Illustrating memory operation reordering in uniprocessors.

example is the aggressive conditions for properly-labeled programs (e.g., Figure 4.14 for PL1) which allow a large number of reordering optimization and yet guarantee sequentially consistent results for such programs. The situation is quite similar for uniprocessors. The optimizations exploited in uniprocessors depend on the assumption that the result of executing a program on a conservative system that strictly maintains the sequential order among *all* memory operations is the same as executing the program on an aggressive system that maintains the sequential order among conflicting operations *only*.

The equivalences discussed above do not necessarily hold if we allow programmers to observe additional characteristics of an execution beyond the values returned by read operations. In effect, additional knowledge about the execution may enable the programmer to partially or fully reconstruct the actual execution order among memory operations. With this information, programmers can trivially distinguish executions on the conservative and aggressive systems discussed above. Consider the uniprocessor example in Figure J.1(a). Figure J.1(b) shows a legal transformation of the program that may arise due to either dynamic reordering in hardware or static reordering by the compiler. If we consider the values returned by read operations, the possible results for the two program segments are identical (i.e., (u,v)=(1,1)). However, the order in which memory locations are modified is clearly different in the two programs, making it possible to distinguish the executions if this order is either directly or indirectly observable.

There are numerous ways in which programmers may be able to partially reconstruct the actual execution order for an execution in either uniprocessor or multiprocessor systems. For example, the ability to monitor the contents of memory allows the programmer to infer the actual order among memory writes. Referring back to the program segments in Figure J.1, this makes it easy to determine whether the writes to locations A and B are reordered. A programmer may be able to monitor memory contents under a number of scenarios. For example, most debuggers allow the programmer to single step a program and examine the contents of registers and memory. For this reason, debugging is often done with unoptimized code even in uniprocessors since the reordering optimizations done by the compiler may no longer be transparent. In actual runs of the program, paging activity can indirectly provide snapshots of memory contents for the pages that are pushed to disk, even though programmers typically do not have default access to the swap data on disk. Hardware monitors may also allow the order of writes to memory to be observed. Any of these approaches enable the programmer to distinguish among executions even if the return values for read operations are identical.

Figure J.2 shows a multiprocessor example where having an accurate measure of real time allows the programmer to observe anomalies. Assume a conceptual implementation that provides an accurate time for when an operations is issued and when it is considered to be complete (i.e., when the processor issues the next operation). For example, according to the time bars shown in the figure, the write to A on P_1 is considered

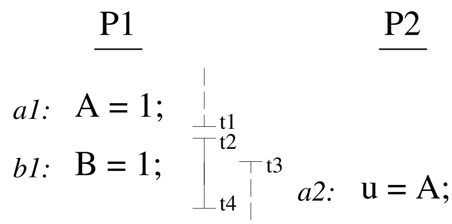


Figure J.2: An example multiprocessor program segment.

complete at time t1 and the read of A on P_2 is initiated at time t3. A programmer who is aware of such actual timing may naturally expect the read of A to return the value 1. However, even the sequential consistency model allows the value of 0 to be returned for the read of A, regardless of the actual timing of operations. Another way programmers may observe anomalies is if there is an independent means of communication among the processors. For example, P_1 may signal P_2, through a mechanism separate from memory, when it initiates the write to B. If P_2 observes this signal in an execution before it actually initiates its read of A, then the return value of 0 would again seem anomalous even though it is allowed under sequential consistency. The above examples show that the equivalence among specifications and implementations is heavily dependent on what is considered to constitute the result of an execution.

Even though our simple notion of result has many merits, it also has some shortcomings that we describe below. The most serious shortcoming is that the simple notion does not fully encompass a pragmatic notion of result as understood by the programmer. For example, external input and output are not included in the event types even though external output operations are often considered as part of the outcome for the program by a programmer. Similarly, the final value of some memory locations may be considered as part of the execution outcome by the programmer (e.g., these values may be saved to disk), but are not included in our simple notion of result.[1] The other shortcoming of our simple notion of result is that it can capture excess behavior; programmers often do not consider (or cannot observe) the value for each and every read operation as part of the execution outcome. Including the return value of all reads in the notion of result can therefore exclude certain desirable optimizations. For example, common compiler optimizations, such as register allocation, common subexpression elimination, and hoisting loop invariants, have the effect of eliminating some read operations. Other compiler or hardware optimizations, such as speculative execution, may introduce extra read operations into the execution. Another example is the optimization allowed by the PL3 model for synchronization loops where operations issued by unsuccessful iterations of the loop must be ignored for achieving equivalence with sequential consistency (refer back to Section 3.2.3).

We will discuss a more realistic notion of result in Section J.5 after discussing various extra events that may need to be incorporated into the system abstraction.

J.2 External Devices

This section describes the issues that arise in modeling external input and output devices. We begin by discussing external input and output operations whose effect can be an important part of what a programmer

[1] One possible way to include the latter issue in the notion of result is to assume that every program is augmented with reads of all memory locations after the original program is complete.

perceives as the outcome of an execution. We next consider the issues that arise when external devices are allowed to directly access main memory.

External Input and Output Operations

External input and output operations are commonly used to communicate with user input and output devices (e.g., keyboard, terminal) and other external devices such as disk or network controllers. Depending on the architecture, I/O operations are either encoded as special instructions or are generated by issuing ordinary read and write memory operations (typically uncached) to a predesignated address space (referred to as memory-mapped I/O). Due to the common use of memory-mapped I/O, external input and output operation are often referred to as I/O read and I/O write, respectively. The effect of an external output operation, or I/O write, is to change the state at its target external device. Similarly, an external input operation, or I/O read, returns a value representing a certain state at the target external device. In some cases, I/O reads may also modify state at the external device, thus functioning like a read-modify-write.

Even though I/O operations may appear similar to memory operations, their behavior and semantics can be quite different. *First*, the functionality achieved by an I/O operation can be quite complex. For example, an I/O write to a given device may enqueue its value at the tail of an internal queue on the device, while a subsequent I/O read dequeues and returns the value at the head of this queue. In other cases, an I/O read may return a value that is unrelated to previous I/O writes to the device (e.g., I/O read that returns the character typed at the keyboard). Therefore, optimizations that are common for memory operations, such as merging two writes to the same address into a single write or forwarding the value of a write to a later read to the same address, can easily violate the intended semantics for many I/O operations. *Second*, the order of I/O operations to different addresses can be important. For example, an I/O write to one address may potentially affect the value returned by an I/O read to a different address. Therefore, unlike ordinary memory operations, maintaining order among conflicting I/O operations may not be sufficient for guaranteeing correct behavior. *Third*, the programmer may be able to infer the actual execution order for certain I/O operations. In some cases, this order is directly observable in real time (e.g., print statements to a terminal). In other cases, it may be possible to reconstruct the order because of the semantics of the operations (e.g., consider I/O writes whose values are sequentially saved on disk). This further emphasizes the importance of avoiding optimizations such as merging operations to the same address and maintaining the appropriate order among I/O operations to different addresses.

The above issues affect the specification methodology for both uniprocessor and multiprocessor systems. Figure J.3 shows several examples to illustrate this. Examples (a)-(b) correspond to the uniprocessor case. Each I/O location is identified by an identifier and a device number. For example, io(A,dev1) refers to the location identified by A on device dev1. Figure J.3(a) illustrates a uniprocessor example with I/O operations to different locations on the same device. Maintaining the program order among these non-conflicting operations may be important; for example, the I/O write to A may affect the return value for the I/O read to B. Figure J.3(b) shows a couple of program segments with I/O operations to separate devices. The example on the left hand side shows two consecutive I/O writes to separate devices. Maintaining the order among such writes may be important; the writes may be to two output devices that are monitored by the user during the execution of the program. This example illustrates how uniprocessors may exhibit multiprocessor effects due to the fact that I/O operations are sometimes visible to the user. Maintaining order among I/O operations to

```
a1:  io(A,dev1) = 1;
b1:  u = io(B,dev1);
```

(a) non−conflicting operations

```
a1:  io(A,dev1) = 1;
b1:  io(B,dev2) = 1;
```

```
a1:  io(A,dev1) = 1;
b1:  while (io(Status,dev1) == 0);
c1:  io(B,dev2) = 1;
```

(b) operations to different devices

P1	P2
a1: A = 1;	*a2:* io(B,dev1) = 2;
b1: io(B,dev1) = 1;	*b2:* u = A;

(c) behavior in multiprocessors

Figure J.3: Examples illustrating I/O operations.

separate devices is typically more difficult and often less efficient than maintaining order among operations to the same device. Some systems may require an explicit mechanism for enforcing this order, such as reading the status from one device to ensure the completion of operations destined to it before issuing an operation to a different device. An example of this is shown in the right hand side of Figure J.3(b), where we assume a non-zero return value for the read of Status on the first device signifies that previously issued I/O operations have completed with respect to the device.

Finally, Figure J.3(c) shows an example of how I/O operations may influence the specification of multiprocessor memory behavior. The program segment shows P1 writing to A followed by an I/O write to an output device and P2 doing another I/O write to the device before reading A. Assume an execution where the user observes the I/O write from P1 before the I/O write from P2. Serial semantics will lead the user to expect the read of A to return the value of 1. However, this suggests that memory behavior depends on the order of I/O operations. Similarly, the behavior of I/O operations may depend on the order of memory operations; for example, memory operations may be used to synchronize access to a device by ordering I/O operations to that device from multiple processors.

To achieve higher performance, many designs, including uniprocessors, forgo serial semantics among I/O operations or between I/O and memory operations. The exact order maintained among I/O operations, and between I/O and memory operations, are often specified in an implementation-dependent manner. Similarly, mechanisms for enforcing extra orders are often implementation-dependent.

Memory Operations by External Devices

In addition to servicing I/O operations, some I/O devices have the ability to directly perform read and write operations to memory. For example, disk and network controllers may directly move data between physical

```
     Processor                              I/O device

a1:  A = 1;
b1:  B = 1;
       . . .
e1:  io(Ready,dev1) = 1;                    observe Ready set
                                            access memory
f1:  while (io(Done,dev1) == 0);            set Done
       . . .
```

Figure J.4: Synchronization between a processor and an I/O device.

memory and the disk or network. The ability to directly operate on memory introduces many processor-like behaviors for I/O devices which need to be modeled appropriately to capture the behavior of a realistic system.

Memory access by an I/O device may either be *synchronous* or *asynchronous* with respect to the program execution (analogous to exceptions versus interrupts). The scenario shown in Figure J.4 corresponds to a synchronous access since it is initiated by the program (e.g., as in initiating a write to disk). On the other hand, paging activity to disk for supporting virtual memory is an example of asynchronous access to memory since it is not explicitly initiated by the program. Regardless of whether the memory access by an I/O device occurs synchronously or asynchronously, such accesses are virtually always synchronized. That is, the processor and the I/O device coordinate their accesses to memory to avoid simultaneous conflicting operations. This synchronization may be in the form of either special I/O operations (e.g., I/O write to initiate a transfer to disk) or memory operations (e.g., network controller may set a flag in memory to signify completion of a transfer). Figure J.4 shows an example of such synchronization, where a processor signals the I/O device to access memory by performing an I/O write and waits for the device to signal completion by polling on a different I/O location. Even though the example corresponds to a uniprocessor system, the presence of the I/O device makes the behavior of the system multiprocessor-like since the program order relationship between memory operations and the I/O operations becomes significant.

One of the subtle issues that arises with respect to asynchronous memory operations by I/O devices is whether they should be included as part of the result of a program. For example, when a disk controller is invoked to read and transfer the contents of memory page as part of the operating system's paging activity, the values read represent a snapshot of the memory contents at a point in the execution. Such a snapshot of memory can unveil many of the reordering optimizations that take place even in uniprocessor systems (e.g., compiler optimizations such as code motion and register allocation). Therefore, considering such asynchronous snapshots of memory as part of the result would disallow many of the reordering optimizations if we want to provide sequential semantics. For this reason, such snapshots are often excluded from the definition of the result, especially since they are often not directly visible to the programmer.

J.3 Other Event Types

This section describes other event types, aside from I/O operations, that may need to be included in the specification of a real system, in addition to issues that arise with respect to each event.

P1		P2	
a1:	A = 1;	*a2:*	u = A;
b1:	B = 1;	*b2:*	v = B;

(a)

P1		P2	
a1:	A = 1;	*a2:*	u,v = A,B;
b1:	B = 1;		

(b)

Figure J.5: Multiple granularity access to memory.

Instruction Fetches

The first type of event is instruction fetch or read from the instruction space. Modeling instruction fetches is important if the system allows dynamic code modification. This issue arises in uniprocessor systems as well. Ideally, an instruction fetch should be simply treated as an extra memory read operation that is issued by the instruction. Therefore, maintaining sequential semantics would imply that the value returned by the read should correspond to the last write to the location in program order. However, most uniprocessor systems sacrifice the sequential semantics for read and write operations to the instruction space in order to achieve higher performance. Optimizations such as instruction prefetching and pipelining, separate instruction and data caches, issuing multiple instructions in the same cycle, and the out-of-order issue of instructions can lead to the possibility of temporarily stale copies of instructions being fetched and executed. Therefore, a write to the instruction space may not be reflected in instruction fetches that immediately follow it in program order.[2] As a result, the programmer is exposed to the non-sequential semantics. Ensuring correct behavior often requires adding a set of architecture or implementation-specific operations, such as a fence instruction, that delay future instruction fetches and flush stale instruction copies.

Atomicity Relation among Events

The framework presented in Chapter 4 deals with three atomic operations: a read, a write, and a read-modify-write to a single address, all with the same granularity of access to data. In general, a system may define a larger set of memory operations with varying access granularities and atomicity constraints. By far the most common feature is to support multiple granularity read and write operations. For example, the Alpha architecture supports atomic read and write operations to consecutive one, two, four, or eight byte regions [SW95]. We briefly discuss the issues in capturing the behavior of multiple granularity operations below.

Consider the two examples in Figure J.5. Assume operations in Figure J.5(a) are at a four byte granularity, and that locations A and B lie within an aligned eight byte boundary. Figure J.5(b) shows a similar code segment except the operations on P2 appear as an atomic eight byte read operation. Under sequential consistency, the outcome (u,v)=(0,1) is legal for the example on the left, but is not allowed for the example on the right since the read operations to A and B must appear atomic.

Our specification methodology is general enough to incorporate the type of multiple granularity operations discussed above. There are numerous ways in which this can be achieved; we briefly discuss one option

[2]Capturing this effect is simple within our framework. For an instruction fetch I, the initiation condition can be relaxed to not require $W_{init}(i) \xrightarrow{xo} I_{init}(i)$ given $W \xrightarrow{po} I$. In addition, uniprocessor dependence would not constrain $W(i) \xrightarrow{xo} I(i)$ in the above case. This allows the instruction fetch to return the value of an older write.

below. First, we need to introduce multiple granularity operations into the operation set. Sub-operations are still modeled at the lowest granularity. For example, assume a system with word and double-word operations. A double-word operation consists of two sets of sub-operations, one set for each word boundary. Second, we must extend the notion of conflict to larger granularity operations; two operations are considered to conflict if any of the constituent sub-operations conflict. Third, we must provide explicit conditions, similar to Condition 4.7, to capture the atomicity relation among the sub-operations of a larger granularity operation. As with our current specifications, chains such as the multiprocessor dependence chain are defined among conflicting operations while the implied constraints can be defined on conflicting sub-operations. Constraining the execution order only among conflicting sub-operations makes the specification more challenging. To capture the behavior of larger granularity operations, we may need to increase the number of chains in the specification. For example, consider the code segment in Figure J.5(b) under sequential consistency. Each of the word writes on P1 conflicts with the double-word read on P2. However, the writes on P1 do not conflict with one another. Therefore, we may require an explicit chain of the form W1 $\xrightarrow{po}$ W2 $\xrightarrow{co}$ R to disallow the outcome (u,v)=(0,1). This type of a chain is implicit in specifications with single granularity operations since W1 and W2 would have to be conflicting (i.e., given W1 conflicts with R and W2 conflicts with R) and the uniprocessor dependence and value conditions along with the transitivity of $\xrightarrow{xo}$ would end up appropriately constraining the execution order between the sub-operations of W1 and R.

Operations to Private Memory

The framework presented in Chapter 4 treats memory as a single shared address space. Nevertheless, some programming languages may allow the programmer to distinguish between shared and private memory, with private memory locations only accessible to a single process or thread. Because private memory locations are not read and written by multiple processes, operations to such locations need not obey the same strict ordering constraints as for shared memory locations. In fact, the necessary constraints are analogous to uniprocessor memory operations; conflicting operations from the same process must appear to execute in program order. Exploiting the distinction between private and shared memory operations can potentially lead to higher performance if (i) the relative frequency of private memory operations is significant, and (ii) the base memory consistency model imposes strict ordering constraints on the shared memory operations (e.g., a model such as SC). Furthermore, the fact that only a single processor needs to access a private memory location at any time may be useful for customizing the coherence protocol for private memory operations.

Consider the example in Figure J.6(a). By convention, private memory locations are appended with a “p” (e.g., pX). Assume a typical sequentially consistent system. Without distinguishing private and shared memory operations, the system would conservatively service one memory operation at a time. By distinguishing private memory operations, the system can treat such operations less conservatively by only maintaining the order among conflicting operations (i.e., between (b1) and (e1)). Furthermore, the program order between shared and private memory operations need not be maintained.

Even though a private memory location is accessed by a single process, process migration effectively allows more than one processor to access the location and may also lead to multiple copies of the location. To uphold the correct semantics for private memory locations, we need to ensure a consistent view of the private memory as processes are migrated. As we will see below, issues such as efficient and transparent support for process migration may reduce the incentive for distinguishing between private and shared memory operations.

```
     P1
a1:  A = 1;
b1:  pX = 1;
c1:  B = 1;
d1:  pY = 1;
e1:  u = pX;
```

(a) without migration

```
     P1                    P2
a1:  A = 1;
b1:  pX = 1;
c1:  B = 1;
d1:  pY = 1;               while (Flag ==0);
     <interrupt>       a2: u = pX;
     Flag = 1;
```

(b) with migration

Figure J.6: Interaction of private memory operations and process migration.

Figure J.6(b) shows the same program as in Figure J.6(a), except the process is migrated from P1 to P2 during its execution; this is shown as an interrupt on P1 to preempt the processor, along with a flag synchronization to signal the fact that the process is ready to move to P2. As a result of the migration, location pX is written by P1 and read by P2. Nevertheless, to provide transparent migration, the read to pX on P2 must still return the value of 1. A system that distinguishes between private and shared operations may require extra mechanisms to ensure transparent migration. For example, the private memory for a process has to be accessible from any processor, even though at any one time only a single processor will be accessing it. Similarly, migrating the process may result in multiple copies of private locations; these multiple copies must either be eliminated or must be kept coherent. One way to support process migration is to wait for all private memory operations to complete, flush any private memory locations associated from the process from the current processor's cache, migrate the private memory pages associated with the process to the target processor's node, and finally resume the process at the target process. Unfortunately, the cost of process migration can become quite large under such a scheme. At the other extreme, a consistent view is trivially guaranteed if we treat private memory operations the same as shared memory operations, and process migration can be supported much more efficiently in this way.[3] Therefore, there can be a trade-off between the performance gains from distinguishing between private and shared memory operations and the cost incurred for process migration.

Overall, most existing systems do not exploit the distinction between private and shared memory operations. Treating private and shared memory operations differently incurs hardware support and complexity. At the same time, the gains from making this distinction are less significant if the shared memory model is already relaxed since much of the reordering optimizations are already allowed for shared memory operations. Finally, making this distinction can potentially sacrifice efficient process migration.

Address Translation

Another event that is relevant to memory operations and instruction fetches is address translation. To support virtual memory, every instruction fetch and every memory operation can potentially require a translation of its address. The appropriate sequential semantics for address translation would be that any modification

[3] Section 5.7.1 in Chapter 5 describes the requirements with respect to shared memory operations for supporting transparent process migration.

to the translation should be reflected in subsequent translations. However, due to the various pipelining optimizations, even uniprocessor systems may not maintain this semantics and may require the use of special instructions to guarantee the serial behavior.

The interpretation of addresses is also important for determining whether two operations reference the same logical location (e.g., for determining whether two operations conflict). For virtual addresses, it is possible for two distinct addresses to refer to the same logical location. This sort of aliasing is not an issue with physical addresses. However, dealing with physical addresses requires including the behavior of I/O devices. For example, paging can cause a given logical location to be moved from one physical address to disk and back to a different physical address.

Exceptions and Other Miscalleneous Events

Another class of relevant events are exceptions. There are various types of exceptions, some related to integer and floating point operations and others related to memory operations. For example, an exception may occur during address translation if the translation is not readily available (e.g., TLB or page miss). Another class of exceptions related to memory operations are parity or ECC errors during cache or memory accesses. An important issue is whether the system provides support for precise exceptions. For efficiency reasons, most architectures do not provide precise exception semantics for every exception, e.g., an ECC error on write operations. Therefore, the programmer may again be exposed to the non-sequential behavior of such events even in uniprocessors.

There are other miscalleneous events that may also be of interest in other types of systems. For example, in a system that provides a stable storage facility, it may be important to know when write operations are committed to memory as opposed to simply guaranteeing the appropriate order among writes. Therefore, it may be important to model the actual committing of the write to memory.

J.4 Incorporating various Events into Abstraction and Specification

In addition to memory reads and writes issued by the processors, we have discussed a variety of other events that may need to be incorporated in a complete specification of a system. Below, we briefly enumerate the types of changes that are required for incorporating the larger set of events within our abstraction and specification methodology.

First, the abstractions for a shared-memory system and a shared-memory program (Definitions 4.1 and 4.2) need to be generalized to incorporate the various events. We also need to provide an abstraction for each type of event, similar to the abstraction presented for memory reads and writes in Section 4.1.3. For example, the abstraction for instruction fetches or operations to private memory may be identical to that of memory operations. We can use a similar abstraction to model memory operations issued by I/O devices. On the other hand, I/O operations issued by the processor may require a different abstraction. The issue of multiple copies is not present for I/O operations, allowing us to model I/O writes as a single as opposed to multiple sub-operations. In addition, the semantics for I/O operations is different from typical read and write memory operations. Finally, events such as address translations or exceptions also need to be modeled.

Second, the notion of result needs to be clearly defined and needs to incorporate all events of interest. For example, the effect of I/O operations needs to be incorporated in the notion of result. Furthermore, the manner in which events are incorporated into the notion of result determines the flexibility in constraining orders among such events for maintaining equivalent results.

Third, we need to generalize the components of an execution (e.g., Definition 4.11) to include the various events and to define the relevant relations such as program order and execution order among them. The execution order relation needs to be defined as a total order on all events (for example, ordering memory operations with respect to other memory or I/O operations). Similarly, the program order relation needs to be defined among the relevant events, again possibly relating events of different types with respect to one another.

Finally, the conditions that impose constraints on the execution order need to be extended to impose the appropriate orders among the various events. As discussed above, the execution order becomes a total order on all events. To maintain aggressive implementations, it is important to constrain execution order among as few events as possible. For most specifications, it should be possible to constrain the execution order among events of the same type, thus avoiding execution order constraints on events of different types. For memory operations, we can continue to constrain execution order among conflicting operations only, unless the notion of result is radically changed. For I/O operations, however, ordering constraints may in general be required for operations to different locations due to the non-memory-like semantics. Furthermore, chains that capture the order among events need to be generalized to include different types of events. For example, memory operations may be used to order I/O operations (e.g., to synchronize access to an I/O device) and I/O operations may be used to order memory operations (e.g., for synchronizing accesses to memory by I/O devices).

J.5 A More Realistic Notion for Result

The notion of result can be made more realistic once I/O devices are fully incorporated into the abstraction and specification of a model. Programmers may observe the effects of some I/O operations, such as output to a terminal. In addition, some of the values returned by memory reads issued by I/O devices may be observable by the programmer; for example, an I/O device like the disk controller may read various regions of memory and save the contents on disk for later retrieval. In many cases, the above two things form a complete notion of the result for the programmer. Therefore, one possibility is to define the notion of result as consisting of the sequence of certain I/O operations issued by the processors plus a subset of the return values for memory reads issued by the I/O devices.

J.6 Summary on Extensions to Framework

This appendix explored the issues that arise in modeling a more general set of events in addition to read and write memory operations. Many of the issues discussed are not particular to multiprocessors and apply to uniprocessors as well. Furthermore, even in the context of uniprocessors, maintaining sequential semantics for some types of events (e.g., I/O operations) is sacrificed for purposes of efficiency. In addition, the behavior of the more general set of events is often not precisely defined. Fortunately, only a few programmers need to deal with the semantics of the more general set of events. While it is possible to incorporate various events into our specification methodology, the remaining question is what the ordering semantics should be on such events; this is closely related to the notion of result that is assumed by programmers.

Appendix K

Subtle Issues in Implementing Cache Coherence Protocols

Designing a correct and efficient cache coherence protocol for a large scale shared-memory system can be reasonably challenging. A number of cache coherence protocol designs have been described in previous publications (e.g., [Len92, CKA91]). This appendix briefly reviews some of the more subtle aspects in designing such protocols, primarily based on our experience with the DASH [LLG+90] and FLASH [KOH+94] protocols.

One of the big challenges is to implement an efficient cache coherence protocol while still maintaining correctness. From a performance point of view, it is important to reduce the latency and number of messages associated with servicing each memory operation in addition to minimizing serializations in the protocol. Furthermore, since most systems are built around commercial processors, the design of the cache coherence protocol among different processors is typically influenced by protocol decisions that have been made for maintaining coherence within the processor cache hierarchy. The difficulties on the correctness side arise from two sources: (a) the distribution of the protocol state information makes it practically impossible to atomically access and modify the global state associated with a given memory line, and (b) the presence of multiple paths within the network leads to minimal message ordering guarantees. In addition, the protocol must ensure deadlock-free and livelock-free operation (most protocols do not attempt to alleviate starvation).

This appendix begins by briefly discussing protocol deadlock issues. The remaining parts of the appendix primarily focus on interactions of operations to the *same cache line*. We provide several examples of transient or corner cases that arise due to the distributed nature of the protocol. In addition, we describe issues related to serializing simultaneous operations to the same line, cross checking messages on incoming and outgoing paths, and maintaining point-to-point ordering among specific messages.

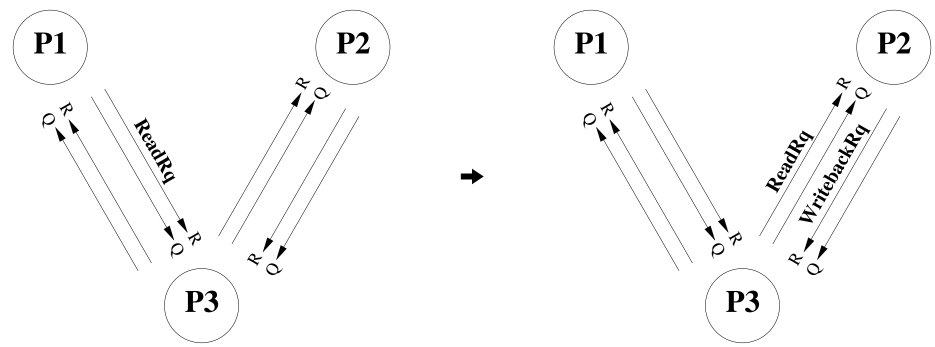


Figure K.1: A transient write-back scenario.

K.1 Dealing with Protocol Deadlock

Protocol deadlock may arise from circular dependencies among resources, and such dependencies can arise at many levels within a cache coherence protocol. A simple example of a circular dependences involves two processors issuing requests to each other's caches; deadlock can arise if each cache waits for its reply before servicing its requests. As we discussed in Section 5.2, the buffering and handling of messages within the cache hierarchy and in the network is another potential source of circular dependences. A common solution to this problem is to separate request and reply messages and to use logically different paths (either separate virtual or separate virtual lanes) for transferring such messages within the network and cache hierarchy. A consequence of this solution is that even point-to-point ordering is not guaranteed among all message types.

K.2 Examples of Transient or Corner Cases

The distribution of state information and the lack of order among coherence messages introduce a number of subtle transient cases that can occur under normal operation and must be correctly handled by the protocol. Transient conditions typically arise because multiple processors simultaneously act on a given address and these action do not take effect atomically with respect to the global state of the line; therefore, different processors may observe the line in different states at the same time. This section describes some classic examples of such transient behavior.

The first example, shown in Figure K.1, illustrates the effect of the distributed state information. The figure shows three processing nodes along with the logical request and reply network paths denoted as "R" and "Q", respectively. Consider a line with a home at P_3 that is initially dirty in P_2's cache. Assume P_1 issues a read request to this line, which is sent to the home node and is then forwarded to P_2. Meanwhile, assume P_2 replaces the dirty line (leading to a write-back message destined to the home) while P_1's request is in transit towards P_2. The read request and the write-back request travel in opposite directions in the network and can easily pass one another without detection (cross checking messages in the network is impractical). Therefore, when the read request reaches P_1's cache hierarchy, it finds that the line is no longer available. The above anomaly arises due a window of time when the global state of the line can change while the read request is in transit towards P_2. There is a related window of time while the write-back request is in transit that the directory information is no longer correct.

One solution to anomalies similar to the above is to generate a *negative acknowledgement* (nack) reply

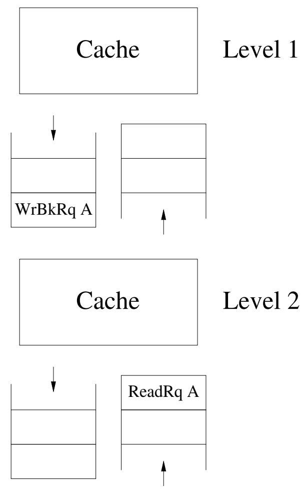


Figure K.2: Messages bypassing one another in a cache hierarchy.

as a response to forwarded requests that do not find the line in the appropriate state at the target cache. The nack reply can be used to signal the requesting processor that the given request must be reissued, which implies a mechanism for retrying requests from each node. Nack replies can also play an important role in deadlock avoidance solutions (based on separating requests and replies) by eliminating certain request-request dependences. For example, a request that generates additional request messages (e.g., a read request that is received by the home and forwarded to another node) can simply be nacked if the outgoing request buffer is temporarily full.

Transient cases similar to the one shown in Figure K.1 can also occur within a cache hierarchy. Figure K.2 shows one such scenario where an incoming read request (from a different processor) bypasses a write-back request for the same line. In contrast to the network, it is possible to detect these types of bypasses within a cache hierarchy through some form of cross checking between the incoming and the outgoing paths. By detecting this case, the cache hierarchy can take appropriate action by grabbing the data from the write-back to form a reply for the read. There are several ways to achieve the cross check. The brute force way is to use associative lookup on the queues. Another approach involves appropriately flushing a set of queues before servicing requests from other queues. The effect of cross checks may also be achieved by exploiting the cache states. For example, if an incoming read requests arrives at the level 2 cache first, it can appropriately change the state of the line (e.g., from dirty-dirty, signifying that the hierarchy has a dirty copy and the actual dirty copy is at a higher level, to clean-dirty) such that the outgoing write-back is automatically turned into an outgoing read reply by this cache (ordinarily, the write-back expects a dirty-dirty state). This last solution does not apply to the lowest level buffers in the hierarchy (i.e., past the lowest level cache). Of course, the simplest option is to use the same solution that is used by the network: allow the messages to bypass one another and simply nack the read if it does not find the line in the appropriate state. However, since most bus-based systems do not support nacks, commercial processors typically provide some form of cross checking functionality.

Figure K.3 shows another example of a transient case where the processor receives an invalidate request while its read request is pending to the same line. Assume the home for the line is at P_2. The sequence in the figure shows P_1's read request is satisfied by P_2 and a reply is sent back to P_1. Meanwhile, assume P_2 issues

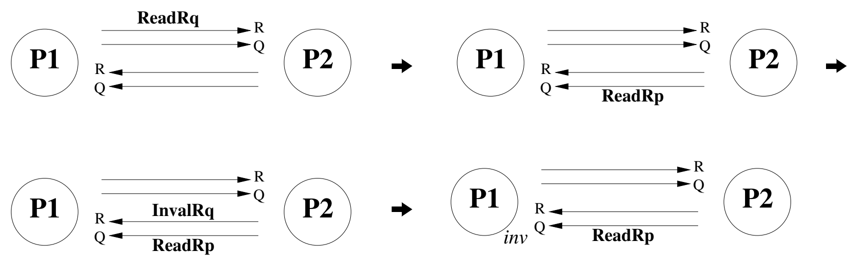


Figure K.3: Example of a transient invalidate from a later write.

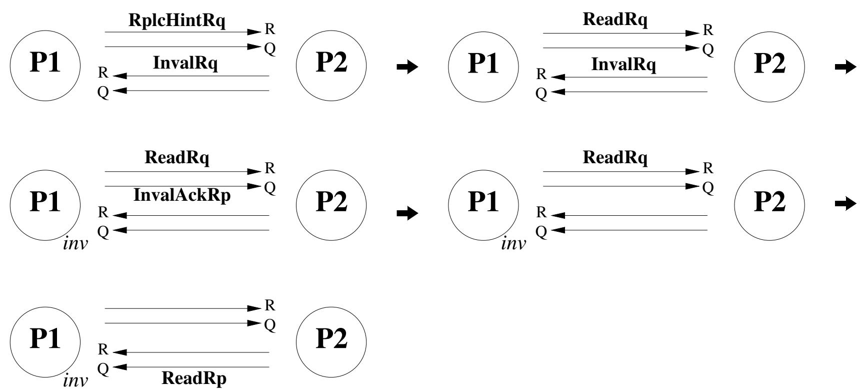


Figure K.4: Example of a transient invalidate from an earlier write.

a write to the same address which results in an invalidate request to be sent to P_1. As shown in the figure, the invalidate can bypass the read reply because they use different paths. Therefore, P_1 observes an invalidate for a line it is waiting upon. From a correctness point of view, P_1 must avoid caching the data returned by the reply since it is already stale. The value of the read can be provided to the processor as long as it is not cached; alternatively, the read can be reissued.

Figure K.4 shows a similar scenario to the above, except the invalidate request is actually from an earlier write (i.e., a write that is serialized before the read value is bound). Assume P_2 is the home for the line and that P_1 is initially caching a copy. The figure shows P_1 replacing the line, leading to a replacement-hint request (replacement of clean lines may be silent in some protocols). Meanwhile, P_2 writes to the line, and since the directory still shows P_1 as a sharer, an invalidate request is sent to it. The sequence in the figure shows P_1 attempting to read the line before the invalidate reaches at. As shown, the invalidate can actually be acknowledged back to the home before this read reaches the home due to the separate request/reply paths. The read eventually gets serviced and actually carries the most up-to-date data back to P_1. However, in most protocols, P_1 cannot distinguish this scenario from the scenario in Figure K.3; in both cases, P_1 simply receives an invalidate to a line while its read is pending. Therefore, the implementation must conservatively treat the invalidate as possibly belonging to a later write, which means that it cannot cache the returning reply.[1]

The above two transient cases arise primarily due to a lack of point-to-point order between requests and replies. However, even if the network provides point-to-point order among all messages, similar transient

[1] Analogous transient scenarios can arise with update requests to pending reads. Similar to the invalidation case, the implementation cannot distinguish whether the update belongs to an earlier or later write relative to the read reply (see Figure K.8.

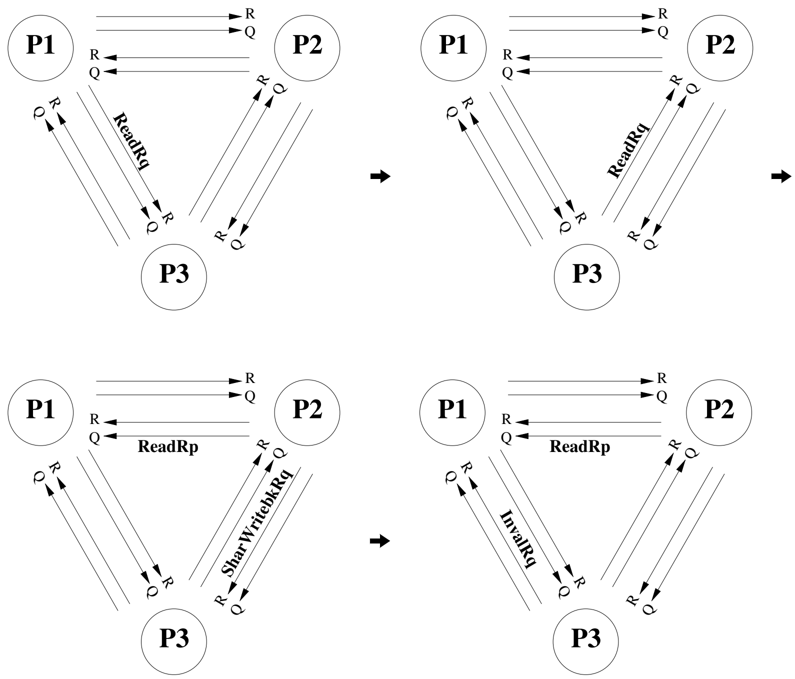


Figure K.5: Example of a transient invalidate from a later write in a 3-hop exchange.

conditions can arise when more than two nodes are involved. Figure K.5 shows one such scenario. Assume P_3 is the home node for the line, and P_2 initially has a dirty copy. The sequence shows P_1 making a read request, which is forwarded from the home to P_2. The read reply is shown to be directly sent from P_2 to P_1, along with a sharing write-back message that informs the directory about the new sharer and write backs the up-to-date data to memory; both the DASH and the FLASH protocols use this forwarding optimization. However, assume that P_3 issues a write to the line before the reply returns to P_1. This scenario is analogous to the example shown in Figure K.3 where an invalidate from a later write is received by the requesting node while it has a pending read. In this case, the problem arises due to the forwarding optimization which leads a different point-to-point path for the reply (P_2 to P_1) relative to the path for the invalidate request from the home (P_3 to P_1).

Figure K.6 illustrates another subtle transient scenario where an invalidate request arrives while a processor has an outstanding exclusive request to the same line. Assume P_2 is the home node and P_1 initially holds a clean copy of the data. Assume P_1 and P_2 both attempt to write the line, leading to an exclusive request from P_1 to the home, and an invalidate request from P_2 to P_1. The figure shows P_2's invalidate being acknowledged before P_1's original exclusive request gets to P_2 (another example of request-reply bypassing). There are three possible solutions for servicing the exclusive request. All three solutions require the directory at the home to detect that P_1 no longer maintains a valid clean copy of the line; this can be easily determined by examining the list of sharers at the home. The first solution involves responding to P_1 with a read-exclusive reply (instead of the expected exclusive reply) and modifying the directory to point to P_1 as the new exclusive owner; this implies that the implementation can robustly accept a read-exclusive reply to an exclusive request (even when an invalidate request arrives in between). The second solution is to respond with a negative acknowledgement reply, which does not change the state of the directory and forces P_1 to retry the request;

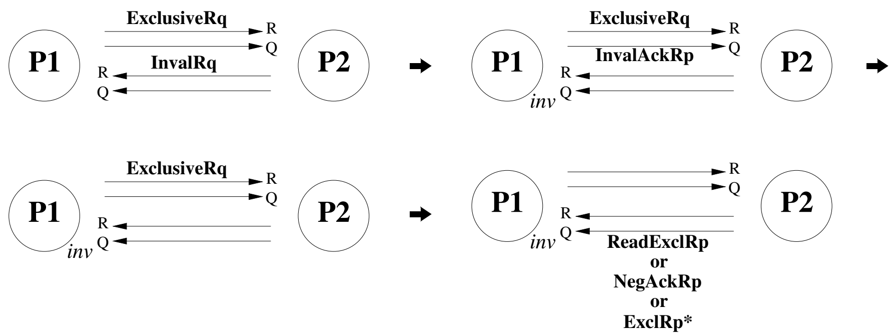


Figure K.6: Example of a transient invalidate with a pending exclusive request.

the retry from P_1 will lead to a read-exclusive request since it no longer has a clean copy. Finally, the third solution is to speculatively respond with an exclusive reply before fully checking the sharer list. P_1 must still drop the exclusive reply because it received an invalidate while its exclusive request was outstanding and no longer has the up-to-date data for the line; similarly, the directory must check the list and not modify the ownership information. The motivation for the last solution is to reduce the latency of an exclusive request in the common case (i.e., no transient situation), especially if checking the directory takes a relatively significant time. The FLASH protocol uses the third solution;[2] such scenarios do not arise in the DASH protocol because of the lack of support for exclusive requests (i.e., the protocol always generates a read-exclusive).

Overall, even though a protocol must correctly handle transient cases, the occurrence of such scenarios is typically infrequent. A necessary prerequisite is for multiple cache hierarchies to simultaneously act upon the same line. Furthermore, some of the scenarios arise only if messages experience substantially different delays within the network or associated network queues. Given the low frequency of transient cases, the efficiency of handling transient cases is not important as long as the adopted solutions do not degrade performance for the common coherence operations.

K.3 Serializing Simultaneous Operations

Serializing simultaneous operations on the same line is a powerful design strategy for eliminating a large number of transient cases in a protocol. Section 5.3.5 already describes the benefits of disallowing new requests from being serviced while a line has pending invalidations or updates; detection of write completion and support for category three dependence chains are both simplified. As discussed in Section 5.3.5, the functionality of disallowing new requests can be supported at either the requester or the home depending on which is responsible for gathering the invalidation acknowledgements. This section describes a number of other scenarios where such serializations can greatly simplify the design of the protocol.

Figure K.7 illustrates an example where allowing a write to proceed while the previous write's invalidations are pending can lead to subtle problems. Assume P_2 is the home node and that P_1 initially has a clean copy of the line, and assume P_1 and P_2 simultaneously attempt to write the line. The scenario in the figure shows P_2 reaching the directory first and generating an invalidation request to P_1 while P_1 has an exclusive request

[2]Due to the fact that the sharers are represented as a linked list in FLASH, the protocol also speculatively sends invalidations to sharers before having checked the whole list to see whether the exclusive reply will fail. Exclusive requests due to a store-conditional use the second solution above, however, both to provide correctness and to avoid livelocks (see Section 5.3.7 and Appendix N).

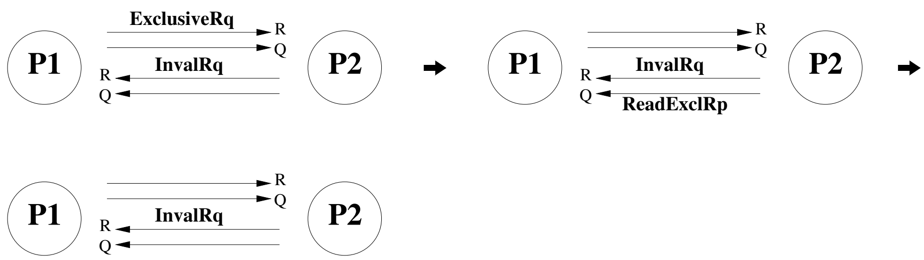


Figure K.7: Example with simultaneous write operations.

in transit. Assume the directory actually services the exclusive request from P_1 before the invalidate request from the previous write is acknowledged. As mentioned in the previous section, the directory can detect that P_1 is no longer on the sharing list and responds with a read-exclusive reply. It is now possible for this reply to reach P_1 before the original invalidation request due to the lack of point-to-point order between requests and replies. Therefore, P_1 may later receive an invalidation request to a dirty line, which can lead to anomalous behavior in most protocols. A simple solution for alleviating this behavior is to disallow any new requests to the line while there are outstanding invalidations for the line.

Figure K.8 shows that disallowing a new read while there are outstanding updates from previous writes can simplify the design of update-based protocols. Again, assume P_2 is the home for the line. Figure K.8(a) shows a scenario where P_1 replaces a line and then reads the line while there is an outstanding update request destined for it. As shown, the read reply can reach P_1 before the update request. Even though the read reply provides P_1 with the most up-to-date data, the value can be made incorrect if P_1 performs the update from the earlier write when it later receives the request. The difficulty arises from the fact that P_1 cannot simply drop the update request since it cannot distinguish whether the update is due to an earlier or later write relative to its cached value. Figure K.8(b) shows the latter case, where the update that belongs to a later write (e.g., from P_2) must actually be performed by P_1. Note that this problem is easier to solve with an invalidation-based scheme because the line can be conservatively invalidated even if the invalidate request belongs to an earlier write; therefore, the invalidate can be accepted in both cases. However, with an update-based scheme, it is incorrect to uniformly accept or reject the update request under *both* scenarios. Disallowing a future read from proceeding until the updates from previous writes are acknowledged turns out to solve this problem by eliminating the possibility of receiving an update from an earlier write after the processor receives the read reply.[3]

A larger number of simplifications are possible by further serializing operations to the same location by disallowing a new request to be serviced either (a) while there are pending invalidations and updates from previous writes to the line (described above), or (b) while a previous request that is forwarded from the home (to the current owner) is not yet satisfied. The FLASH protocol achieves the above serialization by maintaining a *pending* bit per memory line (as part of the directory entry) that determines whether a line is in one of the above two states;[4] new read, read-exclusive, and exclusive requests to a line in pending state are

[3] This solution is described in Appendix M; the other solution described there is to enforce point-to-point order between incoming read replies and update requests, for example, by placing both of them on the same network path. It turns out that the DASH protocol does not use either solution, and therefore fails to enforce the coherence requirement if update writes are used. This is the only known error in the protocol. The error probably arose because the protocol was only tested for invalidation writes; it was uncovered through the reasoning presented in Chapter 5.

[4] The pending bit is reset when all invalidations or updates are acknowledged to the home, or when a sharing write-back or ownership change request arrives at the home in response to a forwarded request.

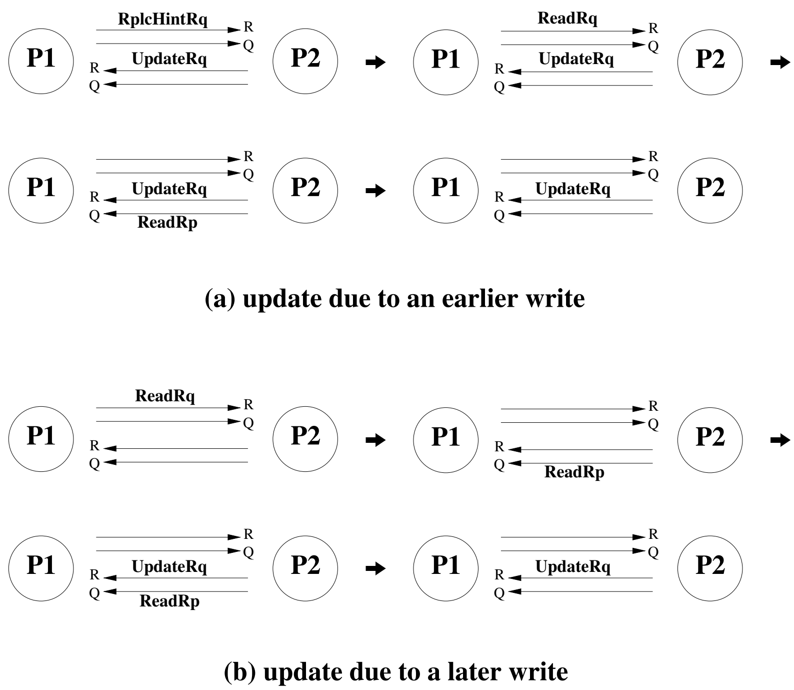


Figure K.8: Example transient problems specific to update-based protocols.

forced to retry. This type of serialization makes the FLASH protocol significantly simpler than the DASH protocol. Figure K.9 shows a scenario that illustrates the type of complexity present in DASH. Assume P_3 is the home node and P_1 is the initial owner. The sequence in the figure shows P_2 grabbing the line next; the reply is directly forwarded from P_1 and an ownership change request is sent to the home to signal the new owner. Meanwhile, P_3 also writes to the location, generating a read-exclusive request to P_1 before the ownership change reaches the home. The sequence shows P_1 grabbing the line again, and finally the request from P_3 succeeds in grabbing the line. As shown, this interaction can lead to two ownership change requests to be simultaneously present in the network; the protocol will fail to properly keep track of the current owner if the ownership change request from P_1 reaches the home first. To avoid this race, the DASH protocol has an extra message that is sent by the home to the requester of a forwarded message to signal the fact that the ownership change request has reached the home; before then, the requester disallows other processors (e.g., P_3 in the example) from accessing the line and also avoids writing back the line. In FLASH, this problem is trivially solved by disallowing multiple operations on the line as described above.[5]

K.4 Cross Checking between Incoming and Outgoing Messages

Cross checks are typically done to detect messages to the same line that bypass one another on the incoming and outgoing paths. As we mentioned before, most types of cross checking can be virtually alleviated by depending on negative acknowledgement (nack) messages and simply retrying a request that fails to find the data. However, bus-based designs typically do not support this type of mechanism and therefore depend

[5]For correctness, an ownership change request that arrives at the home checks to make sure the line is not yet written back by the new owner before changing the directory state to point to the new owner; otherwise, the ownership change request is simply ignored.

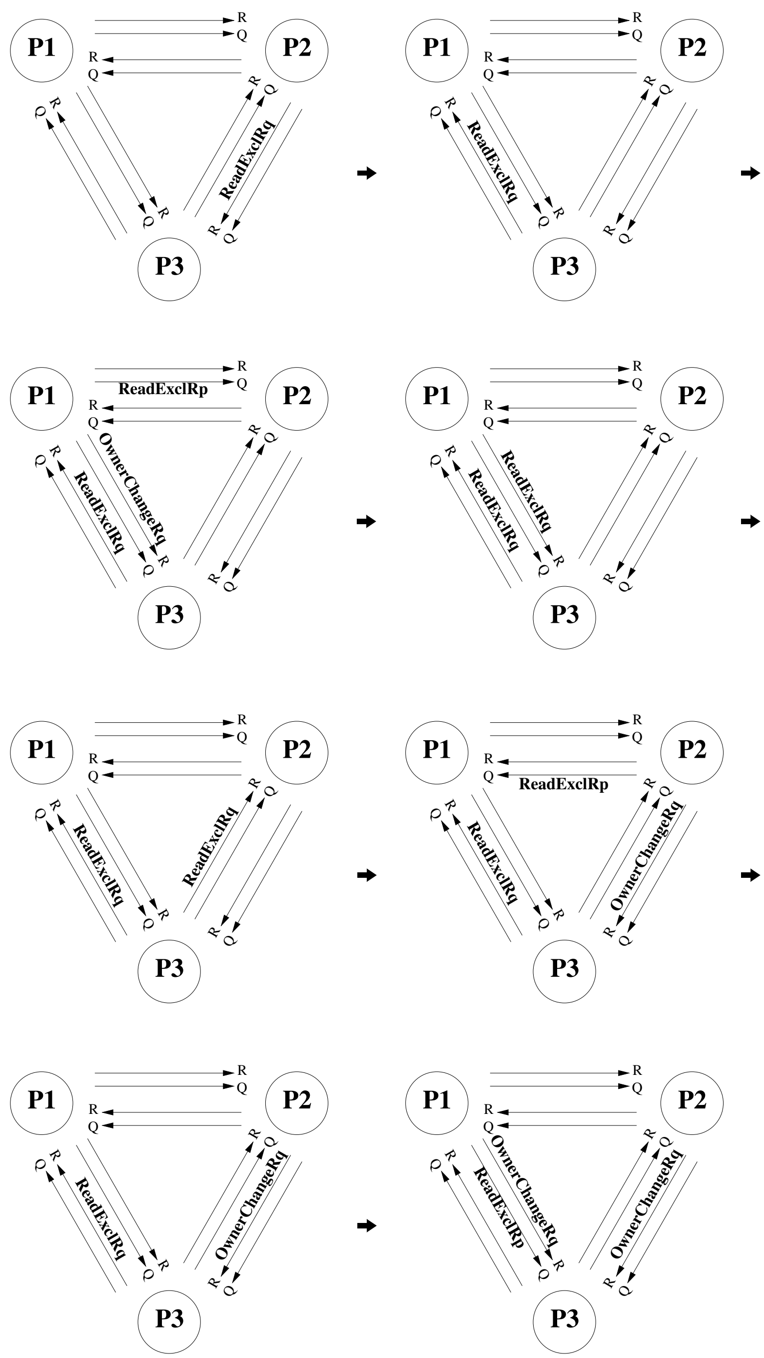


Figure K.9: Complexity arising from multiple operations forwarded to an exclusive copy.

on the appropriate cross checks to be done within the cache hierarchy. Referring back to Figure K.2, the scenario is a classic example where cross checking is important if the implementation does not support nacks (incoming read or read-exclusive request with outgoing write-back request).

As an example of a cross check that may be important for correctness (i.e., even if the protocol supports negative acknowledgements), consider an outgoing exclusive request and an incoming invalidate request within a cache hierarchy. For correctness, most bus-based protocols depend on this interaction to be detected and for the exclusive request to be transformed to a read-exclusive request which fetches an up-to-date copy of the line in addition to obtaining ownership. Architectures that support store-conditionals (see Section 5.3.7) depend on a similar cross check between an outgoing exclusive request generated by the store-conditional and an incoming invalidate or update request; to provide correctness and to avoid livelock, the store-conditional must be forced to fail in such a case without invalidating or updating other processor's copies.

The cross checks described in the previous paragraph involving exclusive requests are actually important for correctness in snoopy protocols since without detecting them, the protocol can fail to appropriately serialize simultaneous writes to the same line (or support the atomic semantics for store-conditionals). In contrast, directory-based protocols typically have sufficient information at the directory to detect such races without any cross checking; this was described earlier in the context of Figure K.6. Therefore, by using negative acknowledgement messages and choosing the appropriate solutions for dealing with exclusive requests, a directory-based protocol can actually eliminate the need for all types of cross checks.[6]

K.5 Importance of Point-to-Point Orders

Even though the network and cache hierarchy provide only a few message ordering guarantees, maintaining some of these orders may be important for providing simple and efficient implementations of some protocols. This section is mainly concerned with point-to-point ordering guarantees among messages to the *same* address. The major trade-off here is between further relaxing the ordering guarantees in the network and cache hierarchy to achieve better message performance versus more aggressively exploiting the stricter ordering guarantees to achieve simpler and more efficient protocols.

Consider the following example of how point-to-point ordering among messages can simplify the design of a cache coherence protocol. Assume a processor replaces a clean line, generating a replacement-hint request to the home, and next issues a read request to the same line. The ordering of these two requests is important in most protocols since a reordering would lead the directory to incorrectly discard the processor from the sharing list while the processor maintains a copy as a result of the read reply. The design of the protocol can actually become complicated if the cache hierarchy and network do not provide the above ordering guarantee. This is because it is difficult to provide additional state at the directory, other than using sequence numbers or time stamps to reconstruct the message order, to distinguish whether the replacement-hint actually occurs before or after the read request. The other option is to provide an acknowledgement reply for the replacement-hint

[6] Of course, choosing the third solution described in the context of Figure K.6 (i.e., speculatively responding to the exclusive request) still requires the cross check on the requester side. One solution is to place a detection table that keeps track of outstanding requests right past the lowest level cache in the hierarchy. By ensuring the cache and this table are accessed atomically by incoming and outgoing requests, the case where the invalidate request is received right after the exclusive request is sent off can be correctly detected. Note that if the invalidate request arrives earlier, it will invalidate the copy at the lowest level cache, and the exclusive request can detect this and turn into a read-exclusive request (or fail) since the hierarchy no longer has a clean copy. The same detection table can be extended to also detect the types of transients described in the previous sections (e.g., an invalidate occurring while a read request is outstanding).

request and delay read requests from the processor until the replacement-hint is acknowledged. Both options are likely to make the protocol less efficient. Other similar ordering guarantees may also be important: for example, the order from an outgoing replacement-hint request followed by a read-exclusive request, or the order from an outgoing write-back (or sharing write-back, or ownership change) request followed by a read or read-exclusive request. The directory state in most protocols may have sufficient state to allow the protocol to robustly deal with the latter types of reordering. For example, to handle the reordering with a write-back request, a read or read-exclusive request can be simply nacked if the directory state shows that the requesting processor still maintains a dirty copy of the line. Nevertheless, the extra checking of state that is required may still increase latencies and occupancies in the protocol. Finally, Appendix M and Section 5.3.3 describe the importance of several point-to-point orders for update protocols.

Most designs maintain a higher degree of ordering within the cache hierarchy as compared to the network. For example, the order between an incoming read reply and a later incoming invalidate request within the cache hierarchy is important for avoiding stale copies (the reverse order is not critical since reordering will lead to conservatively invalidating the line). Note that orders such as the above are not maintained in the network if the request and reply use different paths; this leads to transient cases such as the one depicted in Figure K.3 that must be handled properly external to the cache hierarchy. Further point-to-point orders may also be maintained within a cache hierarchy in designs that do not support nacks. For example, the order from an incoming read or read-exclusive request to an earlier incoming exclusive or read-exclusive reply must be maintained to make sure the request will find the cache line in the appropriate state.

Overall, the degree to which a protocol depends on (or exploits) point-to-point order within the network and cache hierarchy varies among different protocols. The amount of changes required in a protocol to correctly deal with more relaxed point-to-point orders can therefore vary greatly across different protocols.

Appendix L

Benefits of Speculative Execution

This appendix provides a few examples of how speculative execution helps in aggressively exploiting a relaxed memory model (see Section 5.2 for the general discussion on this).

Consider the example program segment in Figure L.1(a). Without speculative execution, the processor must first resolve the conditional clause which involves stalling for the return value for the read of A before issuing the operations within the conditional statement. Speculative execution along with branch prediction enable the processor to fetch and issue instructions from the predicted path before the branch condition is actually resolved. Therefore, with correct branch prediction, the processor has the opportunity to overlap the read of A with useful computation and memory operations within the conditional.

Speculative execution typically requires a rollback mechanism in the case of an incorrect branch prediction. The rollback mechanism logically discards the results of speculative instructions that appear after the incorrectly predicted branch and restarts the processor by issuing new instructions from the correct path. Most instructions can be executed speculatively, though the result of the instruction is not logically committed until previous branches are known to be correctly predicted. Therefore, memory read operations can be speculatively issued to the memory system. However, memory write operations are typically not issued speculatively to the memory system since rolling back would be too difficult. Nevertheless, the write may be issued internal to the processor and later read operations to the same address can speculatively return the value written by the write. For example, referring back to Figure L.1(a), the read of B can be speculatively issued to the memory system, the write to C can be issued internally, the read of C can speculatively return the new value for C, and finally the value of this read may be used to speculatively execute the following add operation, all before the read of A is complete.

Even though we discussed speculative execution in the context of conditional branches, there are a number of other scenarios where speculative execution can increase overlap. For example, processors that support precise exceptions can benefit from speculative execution. Consider the program segment in Figure L.1(b). The example shows three consecutive reads, with the return value for the first read determining the address of the second read. Assume the three reads are long latency operations, and that any read can result in an exception

```
a1: r1 = A;
b1: if (r1 == 0) {
c1:     r2 = B;
d1:     C = 1;
e1:     r3 = C;
f1:     r5 = r3 + r4;
g1: }
```

(a)

```
a1: r1 = A;
b1: r2 = B[r1];
c1: r3 = C;
```

(b)

Figure L.1: Example program segments for illustrating speculative execution.

such as a page fault. To provide precise exceptions, a processor without support for speculative execution cannot issue the read of C until the address for the second read is resolved and checked for exceptions, thus eliminating overlap between the reads of A and C. Speculative execution allows the processor to issue the read of C early by assuming (or predicting) that previous unresolved addresses will not lead to an exception, and using rollback if an exception does occur (analogous to branch misprediction).

Appendix M

Supporting Value Condition and Coherence Requirement with Updates

Section 5.3.2 describes the support for the value condition and coherence requirement in invalidation-based designs. This appendix considers implementations with update-based schemes, assuming a single level write through cache per processor. We consider two implementations similar to those described for the invalidation case.

In the *first* implementation, the write remains in the write buffer until the processor is notified that the write is serialized with respect to other writes to the same location. The update reply from the home signals the fact that the write has been serialized. The write can be retired from the write buffer when this reply arrives at the issuing processor. Update requests to stale cache copies emanate from the home node, thus serializing updates to a given location. Incoming requests to the cache are handled in the usual way while there is a write miss outstanding.

The ordering between an incoming update reply and an incoming update request (to the same address) sent from the home to a target cache is important for maintaining coherence. Specifically, if an update request from a write that is serialized before (after) this processor's write arrives after (before) the update reply, then the order in which the writes complete with respect to this processor will not be the same as the order in which they are serialized. The required order can be maintained in networks that support point-to-point order by sending both types of message on the same lane; for example, the update reply may be placed on the lane that is typically used for requests.[1] Similarly, for cache hierarchies that provide multiple incoming paths, both messages can be sent on the same path; for example, the update request may be placed on the path that is typically used for replies.[2] Furthermore, designs with multiple level cache hierarchies may require the

[1] This turns out to be safe from a deadlock avoidance perspective; the original update request can simply be nack'ed through the reply lane in case the request lane is full.

[2] The incoming update request is acknowledged as soon as it arrives at the cache hierarchy in virtually all practical cache designs (see Section 5.4.1). Therefore, no reply needs to be generated for the update request after it enters the cache hierarchy. This allows the update request to be sent on the same path that is used by incoming replies (see deadlock avoidance discussion in Section 5.2.3).

incoming update reply to carry back the data from the original update request so that copies in the lower level caches can be updated on the incoming path; updating the copies on the incoming instead of the outgoing path simplifies the task of performing updates in the same order they are serialized by the home.

Another important order is between an incoming read reply and an incoming update request (to the same address) from the home to a given cache copy. This order can be maintained within the network and the cache hierarchy using similar techniques to the ones discussed above. Ordering these messages within the network may be unnecessary in protocols that disallow the home from servicing a read request while the line has outstanding updates that have not been acknowledged.[3] Finally, protocols that allow multiple update requests to the same address to be in transit from the home to a target cache must ensure the serialization order is maintained among the update requests sent to each cache.[4] Even if the protocol disallows multiple update requests within the network from the home to a target cache (e.g., by disallowing an update request to be sent while there are update requests from a previous write that have not been acknowledged), it may still be necessary to maintain the order among incoming update requests to the same address within the cache hierarchy in designs with early update acknowledgements (see Section 5.4.1).

The *second* update-based implementation, that retires the write into the cache before it is serialized with respect to other writes, is slightly trickier. Assume the smallest granularity for an update is a word. While an outgoing update request is pending (i.e., not serialized), the cache must ignore incoming update requests to the same word. This is simple to achieve with a bit per word as long as the cache only allows a single outstanding update per word in the line. Caches that allow multiple outstanding updates per word require a more sophisticated mechanism for determining when to stop ignoring incoming update requests. For example, more than a single bit may be associated with each word to keep a count of the outstanding updates to that word.[5] This count is incremented on an outgoing update request and decremented when the reply that signals the serialization of the update returns. A count of greater than zero is analogous to the valid bit being set, and incoming updates to that word are simply ignored. When the count reaches its maximum, no more updates to the same word can be accepted from the processor. As discussed Section 5.3.3, allowing multiple outgoing updates (from the same processor) to the same word can also make it difficult to satisfy the uniprocessor dependence condition especially if such requests can be forced to retry (e.g., through a negative acknowledgement reply by the home).

[3] It is still necessary to note the reception of an incoming update request while the read request is pending. Such updates must either be applied to the read reply data before it is cached, or the data must be dropped without being caching.

[4] As shown in Section 5.3.5, the gathering of update acknowledgements can also become more tricky in this case especially if the acknowledgements are gathered at the home node.

[5] As with the valid bits discussed in Section 5.2.3, these counts are only needed for cache lines with outstanding requests and not all cache lines.

Appendix N

Subtle Implementation Issues for Load-Locked and Store-Conditional Instructions

This appendix describes some of the more challenging issues in implementing the load-locked and store-conditional instructions described in Section 5.3.7. Much of the complexity involves alleviating livelock conditions whereby no process can successfully complete a load-locked/store-conditional sequence. The first requirement for avoiding livelock is to disallow unnecessary replacements of the cache line corresponding to the lock-address which could in turn lead to the clearing of the lock-flag. Such replacements can be induced by either instruction or data fetches in the middle of the sequence. Solutions for avoiding replacements due to instruction fetches include using split instruction and data caches, using two-way set associativity within a unified cache (imposes the constraint that the sequence is only guaranteed to succeed on the fall through path if there are any branches), or disallowing instruction fetches from displacing the lock-address line. A common solution for avoiding replacements due to other read and write memory operations is to disallow the use of such operations in between the load-locked and the store-condition sequence. Instruction prefetching, speculative execution, and out-of-order issue cause further complexity since instruction and data fetches for instructions from outside the sequence may also lead to the replacement of the line.

The second requirement for avoiding livelock is to avoid unnecessary invalidation or update messages that cause other processors' store-conditionals to fail. For example, a failing store-conditional or a load-locked operation should not cause invalidations or updates that would lead to the clearance of another processor's lock-flag. This latter requirement requires special support within a cache coherence protocol since a store conditional cannot be treated in the same way as an ordinary write operation; the protocol must ensure that a store-conditional fails, without sending any coherence messages to other copies, if another write beats it to the serialization point. Ensuring this requirement in a bus-based scheme involves forcing the store-conditional to fail if the cache no longer has a valid copy of the line or if cross checking the incoming messages at the external interface indicates a match with an invalidation request; both conditions above indicate that another write may have beaten the store-conditional to the serialization point. A directory-based protocol with a general network makes satisfying the above requirement more challenging since the simple cross-checking is no longer sufficient (or necessary). In an invalidation-based scheme, the exclusive request resulting from a store-conditional must check the list of sharers at the home directory to see whether the requesting processor is still on the list. Failure to find the requester on the list indicates that another write successfully reached the home before the store-conditional and the store-conditional must be forced to fail without generating any invalidations. An update-based scheme is even more difficult to support since the state at the directory cannot be easily used to tell whether another write operation has beaten the store-conditional to the serialization point. Here is one possible solution. The store conditional request can lock down the line at the home to disallow any other requests and proceed to send a message back to the requesting processor (point-to-point order with other incoming updates is important). On receiving the message, the processor checks the lock-flag. If the lock-flag is still set (indicating no updates were received), an update request is sent to the home, the line is released at the home, and the update is forwarded to all sharers. Otherwise, the store-conditional is forced to fail and a message is sent back to the home to release the line.

Appendix O

Early Acknowledgement of Invalidation and Update Requests

This appendix provides several examples, and further intuition, for the first technique described in Section 5.4.1 for dealing with early acknowledgements. This techniques guarantees correctness by imposing specific orders among incoming messages.

Assume the technique described above with an underlying invalidation-based protocol. Figure O.1 shows an example of a category three multiprocessor dependence chain (same chain as in Figure 5.14(d)). Figure O.1(a) and (b) show the ordering among events in designs without and with early acknowledgements, respectively. Assume the first solution described in Section 5.3.5 for satisfying the third category of chains, whereby the write must complete or commit with respect to all processors before another processor is allowed to read the new value. For the conservative design, the write of A is clearly guaranteed to complete with respect to P3 before the read of A. For the aggressive design, the ordering shows that the write of A is committed with respect to P3 before the read of A completes on P3. However, we need to show that the write of A also completes with respect to P3 before the read of A (i.e., the dashed arrow). Consider the commit event for the write of B with respect to P3 which occurs before the read of B on P3, implying that P3 receives an incoming reply to the read sometime after the write of B commits with respect to it. We also know that the write of A is committed with respect to P3 before the write of B commits. The above two observations imply that the write of A completes with respect to P3 before the read of B, and therefore also before the read of A.

Figure O.2 shows the same chain as above except with the second solution (described in Section 5.3.5) for satisfying such chains, whereby certain program ordering constraints from a read to a following operation are enforced by not only waiting for the read to complete but also waiting for the write that caused the value of the read to complete with respect to every processor (for simplicity, we assume it is complete with respect to the issuing processor as well). The figure shows the read of A on P2 with two events, one of them subscripted with “g” to denote the fact that the write which provided the value for the read is “globally” committed or performed. The reasoning for how the chain is maintained with early acknowledgements is quite similar to

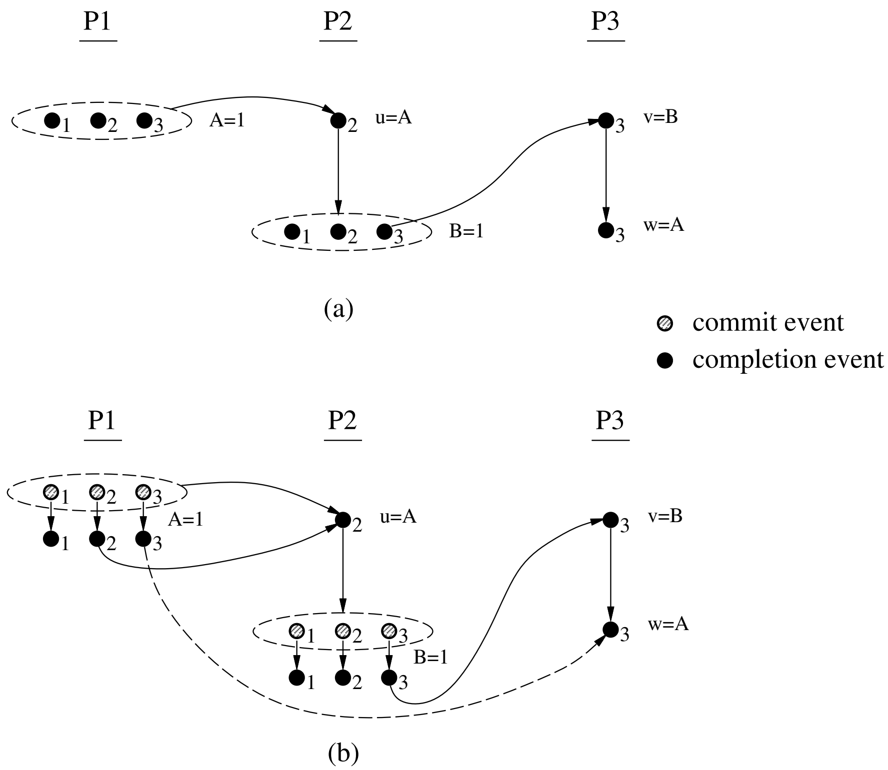


Figure O.1: Reasoning with a category three multiprocessor dependence chain.

the reasoning for the first solution depicted in Figure O.1.

Figure O.3 shows how orders are carried along a chain across conflicting pairs of operations in the middle of a chain.[1] The cases for W $\xrightarrow{co}$ R and W $\xrightarrow{co}$ W are pretty simple. Any write that is committed with respect to P_j before the pivot event shown for each case can be shown to also have completed with respect to P_j before the operation that follows (in program order) the conflict order. As previously seen in Figure 5.23, the R $\xrightarrow{co}$ W conflict order is a little bit more tricky. Figures O.3(c) and (d) show two separate cases where the previous operation in program order is either a read or a write. Figure O.3(c) shows a miss operation for the read on P_i which is analogous to the write miss operation described in Section 5.4.1 (in the context of Figure 5.23). The arrow labeled (II) exists because otherwise R2 $\xrightarrow{co'}$ W could not hold (same reasoning as in Figure 5.23). In addition, the dashed arrow labeled (I) shows that if a write completes with respect to P_i before R1, then the atomicity relationship between R1 and its miss operation implies that the write must complete before the miss operation as well. The arrows labeled (I) and (II) in Figure O.3(d) are analogous to the above. The dashed arrow coming into the miss operation on P_i is used for reasoning with chains that have consecutive occurrences of R $\xrightarrow{co'}$ W.

Figure O.4 illustrates the atomicity properties of miss operations with respect to other operations to the same line. Figure O.4(a) shows the atomicity with respect to read miss operations, while Figure O.4(b) shows it for write miss operations (the dashed lines are implied by the atomicity property). To ensure the atomicity

[1] The R $\xrightarrow{co}$ W $\xrightarrow{co}$ R conflict order is not shown in the figure since it can be treated as a combination of the R $\xrightarrow{co}$ W and W $\xrightarrow{co}$ R cases, with the assumption that the W $\xrightarrow{co}$ R order is upheld using one of the techniques described in Section 5.3.5 for maintaining multiple copy atomicity for writes.

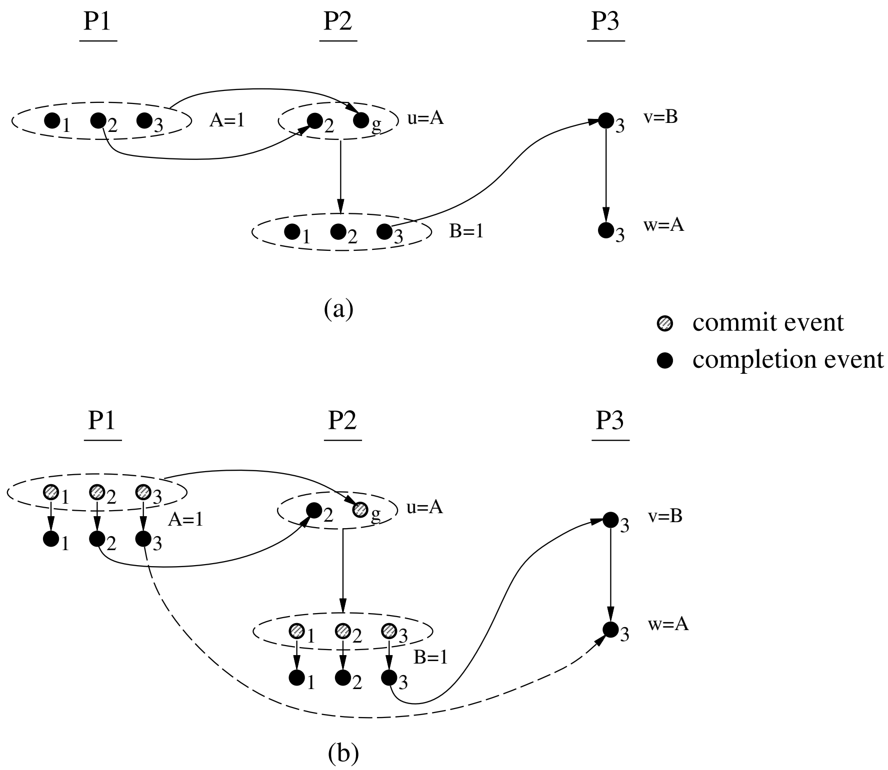


Figure O.2: Another design choice for ensuring a category three multiprocessor dependence chain.

property shown for W $\xrightarrow{co}$ R, a cache with a dirty exclusive copy must change the state of the line after providing the value to an incoming read request (even if the return value is not cached by the other processor). For simplicity, the W1 $\xrightarrow{co}$ W2 is shown assuming a protocol that disallows a write to be serviced until the previous write to that line is committed with respect to all processors.

Figure O.5 shows a more complex example with a category three chain comprising of R $\xrightarrow{co}$ W conflict orders. Figure O.5(a) and (b) show the execution orders without and with early invalidation acknowledgements, respectively. Again, assume the first solution for satisfying the third category of chains, whereby the write must complete or commit with respect to all processors before another processor is allowed to read the new value. The reasoning for the aggressive case is similar to that in Figure 5.23.

Section 5.4.1 also describes a second technique for dealing with early acknowledgements that maintains correctness by servicing incoming messages as part of the enforcing of program orders. As mentioned there, reasoning about the correctness of the second technique is simpler than reasoning about the first technique. The example in Figure 5.25 in Chapter 5 was used to illustrate this point. Figure O.6 shows another example of the simpler reasoning with the second technique for the same chain shown in Figure O.5 with the first technique. The observations made in the context of the example in Figure 5.25 hold here as well.

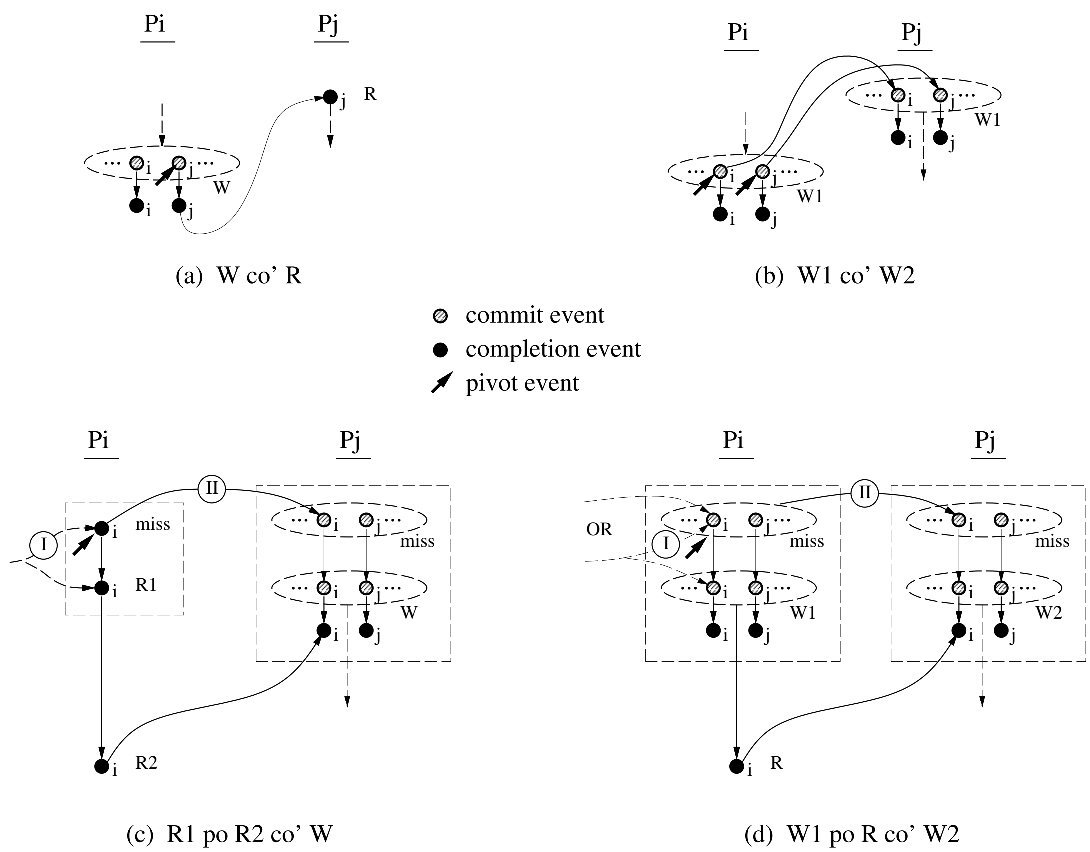


Figure O.3: Carrying orders along a multiprocessor dependence chain.

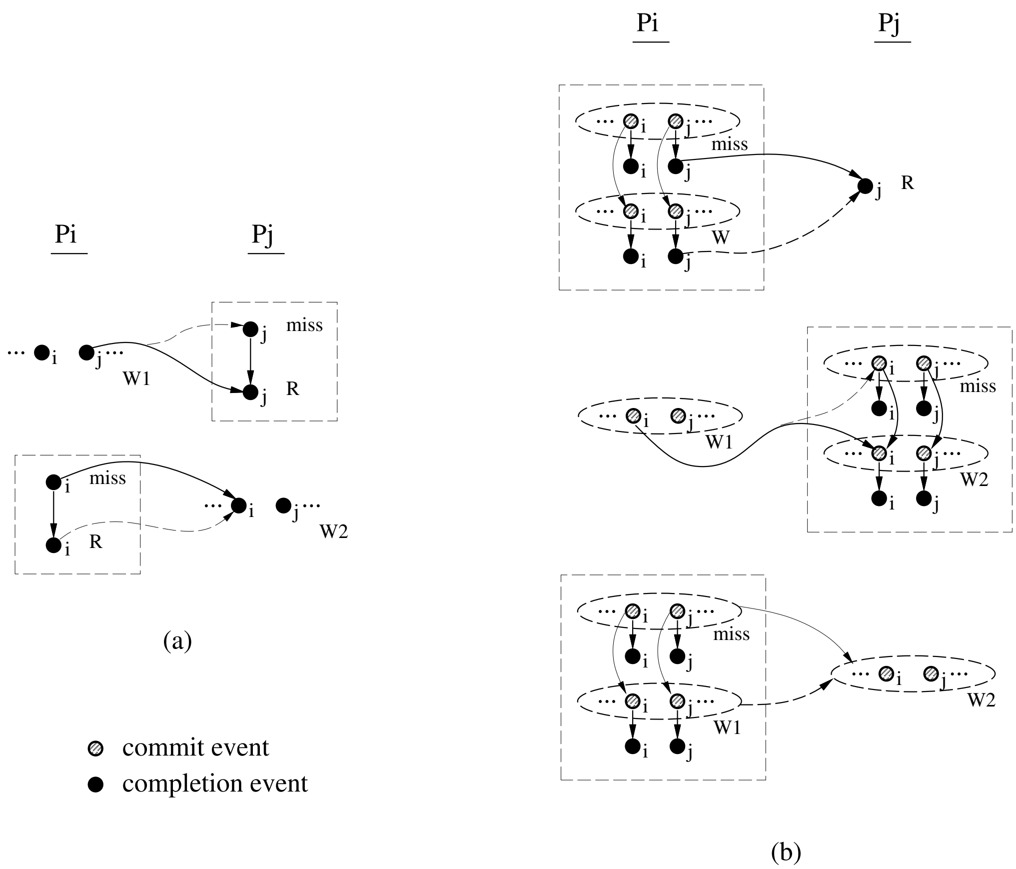


Figure O.4: Atomicity properties for miss operations.

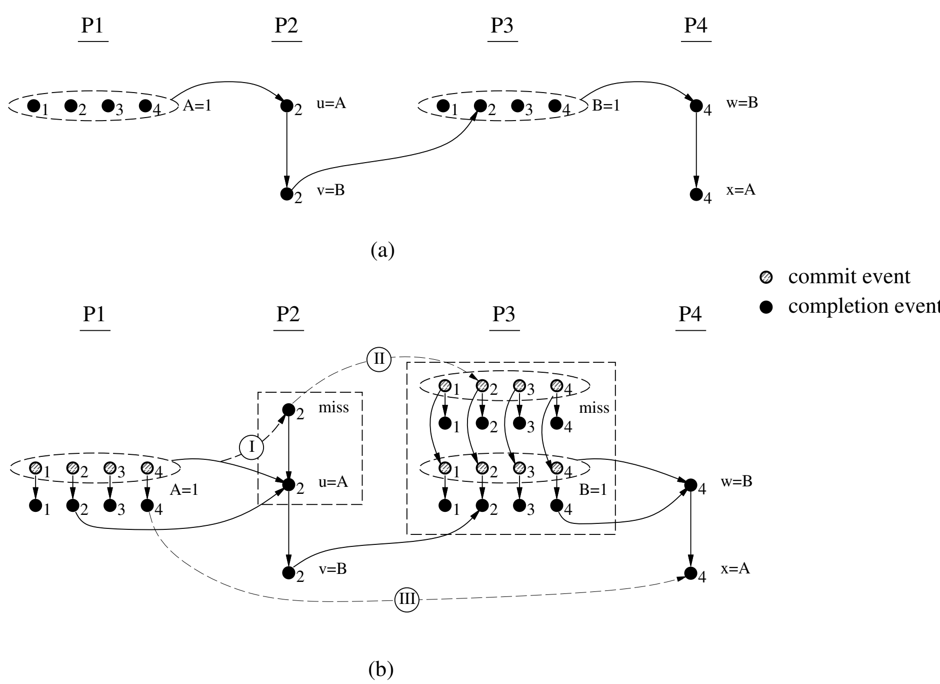


Figure O.5: Category three multiprocessor dependence chain with R $\xrightarrow{co}$ W.

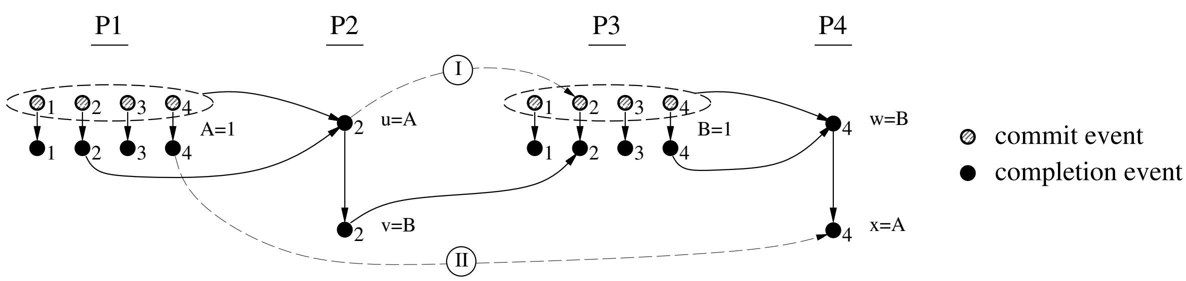


Figure O.6: Category three multiprocessor dependence chain with R $\xrightarrow{co}$ W.

Appendix P

Implementation of Speculative Execution for Reads

This appendix presents an example implementation of the speculation execution technique for reads described in Section 5.4.3. The latter part of the appendix illustrates the execution of a simple code segment with speculative reads.

P.1 Example Implementation of Speculative Execution

This section describes an example implementation of the speculative technique. As with the prefetching technique described in Section 5.4.2, the speculative technique benefits from the lookahead in the instruction stream provided by dynamically scheduled processors with branch prediction capability. In addition, the correction mechanism used by the branch prediction machinery can be easily extended to handle correction for speculative read operations. Although such processors are complex, incorporating speculative execution for read operations into the design is simple and does not significantly add to the complexity. The section begins with a description of the dynamically scheduled processor that is used as the base of the example implementation, followed by a discussion of implementation details for speculative read operations.

The organization for the base processor is directly derived from a study by Johnson [Joh89, Joh91]. Figure P.1 shows the overall structure of the processor. We describe this organization in Section 6.4 of Chapter 6 in the context of architectures with non-blocking reads. As discussed there, the key component in this design is the *reorder buffer* which is responsible for several functions: eliminating storage conflicts through register renaming, allowing conditional branches to be speculatively bypassed, and providing precise interrupts. We will discuss the effect of requiring precise interrupts on the implementation of consistency models later in the section.

To implement the speculative read technique, only the load/store (memory) unit of the base processor needs to be modified and the rest of the components remain virtually unchanged. Figure P.2 shows the

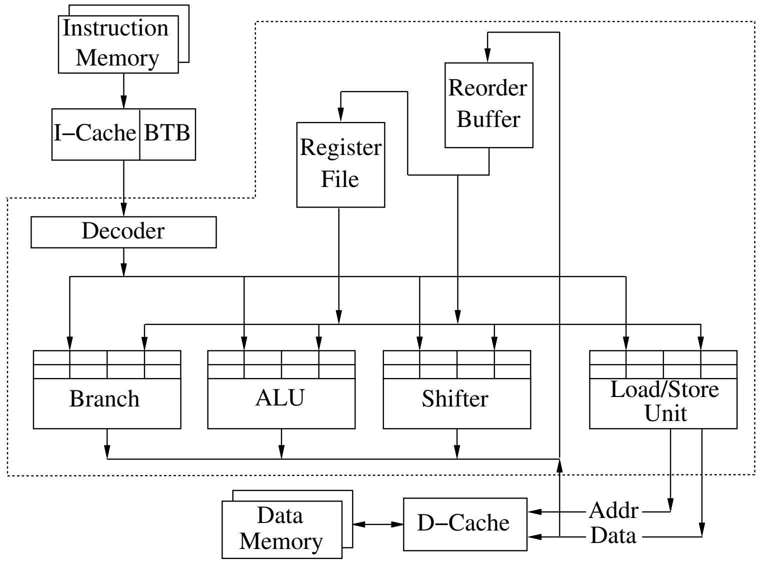


Figure P.1: Overall structure of Johnson's dynamically scheduled processor.

components of the memory unit. We first describe the components shown on the left side of the figure. These components are present regardless of whether speculative reads are supported. The only new component that is required for supporting speculative reads is the *speculative-load buffer* that will be described later.

The *load/store reservation station* holds decoded load and store instructions in program order. These instructions are retired to the address unit in a FIFO manner. Since the effective address for the memory instruction may depend on an unresolved operand, it is possible that the address for the instruction at the head of the reservation station is not yet computed. The retiring of instructions is stalled until the effective address for the instruction at the head can be computed.[1] The *address unit* is responsible for computing the effective address and doing the virtual to physical translation. Once the physical address is obtained, the address and data (if it is already computed) for store operations are placed into the *store buffer*. The retiring of writes from the store buffer is done in a FIFO manner and is controlled by the reorder buffer to ensure precise interrupts (the mechanism is explained in the next paragraph). Read operations are allowed to bypass the store buffer and dependence checking is done on the store buffer to ensure a correct return value for the read. Although the above implementation is sufficient for a uniprocessor, we need to add mechanisms to enforce consistency constraints for a multiprocessor.

First consider how sequential consistency can be supported. The conventional method is to delay the completion of each operation until its previous operation is complete. Delaying the write for previous operations is aided by the fact that writes are already withheld to provide precise interrupts. The mechanism is as follows. All uncommitted instructions are allocated a location in the reorder buffer and are retired in program order. Except for a write instruction, an instruction at the head of the reorder buffer is retired when it completes. For write instructions, the write is retired from the reorder buffer as soon as the address translation is done. The reorder buffer controls the store buffer by signaling when it is safe to issue the write to the memory system. This signal is given when the write reaches the head of the reorder buffer. Consequently,

[1] We could use a more aggressive technique for speculatively issuing reads that follow a write with an unresolved address as was discussed in Section 5.4.3. However, the implementation described here does not exploit this optimization.

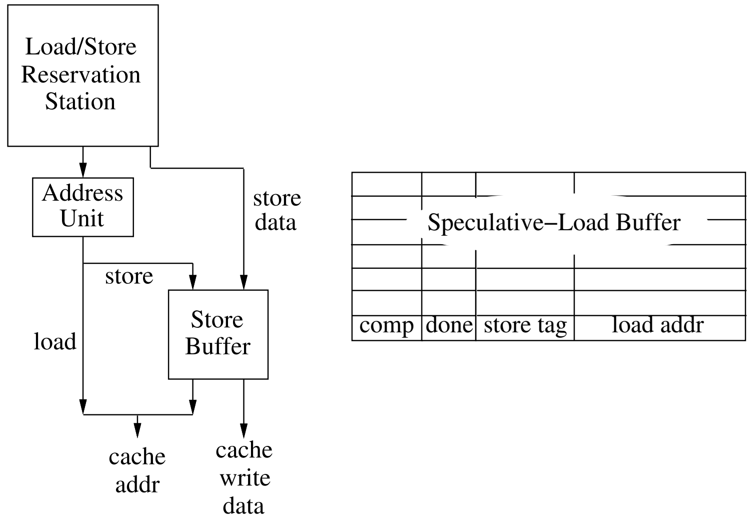


Figure P.2: Organization of the load/store functional unit.

a write is not issued until all previous loads and computation are complete. This mechanism satisfies the requirements placed by the SC model on a write operation with respect to previous reads. To make the implementation simpler for SC, we change the policy for retiring writes such that the write at the head of the reorder buffer is not retired until it completes also (for PL1, however, the write at the head is still retired as soon as address translation is done). Thus, under SC, the write is also delayed for previous writes to complete and the store buffer ends up issuing writes one-at-a-time.

We now turn to how the restrictions on read operations are satisfied. First, we discuss the requirements assuming the speculative read mechanism is not used. For SC, it is sufficient to delay a read until previous reads and writes have completed. This can be done by stalling the load/store reservation station on reads until the previous read is performed and the store buffer empties.

For speculative execution of read operations, the mechanism for satisfying the restrictions on reads is changed. The major component for supporting the mechanism is the speculative-load buffer. The reservation station is no longer responsible for delaying certain read operations to satisfy consistency constraints. A read is issued as soon as its effective address is computed. The speculation mechanism comprises of issuing the read as soon as possible and using the speculated result when it returns.

The speculative-load buffer provides the detection mechanism by signaling when the speculated result is incorrect. The buffer works as follows. Reads that are retired from the reservation station are put into the buffer in addition to being issued to the memory system. There are four fields per entry (as shown in Figure P.2): load address, comp, done, and store tag. The load address field holds the physical address for the read. The comp field is set if the read is considered a competing operation. For SC, all reads are treated as competing. The done field is set when the read completes. If the consistency constraints require the read to be delayed for a previous write, the store tag uniquely identifies that write. A null store tag specifies that the read depends on no previous writes. When a write completes, its corresponding tag in the speculative-load buffer is nullified if present. Entries are retired in a FIFO manner. Two conditions need to be satisfied before an entry at the head of the buffer is retired. First, the store tag field should equal null. Second, the done field should be set if the comp field is set. Therefore, for SC, an entry remains in the buffer until all previous read and write operations complete and the read operation it refers to completes. The original paper describes how an atomic read-modify-write can be incorporated in the above implementation [GGH91b].

We now describe the detection mechanism. The following coherence transactions are monitored by the speculative-load buffer: incoming invalidation (or read-exclusive) requests, incoming updates (or update-read) requests, and cache line replacements.[2] The load addresses in the buffer are associatively checked for a match with the address of such transactions.[3] Multiple matches are possible. We assume the match closest to the head of the buffer is reported. A match in the buffer for an address that is being invalidated or updated signals the possibility of an incorrect speculation. A match for an address that is being replaced signifies that future coherence transactions for that address will not be sent to the processor. In either case, the speculated value for the read is assumed to be incorrect.

Guaranteeing the constraints for PL1 can be done in a similar way to SC. The conventional way to support PL1 is to delay a competing write until its previous operations complete, to delay operations following a competing read until the read completes, and to delay a competing read for a previous competing write. First consider delays for writes. The mechanism that provides precise interrupts by holding back write operations in the store buffer is sufficient for guaranteeing that writes are delayed for the previous competing read, even though it is stricter than what PL1 requires. The same mechanism also guarantees that a competing write is delayed for previous read operations. To guarantee a competing write is also delayed for previous write operations, the store buffer delays the issue of the competing write operation until all previously issued writes are complete. In contrast to SC, however, non-competing writes are issued in a pipelined manner.

Let us consider the restriction on read operations under PL1. The conventional method involves delaying a read operation until the previous competing read operation is complete. This can be done by stalling the load/store reservation station after a competing read until the read completes. However, the reservation station need not be stalled if we use the speculative read technique. Similar to the implementation of SC, reads are issued as soon as the address is known and the speculative-load buffer is responsible for detecting incorrect values. The speculative-load buffer description given for SC applies for PL1. The only difference is that the comp field is only set for read operations that are considered competing under PL1. Therefore, for PL1, an entry remains in the speculative-load buffer until all previous competing reads are completed. Furthermore, a competing read entry remains in the buffer until it completes also and its previous competing write completes (achieved by appropriately setting the store tag field). The detection mechanism described for SC remains unchanged.

When the speculative-load buffer signals an incorrect speculated value, all computation that depends on that value needs to be discarded and repeated. There are two cases to consider. The first case is that the coherence transaction (invalidation, update, or replacement) arrives after the speculative read has completed (i.e., done field is set). In this case, the speculated value may have been used by the instructions following the read. We conservatively assume that all instructions past the read instruction depend on the value of the read and the mechanism for handling branch misprediction is used to treat the read operation as "mispredicted". Thus, the reorder buffer discards the read and the instructions following it and the instructions are fetched and executed again. The second case occurs if the coherence transaction arrives before the speculative read has

[2] A replacement is required if the processor accesses an address that maps onto a cache line with valid data for a different address. To avoid deadlock, a replacement request to a line with an outstanding operation is delayed until the operation completes.

[3] It is possible to ignore the entry at the head of the buffer if the store tag is null. The null store tag for the head entry signifies that all previous operations that are required to complete have completed and the memory model constraints would have allowed the operation to perform at such a time.

completed (i.e., done field is not set). In this case, only the speculative read needs to be reissued, since the instructions following it have not yet used an incorrect value. This can be done by simply reissuing the read operation and does not require the instructions following the load to be discarded.[4] The next section further illustrates speculative reads by stepping through the execution of a simple code segment.

P.2 Illustrative Example for Speculative Reads

This section steps through the execution of the code segment shown at the top of Figure P.3 assuming the sequential consistency model. Both the speculative technique for reads and the prefetch technique for writes are employed. Figure P.3 also shows the contents of several of the buffers during the execution. We show the detection and correction mechanism in action by assuming that the speculated value for location D (originally in the cache) is later invalidated.

The instructions are assumed to have already been decoded and placed in the reorder buffer. In addition, the load/store reservation station is assumed to have issued the operations already. The first event shows that both the reads and the exclusive prefetches for the writes have been issued. The store buffer is buffering the two write operations and the speculative-load buffer has entries for the three reads. Note that the speculated value for read D has already been consumed by the processor. The second event occurs when ownership arrives for location B (assume delayed exclusive replies). The completion of write B is delayed by the reorder buffer, however, since there is an uncommitted instruction ahead of the store (this observes precise interrupts). Event 3 signifies the arrival of the value for location A. The entry for read A is removed from the reorder buffer and the speculative-load buffer since the operation has completed. Once read A completes, the store buffer is signaled by the reorder buffer to allow write B to proceed. Since location B is now cached in exclusive mode, write B completes quickly (event 4). Thus, write C reaches the head of the reorder buffer. The store buffer is signaled in turn to allow write C to issue and the access is merged with the previous exclusive prefetch request for the location.

At this point, assume an incoming invalidation request arrives for location D. Since there is a match for this location in the speculation buffer and since the speculated value is used, the read D instruction and the following read instruction are discarded (event 5). Event 6 shows that these two instructions are fetched again and a speculative read is issued to location D. The read is still speculative since the previous store (store C) has not completed yet. Event 7 shows the arrival of the new value for location D. Since the value for D is known now, the read operation E[D] can be issued. Note that although the operation to D has completed, the entry remains in the reorder buffer since write C is still pending. Once the ownership for location C arrives (event 8), write C completes and is retired form the reorder buffer and the store buffer. In addition, read D is no longer considered a speculative load and is retired from both the reorder and the speculative-load buffers. The execution completes when the value for E[D] arrives (event 9).

[4]To handle this case properly, reply values must be tagged to distinguish between the initial reply, which has not yet reached the processor and needs to be discarded, and the reply from the repeated access, which is the one to be used. In addition, in case there are multiple matches for the address in the speculative-load buffer, we have to guarantee that initial replies are not used by any of the corresponding reads.

Example code segment:

read A	(miss)
write B	(miss)
write C	(miss)
read D	(hit)
read E[D]	(miss)

Event	**Reorder Buffer**	**Store Buffer**	**Speculative−Load Buffer**	**Cache Contents**
1 reads are issued and writes are prefetched	ld E[D] ld D st C st B ld A	st C st B	comp done st tag ld addr ✓ ld E[D] ✓ ✓ st C ld D ✓ ld A	A: ld pending B: ex−prf pending C: ex−prf pending D: valid E[D]: ld pending
2 ownership for B arrives	ld E[D] ld D st C st B ld A	st C st B	comp done st tag ld addr ✓ ld E[D] ✓ ✓ st C ld D ✓ ld A	A: ld pending B: valid exclusive C: ex−prf pending D: valid E[D]: ld pending
3 value for A arrives	ld E[D] ld D st C st B	st C st B	comp done st tag ld addr ✓ ld E[D] ✓ ✓ st C ld D	A,D: valid B: valid exclusive C: ex−prf pending E[D]: ld pending
4 write to B completes	ld E[D] ld D st C	st C	comp done st tag ld addr ✓ ld E[D] ✓ ✓ st C ld D	A,D: valid B: valid exclusive C: ex−prf pending E[D]: ld pending
5 invalidation for D arrives	st C	st C		A: valid B: valid exclusive C: ex−prf pending D: invalid
6 read of D is reissued	ld E[D] ld D st C	st C	comp done st tag ld addr ✓ st C ld D	A: valid B: valid exclusive C: ex−prf pending D: ld pending
7 value for D arrives; read of E[D] is reissued	ld E[D] ld D st C	st C	comp done st tag ld addr ✓ ld E[D] ✓ ✓ st C ld D	A,D: valid B: valid exclusive C: ex−prf pending E[D]: ld pending
8 ownership for C arrives	ld E[D]		comp done st tag ld addr ✓ ld E[D]	A,D: valid B,C: valid exclusive E[D]: ld pending
9 value for E[D] arrives				A,D,E[D]: valid B,C: valid exclusive

Figure P.3: Illustration of buffers during an execution with speculative reads.

Appendix Q

Implementation Issues for a More General Set of Events

This appendix describes some of the practical implementation issues for supporting ordering among the larger set of events discussed in Section 5.9. We primarily refer to the solutions adopted by various commercial architectures such as the Alpha [Sit92], PowerPC [MSSW94], and Sparc V9 [WG94]. The above three architectures use explicit fence instructions as the basic mechanism for ordering memory operations. Alpha and PowerPC provide a full fence instruction (MB in Alpha, SYNC in PowerPC), while Sparc V9 provides a more flexible fence instruction (MEMBAR). Ordering among different event types is enforced by overloading the above fence instructions and using a few specialized fence instructions.

Q.1 Instruction Fetch

An instruction fetch can be modeled as a read from memory. As discussed in the Chapter 4 however, instruction fetches behave differently from ordinary data read operations. Figure Q.1 shows a typical design with split instruction and data caches. Assume copies of locations A and B are initially held in both caches. First consider the order of instruction fetches with respect to previous conflicting writes from the same processor. Because of the separate instruction and data streams and the use of instruction prefetching, an instruction fetch can easily bypass previous conflicting writes from the same processor (e.g., consider the write to A shown in the figure); thus an instruction fetch does not necessarily return the value of the last write to that location in program order. Second, the fact that instructions and data reside in separate caches makes coherence actions non-atomic with respect to the two copies. For example, consider a write to location B by another processor that results in an incoming invalidate request to be sent to both caches. Because the invalidate requests traverse two separate paths, the write may complete at different times with respect to the

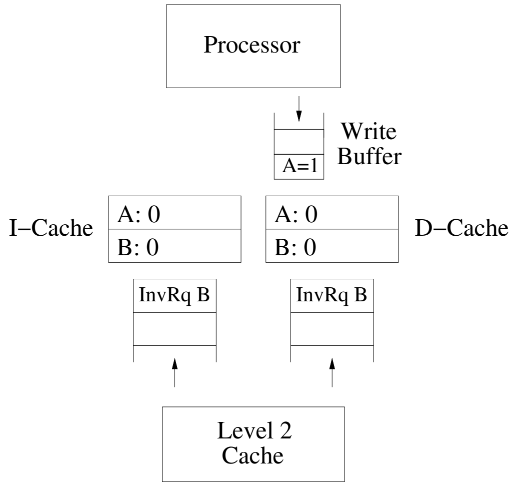


Figure Q.1: Split instruction and data caches.

two caches.[1] Finally, some designs may not provide hardware coherence with respect to the instruction cache. The above characteristics are not unique to multiprocessor systems and arise in uniprocessors as well.

To allow for a consistent view, most architectures provide special fence instructions that order future instruction fetches with respect to previous operations and eliminate stale copies by flushing operations in various queues and in some cases flushing copies from the instruction cache. This functionality is provided by an instruction memory barrier (IMB) in Alpha, an instruction sync (ISYNC) in PowerPC, and a FLUSH in the Sparc V9 architectures. None of the three architectures require hardware coherence for instruction caches. Implementations that support hardware coherence must flush the incoming queue to the instruction cache on an instruction fence in order to service invalidation and update requests that have been previously committed with respect to this processor.[2] The semantics of instruction fences differs among the three architectures when an implementation does not support hardware coherence for the instruction cache. For the Alpha, the IMB requires flushing the instruction cache thus eliminating any possible stale copies. The PowerPC architecture provides an instruction cache block invalidate (icbi) instruction that is issued by the processor that writes to the instruction region and forces specific lines to be flushed from other processors' instruction caches. Finally, the Sparc V9 architecture uses the FLUSH instruction in a similar way to the block invalidate operation of the PowerPC.

Q.2 Multiple Granularity Operations

Most architectures provide memory operations at multiple granularities, such as read and write operations to a byte, word, and double word. A larger granularity operation such as a double word operation can be conceptually modeled as multiple byte operations that are executed atomically with respect other conflicting operations to the same bytes. To enforce this atomic behavior, implementations typically service larger granularity operations as a single unit; to simplify this, the memory and cache line granularities are chosen to

[1] Effectively, there are two completion events for a write with respect to any processor: a completion event with respect to the data stream and another with respect to the instruction stream.

[2] We are assuming an implementation with early invalidation or update acknowledgements. The second solution discussed in Section 5.4.1 is more appropriate in this case since instruction fences are typically infrequent.

P1	P2
a1: A = 1;	*a2:* u,v = A,B;
b1: B = 1;	

(a)

P1	P2	P3	P4
a1: A = 1;	*a2:* u,v = A,B;	*a3:* B = 1;	*a4:* w,x = A,B;

(b)

Figure Q.2: Examples with multiple granularity operations.

be larger than the largest granularity data operation.

To illustrate some of the implementation issues, Figure Q.2 shows a couple of examples with different granularity operations. Assume locations A and B are consecutive 32-bit words on an aligned 64-bit boundary. Consider a natural extension of a model such as sequential consistency to multiple granularity operations. For the example in Figure Q.2(a), the outcome (u,v)=(0,1) would be disallowed. Similarly, for the example in Figure Q.2(b), the outcome (u,v,w,x)=(1,0,0,1) or (0,1,1,0) would be disallowed.

For most part, the implementation techniques for single granularity operations can be easily extended to deal with multiple granularities. A few subtle issues do arise, however. For example, assume an update-based implementation of SC that uses early update acknowledgements based on the second technique described in Section 5.4.1 where incoming updates are serviced as part of enforcing program orders. Consider the example in Figure Q.2(a). Assume P2 maintains cache copies for both locations A and B. With early update acknowledgements, P1's write generates an update request for A and an update request for B into P2's incoming queue. Since the writes to locations A and B are completely disjoint (i.e., they do not overlap on any bits), an aggressive implementation could allow the two updates to be reordered within P2's incoming queue. This could then lead to incorrect behavior since the double-word read on P2 could possibly observe the new value for B and yet the old value for A. To alleviate this problem, the implementation must disallow an update request to any part of an aligned 64-bit region from bypassing previous update or invalidate requests to the same region even if the two requests do not overlap. The above requirement is also required for the example in Figure Q.2(b) to behave correctly. A more conservative option is to enforce order among incoming operations to the same cache line (assumes a cache line is larger than the largest granularity operation which is a 64 bits in this case). Note that this restriction is inherently satisfied by the alternative solution for dealing with early update acknowledgements described in Section 5.4.1 since the order between incoming update requests and previous incoming update or invalidate requests is always upheld even if the requests are to completely different locations.

Another subtle issue arises in enforcing multiple-copy atomicity for writes. The above discussion on the example in Figure Q.2(b) assumes multiple-copy atomicity is enforced by disallowing any other processors from reading the new value for the write until the write is committed (or completed) with respect to all processors. The alternative is to use the second technique described in Section 5.3.5, whereby another processor is allowed to read the new value but is delayed for the write to commit (or complete) on the next program order that it enforces. To provide a correct implementation with this latter alternative, the coherence requirement must be extended to apply to writes to the same aligned 64-bit region even if there is no overlap between the write operations; this solution would also alleviate the anomalous behavior described in the previous paragraph.

Q.3 I/O Operations

I/O operations are typically implemented as uncached memory operations to specific regions in the memory space. The non-memory-like behavior of I/O operations requires disallowing certain reordering optimizations that may be safely used for ordinary data operations. Examples of such optimizations include merging two writes to the same address into a single write or forwarding the value of a write to a later read to the same address. Again, the Alpha, PowerPC, and Sparc V9 architectures allow the appropriate orders to be enforced through a combination of implicit orders and orders imposed by explicit fence operations.

Consider the order from an I/O write followed in program order by an I/O read to the same location. To enforce this order, Alpha overloads the memory barrier (MB) instruction, PowerPC provides the *eieio* (Enforce In-order Execution of I/O) operation, and the Sparc V9 provides the Look-Aside barrier. Implementations that implicitly disallow the read forwarding optimization for I/O can treat such fence operations as a nop. Disallowing the merging of two consecutive I/O writes to the same location is handled slightly different. The Alpha overloads the write memory barrier (WMB) or the ordinary MB for this purpose, the PowerPC uses the eieio instruction, and the Sparc V9 inherently disallows such merging on I/O writes. Enforcing orders among I/O operations to the same I/O device (even if they are not to the same address) can be achieved in Alpha and PowerPC by using the same fence instructions. The Sparc V9 overloads the ordinary MEMBAR for this purpose. Enforcing the order between operations to different I/O devices can be a little bit more tricky due to the non-uniform behavior of different I/O devices. In some cases, for example, an I/O write may need to be proceeded with an I/O read to the same device to determine whether the write has actually completed.

The ordering between I/O operations and ordinary memory operations may also be important in some cases. For example, memory operations may be used to synchronize access to an I/O device. The Alpha and PowerPC enforce such orders by overloading the MB and SYNC fences, respectively, while the Sparc V9 provides a new type of fence instruction (called the memory issue barrier) for this purpose.

A major simplification for I/O operations is that I/O locations are typically not cached; therefore, issues such as maintaining coherence or making writes appear atomic across multiple copies are not relevant. Nevertheless, detecting the completion of an operation such as an I/O write may still require an acknowledgement response.[3]

An implementation may provide various optimizations for handling I/O operations more efficiently. One important optimization is to coalesce I/O writes to consecutive addresses into a single write with a larger granularity. Another optimization is to exploit specific ordering guarantees within a design in order to pipeline

[3]Most commercial processors do not expect an acknowledgement for I/O writes, thus requiring external logic to keep track of outstanding I/O writes. A similar issue was described in Section 5.3.5 with respect to eager exclusive replies.

I/O operations. For example, point-to-point ordering guarantees within a network can be exploited to support pipelined I/O operations from a single processor to a single device while still maintaining the sequential order among the I/O operations. Such optimization are especially important for uses of I/O that require high bandwidth, such as communicating with a fast graphics device.

Q.4 Other Miscellaneous Events

There are typically a number of other events, such as exceptions and traps, whose ordering semantics is important. Many architectures provide yet other specialized fence instructions to ensure such orders. For example, the Alpha architecture provides exception and trap barriers that delay future operations until all previous arithmetic exceptions are resolved. Similarly, the synchronization barrier in the Sparc V9 architecture waits for all types of exceptions, including exceptions caused by address translation.

Q.5 Summary of Ordering Other Events Types

Overall, except for operations to multiple granularities of data, the ordering semantics of other types of events is quite confusing in most architectures. Many of the same issues are also present in uniprocessor systems. Fortunately, a lot fewer programmers have to directly deal with the semantics of such events relative to ordinary memory operations. Furthermore, since optimizations only matter for the frequent events, simple and potentially inefficient solutions are satisfactory for the infrequent events.

Bibliography

[ABM89] Yehuda Afek, Geoffrey Brown, and Michael Merritt. A lazy cache algorithm. In *Symposium on Parallel Algorithms and Architectures*, pages 209–222, June 1989.

[ABM93] Yehuda Afek, Geoffrey Brown, and Michael Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[ACFW93] Hagit Attiya, Soma Chaudhuri, Roy Friedman, and Jennifer Welch. Shared memory consistency conditions for non-sequential execution: Definitions and programming strategies. In *Proceedings of the Fifth Symposium on Parallel Algorithms and Architectures*, June 1993.

[Adv93] Sarita V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, December 1993. Available as Technical Report #1198.

[AF92] Hagit Attiya and Roy Friedman. A correctness condition for high-performance multiprocessors. In *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pages 679–690, May 1992.

[AGG+93] Sarita V. Adve, Kourosh Gharachorloo, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Sufficient system requirements for supporting the PLpc memory model. Technical Report #1200, Computer Sciences, University of Wisconsin - Madison, December 1993. Also available as Stanford University Technical Report CSL-TR-93-595.

[AH90a] Sarita V. Adve and Mark D. Hill. Implementing sequential consistency in cache-based systems. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I: 47–50, August 1990.

[AH90b] Sarita V. Adve and Mark D. Hill. Weak ordering - A new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[AH92a] Sarita V. Adve and Mark D. Hill. Sufficient conditions for implementing the data-race-free-1 memory model. Technical Report #1107, Computer Sciences, University of Wisconsin - Madison, September 1992.

[AH92b] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. Technical Report #1051, Computer Sciences, University of Wisconsin - Madison, September 1991, Revised September 1992.

[AH93] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[AHMN91] Sarita Adve, Mark Hill, Barton Miller, and Robert Netzer. Detecting data races on weak memory systems. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, May 1991.

[ALKK90] Anant Agarwal, Beng-Hong Lim, David Kranz, and John Kubiatowicz. April: A processor architecture for multiprocessing. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 104–114, May 1990.

[And91] Gregory R. Andrews. *Concurrent Programming: Principles and Practice*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[AP87] Todd R. Allen and David A. Padua. Debugging Fortran on a shared memory machine. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 721–727, August 1987.

[ASU86] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[BBD$^+$87] James Boyle, Ralph Butler, Terrence Disz, Barnett Glickfield, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., New York, NY, 1987.

[Bec90] Bob Beck. Shared-memory parallel programming in C++. *IEEE Software*, 7(4):38–48, July 1990.

[Ber66] Arthur J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, October 1966.

[BK89] Vasanth Balasundaram and Ken Kennedy. Compile-time detection of race conditions in a parallel program. In *Proceedings of the 3rd International Conference on Supercomputing*, pages 175–185, June 1989.

[BLL88] Brian Bershad, Edward Lazowska, and Henry Levy. Presto: A system for object-oriented parallel programming. *Software—Practice and Experience*, 18(8):713–732, August 1988.

[BMW85] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 processor-memory element. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 782–789, 1985.

[Buc62] W. Buchholz, editor. *Planning a Computer System: Project Stretch*. McGraw-Hill, 1962.

[BZ91] Brian Bershad and Matthew Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie-Mellon University, September 1991.

[BZS93] Brian Bershad, Matthew Zekauskas, and Wayne Sawdon. The Midway distributed shared memory system. In *Proceedings of COMPCON'93*, pages 528–537, February 1993.

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, October 1991.

[CD88] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-154, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1988.

[CF93] Alan L. Cox and Robert J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[CKA91] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 224–234, April 1991.

[Col92] William W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, Inc., 1992.

[CS90] David Callahan and Burton Smith. *Languages and Compilers for Parallel Computing*, chapter A Future-Based Parallel Language for a General-Purpose Highly-Parallel Computer, pages 95–113. Pitman, London; MIT Press, Massachusetts, 1990. Research Monographs in Parallel and Distributed Computing; also in 2nd Workshop on Languages and Compilers for Parallel Computing, Urbana, Illinois, August 1989.

[CSB93] Francisco Corella, Janice M. Stone, and Charles M. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report Computer Science Technical Report RC 18638(81566), IBM Research Division, T.J. Watson Research Center, January 1993.

[DEC81] Digital Equipment Corporation. *VAX Architecture Handbook*. 1981.

[DKCZ93] Sandhya Dwarkadas, Pete Keleher, Alan Cox, and Willy Zwaenepoel. Evaluation of release consistent software distributed shared memory on emerging network technology. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 144–155, May 1993.

[DS90a] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, pages 1–10, March 1990.

[DS90b] Michel Dubois and Christoph Scheurich. Memory access dependencies in shared-memory multiprocessors. *IEEE Transactions on Software Engineering*, 16(6):660–673, June 1990.

[DSB86] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, June 1986.

[DWB+91] Michel Dubois, Jin-Chin Wang, Luiz A. Barroso, Kangwoo Lee, and Yang-Syau Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Proceedings of Supercomputing '91*, pages 197–206, 1991.

[EKP+91] Philip G. Emma, Joshua W. Knight, James H. Pomerene, Rudolph N. Rechtachaffen, and Frank J. Sparacio. Posting out-of-sequence fetches. Patent Number 4,991,090, February 1991.

[FJ94] Keith Farkas and Norman Jouppi. Complexity/performance tradeoffs with non-blocking loads. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, pages 211–222, April 1994.

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[FP93] Bradly G. Frey and Raymond J. Pedersen. Coordination of out-of-sequence fetching between multiple processors using re-execution of instructions. Patent Number 5,185,871, February 1993.

[FVS92] Keith Farkas, Zvonko Vranesic, and Michael Stumm. Cache consistency in hierarchical ring-based multiprocessors. In *Proceedings of Supercomputing '92*, pages 348–357, November 1992.

[GAG+92] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Programming for different memory consistency models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[GAG+93] Kourosh Gharachorloo, Sarita V. Adve, Anoop Gupta, John L. Hennessy, and Mark D. Hill. Specifying system requirements for memory consistency models. Technical Report CSL-TR-93-594, Stanford University, December 1993. Also available as Computer Sciences Technical Report #1199, University of Wisconsin - Madison.

[GD90] Stephen R. Goldschmidt and Helen Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, 1990.

[GG91] Kourosh Gharachorloo and Phillip B. Gibbons. Detecting violations of sequential consistency. In *Symposium on Parallel Algorithms and Architectures*, July 1991.

[GGH91a] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 245–257, April 1991.

[GGH91b] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages I:355–364, August 1991.

[GGH92] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *Proceeding of the 19th Annual International Symposium on Computer Architecture*, pages 22–33, May 1992.

[GGH93a] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. Technical Report CSL-TR-93-567, Stanford University, April 1993.

[GGH93b] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Revision to "Memory consistency and event ordering in scalable shared-memory multiprocessors". Technical Report CSL-TR-93-568, Stanford University, April 1993.

[GHG+91] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *Proceeding of the 18th Annual International Symposium on Computer Architecture*, pages 254–263, May 1991.

[GLL+90] Kourosh Gharachorloo, Dan Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

[GM92] Phillip B. Gibbons and Michael Merritt. Specifying nonblocking shared memories. In *Symposium on Parallel Algorithms and Architectures*, pages 158–168, June 1992.

[GMG91] Phillip B. Gibbons, Michael Merritt, and Kourosh Gharachorloo. Proving sequential consistency of high-performance shared memories. In *Symposium on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[Goo89] James R. Goodman. Cache consistency and sequential consistency. Technical Report no. 61, SCI Committee, March 1989.

[Goo91] James R. Goodman. Cache consistency and sequential consistency. Technical Report Computer Sciences #1006, University of Wisconsin, Madison, February 1991.

[Han77] Per Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall Inc, Englewood Cliffs, NJ, 1977.

[HKMC90] Robert Hood, Ken Kennedy, and John M. Mellor-Crummey. Parallel program debugging with on-the-fly anomaly detection. In *Supercomputing '90*, pages 74–81, November 1990.

[HMTLB95] Ramin Hojati, Robert Mueller-Thuns, Paul Loewenstein, and Robert K. Brayton. Automatic verification of memory systems which service their requests out of order. In *Conference on Hardware Description Languages*, August 1995.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 1990.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[IBM83] *IBM System/370 Principles of Operation*. IBM, May 1983. Publication Number GA22-7000-9, File Number S370-01.

[JG88] Pierre Jouvelot and David K. Gifford. Parallel functional programming: the FX project. In M. Cosnard, Y. Robert, P. Quinton, and M. Raynal, editors, *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 257–267. North-Holland, October 1988.

[JLGS90] David V. James, Anthony T. Laundrie, Stein Gjessing, and Gurindar S. Sohi. Scalable coherent interface. *IEEE Computer*, 23(6):74–77, June 1990.

[Joh89] William M. Johnson. *Super-Scalar Processor Design*. PhD thesis, Stanford University, June 1989.

[Joh91] Mike Johnson. *Superscalar Microprocessor Design*. Prentice Hall, 1991.

[KCA92] John Kubiatowicz, David Chaiken, and Anant Agarwal. Closing the window of vulnerability in multiphase memory transactions. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 274–284, October 1992.

[KCZ92] Pete Keleher, Alan Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, May 1992.

[Kel75] R. M. Keller. Look-ahead processors. *Computing Surveys*, 7(4):177–195, December 1975.

[KEL91] Eric J. Koldinger, Susan J. Eggers, and Henry M. Levy. On the validity of trace-driven simulation for multiprocessors. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 244–253, May 1991.

[KMRS88] Anna R. Karlin, Mark S. Manasse, Larry Rudolph, and Daniel D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):79–119, 1988.

[KOH+94] Jeffrey Kuskin, David Ofelt, Mark Heinrich, John Heinlein, Richard Simoni, Kourosh Gharachorloo, John Chapin, David Nakahira, Joel Baxter, Mark Horowitz, Anoop Gupta, Mendel Rosenblum, and John Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302–313, April 1994.

[Kro81] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–85, 1981.

[KY94] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel SPMD programs. In *Languages and Compilers for Parallel Computing*, pages 331–345, August 1994.

[KY95] Arvind Krishnamurthy and Katherine Yelick. Optimizing parallel programs with explicit synchronization. In *Conference on Program Language Design and Implementation (PLDI)*, pages 196–204, June 1995.

[Lam79] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, September 1979.

[Lau94] James Laudon. *Architectural and Implementation Tradeoffs For Multiple-Context Processors*. PhD thesis, Stanford University, September 1994.

[Len92] Dan Lenoski. *The Design and Analysis of DASH: A Scalable Directory-Based Multiprocessor*. PhD thesis, Stanford University, February 1992.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[LHH91] Anders Landin, Erik Hagersten, and Seif Haridi. Race-free interconnection networks and multiprocessor consistency. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 27–30, May 1991.

[Lig94] Bruce D. Lightner. The thunder sparc processor. In *Hot Chips VI*, pages 201–217, August 1994.

[LLG+90] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[LLG+92] Dan Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam. The Stanford Dash multiprocessor. *IEEE Computer*, 25(3):63–79, March 1992.

[Lov93] David B. Loveman. High performance Fortran. *IEEE Parallel and Distributed Technology, Systems & Applications*, 1(1):25–42, February 1993.

[LR80] Butler W. Lampson and David D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, 23(2):105–117, February 1980.

[LRW91] Monica Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, April 1991.

[LS84] Johnny K. F. Lee and Alan J. Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 17:6–22, January 1984.

[MB88] Jeffrey D. McDonald and Donald Baganoff. Vectorization of a particle simulation method for hypersonic rarified flow. In *AIAA Thermodynamics, Plasmadynamics and Lasers Conference*, June 1988.

[MC91] Sang Lyul Min and Jong-Deok Choi. An efficient cache-based access anomaly detection scheme. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 235–244, April 1991.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[MG91] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, June 1991.

[Mow94] Todd Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, March 1994.

[MP91] Stephen Melvin and Yale Patt. Exploiting fine-grained parallelism through a combination of hardware and software techniques. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 287–296, May 1991.

[MPC89] Samuel Midkiff, David Padua, and Ron Cytron. Compiling programs with user parallelism. In *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[MSSW94] Cathy May, Ed Silha, Rick Simpson, and Hank Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors*. Morgan Kaufmann Publishers, Inc., 1994.

[NM89] Robert Netzer and Barton Miller. Detecting data races in parallel program executions. Technical Report CS-894, University of Wisconsin - Madison, November 1989.

[NM90] Robert Netzer and Barton Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages II: 93–97, August 1990.

[NM91] Robert Netzer and Barton Miller. Improving the accuracy of data race detection. In *Proceedings of ACM Symposium on Principles and Practice of Parallel Programming*, pages 133–144, April 1991.

[OMB92] Kunle Olukotun, Trevor Mudge, and Richard Brown. Performance optimization of pipelined primary caches. In *Proceeding of the 19th Annual International Symposium on Computer Architecture*, pages 181–190, May 1992.

[Pap86] Christos Papadimitriou. *The Theory of Database Concurrency Control*. Computer Science Press, 1986.

[PD95] Seungjoon Park and David L. Dill. An executable specification, analyzer and verifier for RMO (Relaxed Memory Order). In *Proceedings of the Seventh Symposium on Parallel Algorithms and Architectures*, pages 34–41, July 1995.

[Res92] Kendall Square Research. *KSR1 Technical Summary*. Waltham, MA, 1992.

[RG90] Ed Rothberg and Anoop Gupta. Techniques for improving the performance of sparse factorization on multiprocessor workstations. In *Proceedings of Supercomputing '90*, November 1990.

[Ros88] Jonathan Rose. Locusroute: A parallel global router for standard cells. In *Design Automation Conference*, pages 189–195, June 1988.

[RS84] Larry Rudolph and Zary Segall. Dynamic decentralized cache schemes for MIMD parallel processors. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, pages 340–347, June 1984.

[RSL93] Martin Rinard, Daniel Scales, and Monica Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, 26(6):28–38, 1993.

[SBS93] Per Stenstrom, Mats Brorsson, and Lars Sandberg. An adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[Sch89] Christoph Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors*. PhD thesis, University of Southern California, May 1989.

[SD87] Christoph Scheurich and Michel Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th Annual International Symposium on Computer Architecture*, pages 234–243, June 1987.

[SD88] Christoph Scheurich and Michel Dubois. Concurrent miss resolution in multiprocessor caches. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages I: 118–125, August 1988.

[SD91] Christoph Scheurich and Michel Dubois. Lockup-free caches in high-performance multiprocessors. *Journal of Parallel and Distributed Computing*, 11(1):25–36, January 1991.

[SFC91] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. Technical Report (PARC) CSL-91-11, Xerox Corporation, Palo Alto Research Center, December 1991.

[SG89] Larry Soule and Anoop Gupta. Parallel distributed-time logic simulation. *IEEE Design and Test of Computers*, 6(6):32–48, December 1989.

[SH92] Jaswinder Pal Singh and John L. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experience, results, and implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[SHG92] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Implications of hierarchical N-body methods for multiprocessor architecture. Technical Report CSL-TR-92-505, Stanford University, 1992.

[Sim92] Richard Simoni. *Cache Coherence Directories for Scalable Multiprocessors*. PhD thesis, Stanford University, October 1992.

[Sit92] Richard L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[SLH90] Michael Smith, Monica Lam, and Mark Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, May 1990.

[SP85] James E. Smith and Andrew R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pages 36–44, June 1985.

[SS88] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[SUN91] *The SPARC Architecture Manual*. Sun Microsystems Inc., January 1991. No. 800-199-12, Version 8.

[SW95] Richard L. Sites and Richard T. Witek, editors. *Alpha AXP Architecture Reference Manual*. Digital Press, 1995. Second Edition.

[SWG91] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. Technical Report CSL-TR-91-469, Stanford University, May 1991.

[Tay83a] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[Tay83b] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[TH90] Josep Torrellas and John Hennessy. Estimating the performance advantages of relaxing consistency in a shared-memory multiprocessor. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages I: 26–33, August 1990.

[TLH94] Josep Torrellas, Monica Lam, and John Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.

[Tom67] R. M. Tomasulo. An efficient hardware algorithm for exploiting multiple arithmetic units. *IBM Journal*, 11:25–33, January 1967.

[Web93] Wolf-Dietrich Weber. *Scalable Directories for Cache-Coherent Shared-Memory Multiprocessors*. PhD thesis, Stanford University, January 1993.

[WG94] David L. Weaver and Tom Germond, editors. *The SPARC Architecture Manual*. Prentice Hall, 1994. SPARC International, Version 9.

[Wir77] Niklaus Wirth. Modula: A language for modular multiprogramming. *Software—Practice and Experience*, 7(1):3–36, January 1977.

[WL92] Andrew W. Wilson and Rick P. LaRowe. Hiding shared memory reference latency on the Galactica Net distributed shared memory architecture. *Journal of Parallel and Distributed Computing*, 15(4):351–367, August 1992.

[ZB92] Richard N. Zucker and Jean-Loup Baer. A performance study of memory consistency models. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 2–12, May 1992.

[Zuc92] Richard N. Zucker. *Relaxed Consistency and Synchronization in Parallel Processors*. PhD thesis, University of Washington, December 1992. Also available as Technical Report No. 92-12-05.

End.