Designing Memory Consistency Models For Shared-Memory Multiprocessors

Sarita V. Adve

Computer Sciences Technical Report #1198
University of Wisconsin-Madison
December 1993

DESIGNING MEMORY CONSISTENCY MODELS FOR SHARED-MEMORY MULTIPROCESSORS

by

SARITA VIKRAM ADVE

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

1993

Abstract

The memory consistency model (or memory model) of a shared-memory multiprocessor system influences both the performance and the programmability of the system. The simplest and most intuitive model for programmers, sequential consistency, restricts the use of many performance-enhancing optimizations exploited by uniprocessors. For higher performance, several alternative models have been proposed. However, many of these are hardware-centric in nature and difficult to program. Further, the multitude of many seemingly unrelated memory models inhibits portability. We use a ***3P*** criteria of ***p***rogrammability, ***p***ortability, and ***p***erformance to assess memory models, and find current models lacking in one or more of these criteria. This thesis establishes a unifying framework for reasoning about memory models that leads to models that adequately satisfy the 3P criteria.

The first contribution of this thesis is a programmer-centric methodology, called *sequential consistency normal form (SCNF)*, for specifying memory models. This methodology is based on the observation that performance enhancing optimizations can be allowed without violating sequential consistency if the system is given some information about the program. An SCNF model is a contract between the system and the programmer, where the system guarantees both high performance and sequential consistency only if the programmer provides certain information about the program. Insufficient information gives lower performance, but incorrect information violates sequential consistency. This methodology adequately satisfies the 3P criteria of programmability (by providing sequential consistency), portability (by providing a common interface of sequential consistency across all models), and performance (by only requiring the appearance of sequential consistency for correct programs).

The second contribution demonstrates the effectiveness of the SCNF approach by applying it to the optimizations of several previous hardware-centric models. We propose four SCNF models that unify many hardware-centric models. Although based on intuition similar to the hardware-centric models, the SCNF models are easier to program, enhance portability, and allow more implementations (and potentially higher performance) than the corresponding hardware-centric models.

The third contribution culminates the above work by exposing a large part of the design space of SCNF models. The SCNF models satisfy the 3P criteria well, but are difficult to design. The complexity arises because the relationship between system optimizations and programmer-provided information that allows system optimizations without violating sequential consistency is complex. We simplify this relationship and use the simple relationship to characterize and explore the design space of SCNF models. In doing so, we show the unexploited potential in the design space, leading to several new memory models.

The final contribution concerns debugging programs on SCNF models. While debugging, the programmer may unknowingly provide incorrect information, leading to a violation of sequential consistency. We apply debugging techniques for sequential consistency to two of the SCNF models to alleviate this problem.

Acknowledgements

Many people have contributed to this work. Here, I can only mention a few.

Mark Hill, my advisor, has provided invaluable technical and professional advice, which has guided me throughout my graduate school years and will continue to guide me in the future. Also, his constant support and encouragement made the Ph.D. process so much less overwhelming.

I am indebted to Jim Goodman for the support and encouragement he provided, especially in my initial years at Wisconsin. I have also enjoyed many stimulating discussions with him on memory consistency models in particular, and computer architecture in general.

I am specially grateful to Guri Sohi for raising many incisive questions on this work, and then for his patience in many long discussions to address those questions.

Jim Larus and David Wood have on many occasions provided valuable feedback on this work.

I have enjoyed many, many hours of discussion on memory consistency models with Kourosh Gharachorloo. Some parts of this thesis have evolved from our joint work.

I have often looked towards Mary Vernon for professional advice, and am grateful to her for always making the time for me.

The support and encouragement I have received from my parents, Vikram’s parents, and our brothers and sisters is immeasurable. Their pride in our work made each little achievement so much more pleasurable, and the final goal even more desirable.

Finally, my husband, Vikram, has been both colleague and companion. I have been able to rely on his frank appraisals of my work, and his technical advice and support have come to my rescue umpteen times. His constant presence through all the ups and downs of graduate school has made the ride so much smoother.

Table of Contents

Chapter 1

Introduction

1.1. Motivation

Parallel systems can potentially use multiple uniprocessors to provide orders of magnitude higher performance than state-of-the-art uniprocessor systems at comparable cost. Parallel systems that provide an abstraction of a single address space (or shared-memory systems) simplify many aspects of programming, compared to the alternative of message passing systems. For example, the shared-memory abstraction facilitates load balancing through processor independent data structures, allows pointer-based data structures, and allows the effective use of the entire system memory [LCW93, LeM92]. Shared-memory also permits decoupling the correctness of a program from its performance, allowing incremental tuning of programs for higher performance.

The shared-memory abstraction can be provided either by hardware, software, or a combination of both. Pure hardware configurations include uniform [GGK83] and non-uniform access machines [ReT86]. Most configurations employ caches to reduce memory latency, and either use snooping [Bel85] or directory-based protocols [ASH88, Gus92, LLG90] to keep the caches up-to-date. Runtime software based systems typically use virtual memory hardware to trap on non-local references, and then invoke system software to handle the references [CBZ91, Li88]. Other software-based systems depend on compilers that detect non-local shared memory accesses in high-level code, and convert them into appropriate messages for the underlying message passing machine [HKT92]. Several systems employ a combination of hardware and software techniques. For example, in systems using software-based cache coherence [BMW85, ChV88, PBG85], hardware provides a globally addressable memory, but the compiler is responsible for ensuring that shared data in a cache is up-to-date when required by its processor. The Alewife system [ALK90] and the Cooperative Shared Memory scheme [HLR92] manage cacheable shared-data in hardware for the common cases, but invoke runtime software for the less frequent cases.

Building scalable, high-performance shared-memory systems that are also easy to program, however, has remained an elusive goal. This thesis examines one aspect of shared-memory system design that affects both the performance and programmability of all of the above types of shared-memory systems. This aspect is called the *memory consistency model* or *memory model*.

A memory model for a shared-memory system is an architectural specification of how memory operations of a program will appear to execute to the programmer. The memory model, therefore, specifies the values that read operations of a program executed on a shared-memory system may return. The terms *system*, *program* and *programmer* can be used at several levels. At the lowest level, the system is the machine hardware, a program is the machine code, and a programmer is any person that writes or reasons about such machine code. At a higher level, the system may be the machine hardware along with the software that converts high-level language code into machine-level code, a program may be the high-level language code, and a programmer may be the writer of such programs. A memory model specification is required for every level of the system, and affects the programs and programmers of that level.

The memory model affects the ease of programming since programmers must use the model to reason about the results their programs will produce. The memory model also affects the performance of the system because it determines when a processor can execute two memory operations in parallel or out of program order, when a value written by one processor can be made visible to other processors, and how much communication a memory operation will incur. Thus, the memory model forms an integral part of the entire system design (including the processor, interconnection network, compiler, and programming language) and the process of writing parallel programs.

Most uniprocessor systems provide a simple memory model to the programmer that ensures that memory operations will appear to execute one at a time, and in the order specified by the program (program order). Thus, a read returns the value of the last write to the same location that is before it by program order. However, to improve performance, uniprocessor hardware often allows memory operations to be overlapped with other memory operations, and to be issued and executed out of program order. Nevertheless, it maintains the programmer's model of memory by maintaining uniprocessor data and control dependences (e.g., by using interlock logic), which ensures that memory operations *appear* to execute one at a time in program order. Similarly, uniprocessor compilers often reorder memory operations but preserve the memory model by preserving uniprocessor dependences. Thus, the programmer's model for uniprocessor memory is simple and yet allows for high performance through several hardware and compiler optimizations.

The goal of this work is to establish a framework for specifying shared-memory models that retain the attributes of simplicity and high performance found in the uniprocessor model, and to design useful models within this framework.

The most commonly (and often implicitly) assumed programmer's model of memory for shared-memory systems is *sequential consistency*. This model was first formalized by Lamport [Lam79] and is a natural extension of the uniprocessor model. A system is sequentially consistent if (1) all memory operations appear to execute one at a time in some total order (i.e., memory operations appear atomic), and (2) all memory operations of a given processor appear to execute in program order. This definition allows the programmer to view the system as in Figure 1.1. A system is simply a set of processors and a single shared-memory module with a central switch; after an arbitrary number of time steps, the switch connects the memory to an arbitrary processor and the processor executes the next memory operation specified by the program. Alternatively, the system appears like a multiprogrammed uniprocessor [Pat83-86].

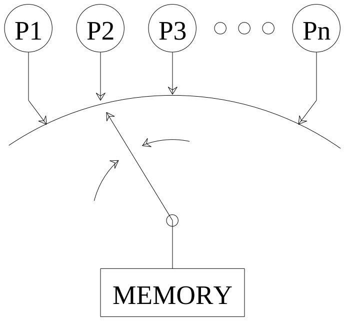


Figure 1.1. A sequentially consistent system.

Figure 1.2 shows two program fragments that illustrate how programmers use sequential consistency. Variables *X*, *Y*, and *flag* in the figure are shared variables, and r1 and r2 are local registers. The code in part (a) shows a producer-consumer interaction where processor P_1 writes *X* and *Y*, and then sets a *flag* to indicate that it has completed its writes. Processor P_2 waits for *flag* to get updated and then reads *X* and *Y*, expecting to get the new values of *X* and *Y*. The code relies on the assumption that after P_2 returns the updated value of *flag*, it will return the updated values of *X* and *Y*. Sequential consistency allows programmers to make this assumption. The code in part(b) is the core of Dekker's algorithm for implementing critical sections. Processors P_1 and P_2 respectively write *X* and *Y* and then read the variable written by the other processor. The algorithm relies on the intuitive assumption that if P_1's read happens before P_2's write of *Y* (i.e., the read returns 0), then P_1's write of *X* will happen before P_2's read (i.e., P_2's read will return 1). Similarly, if P_2's read returns 0, then P_1's read will return 1. Thus, it cannot be that both P_1 and P_2 return 0. Again, sequential consistency allows programmers to make the above intuitive assumption by ensuring that operations of a single processor will appear to execute in program order.

Initially X = Y = flag = 0

P_1	P_2
X = 14	while (flag != 1) {;}
Y = 26	r1 = X
flag = 1	r2 = Y

(a)

Initially X = Y = 0

P_1	P_2
X = 1	Y = 1
r1 = Y	r2 = X

(b)

Figure 1.2. Programming with sequential consistency.

For part (a), sequential consistency ensures that r1 = 14, r2 = 26, and for part (b), sequential consistency ensures that both r1 and r2 cannot be 0.

The view of the system provided by sequential consistency retains the simplicity of the uniprocessor model for programmers; however, implementing systems to provide this view often involves serious compromises in performance [DSB86, Lam79, MPC89]. Consider the code in part(b) of figure 1.2. Consider a system that uses write buffers to allow a read to bypass a write that precedes it in the program. On such a system, processors P_1 and P_2 could both complete their reads while their writes are still in the write buffer. Both reads would return the initial value of 0, violating sequential consistency. Similarly, in a system where processors can reorder their instructions, P1 could issue its read of Y before its write of X, again allowing both P1 and P2 to read the old values of Y and X respectively [DSB86, Lam79]. Note that neither processor has data dependencies among its instructions (because X and Y are different locations); therefore, simple interlock logic will not preclude either processor from issuing its second instruction before the first. The above violations of sequential consistency take place irrespective of whether the system is bus-based or has a general interconnection network, and irrespective of whether the system has caches or not. For the code in part (a) of figure 1.2, consider a system that issues its instructions in program order and does not employ write buffers. Suppose, however, that the system has multiple memory modules and a general interconnection network that provides different paths to the different memory modules. If processors are allowed to overlap their memory operations on the interconnection network, it is possible that P_1's write of *flag* reaches its memory module before P_1's writes of *X* and *Y*. Thus, it is possible for P_2 to return the new value of *flag* but the old values of *X* and *Y*, violating sequential consistency [Lam79]. Analogously, if the compiler reorders memory operations or allocates memory locations in registers (e.g., allocates *Y* in a register in P_2 in figure 1.2(b)), then again the non-sequentially consistent executions described above can occur [MPC89].

Chapter 2 will show that to implement sequential consistency on a general system and with a reasonable level of complexity, a processor must often execute its shared-memory operations one at a time and in the order specified by the program. This prohibits the use of many performance enhancing hardware features commonly used in uniprocessors such as write buffers, out-of-order issue, pipelining (or overlapping) memory operations, and lockup-free caches [Kro81]. Analogously, in the absence of extensive data dependence analysis, compilers cannot reorder instructions that generate shared-memory operations and cannot allocate shared-memory locations to registers, again sacrificing performance enhancing techniques from uniprocessors. Furthermore, to ensure that writes appear atomic on a cache-based system, practically, a newly written value cannot be made visible to any processor until all caches with an old value have been invalidated or updated, possibly incurring additional memory latency and network traffic.

Henceforth, for the hardware and runtime system, we use the term *optimization* to refer to features (such as described above) that allow a memory operation to execute before the completion of operations that precede it by program order, or features that allow writes to be executed non-atomically (i.e., a processor can read a new value even if another processor can still read an old value). Similarly, for the compiler, we use the term *optimization* to refer to the reordering of instructions that access shared-memory, and register allocation of shared-memory locations.

To improve the performance of shared-memory systems, researchers have proposed alternative memory models that impose constraints weaker than sequential consistency. Many of these models have been motivated by hardware optimizations and are specified in terms of these hardware optimizations. This *hardware-centric* nature of the models leads to substantial performance increases [GGH91a, GGH92, ZuB92], but at the cost of losing significant advantages of sequential consistency for the programmer.

The first disadvantage for programmers of hardware-centric models is that the programmer's view of the system is more complex than with sequential consistency. For example, write buffers and caches are effective hardware features for reducing and tolerating memory latency. However, a side effect of using these features aggressively is that writes can appear to execute non-atomically. Thus, many of the models that allow the aggressive use of write buffers and caches also allow writes to appear non-atomic; and reasoning with such models requires programmers to be explicitly aware of the underlying write buffers and caches that result in such non-atomicity. Parallel programming assuming atomic writes is already complex enough; reasoning with non-atomic memory further increases this complexity.

The second, perhaps more serious, disadvantage for programmers of current hardware-centric models is that the absence of any unifying framework for designing memory models has led to several seemingly unrelated models, sometimes with subtle differences in specification that lead to significantly different program behavior. Thus, programmers must reason with *many* different (and fairly complex) interfaces, making porting programs between different models difficult. Figure 1.3 captures some of this variety of interfaces by showing several currently implemented hardware-centric models (to be defined further in Chapter 2) and their relationships with each other [AGG91]. Thus, while for the first disadvantage, it may be argued that programmers may be willing to tolerate a more complex interface to get higher performance, the presence of a multitude of complex interfaces significantly exacerbates the problem.

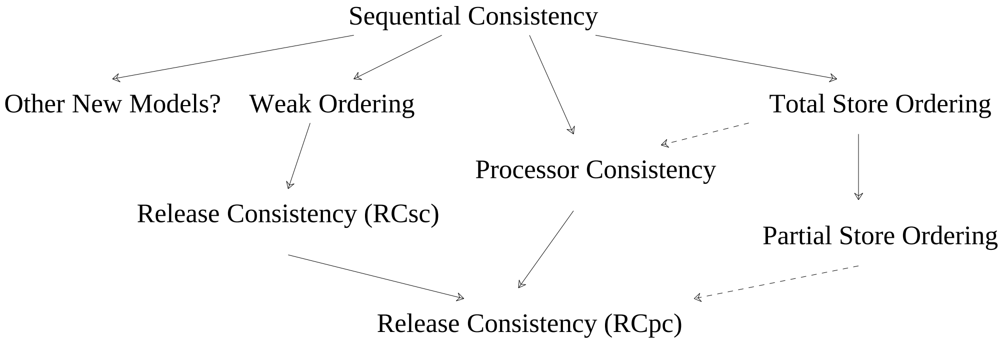


Figure 1.3. Some hardware-centric models.

An arrow from one model to another indicates that the second model allows for more optimizations than the first and hence the second model provides higher performance but a possibly more complex interface than the first. Two models not connected with arrows cannot be compared in terms of performance or programmability. Chapter 2 discusses these models and their relationships in more detail. The dotted arrows exist only if a recent, subtle revision to processor consistency and release consistency (RCpc) [GGH93] is assumed (see Chapter 2), an additional testimony to the complexity of reasoning with hardware-centric memory models.

Finally, even from the system designer's viewpoint, the absence of a unifying framework for designing memory models makes it difficult to determine new, useful memory models that might allow new optimizations.

This thesis establishes a unifying framework for designing memory models and develops memory models within this framework that alleviate the above problems while still allowing the optimizations of the hardware-centric models. Specifically, we use a ***3P*** criteria of **p**rogrammability, **p**ortability, and **p**erformance for evaluating memory models, and show how to design memory models that adequately satisfy all three criteria.[1]

[1] We have recently, in a joint effort, proposed a uniform methodology for expressing the system constraints of various hardware-centric models [GAG93]. This methodology is meant to aid system implementors and does not have the proper-

The memory models and theoretical framework of this thesis apply to all levels of the system; however, the formal definitions of the memory models and the various concepts involved are mostly provided for the lowest level (usually the hardware). While programmers can use the low-level specifications directly, it would be more convenient to convert them to a higher-level. This thesis illustrates how such conversions can be made, but often leaves the formalization of appropriate high-level specifications to future language designers.

This thesis also does not provide any quantitative figures on the performance of the proposed memory models. Other researchers have already demonstrated that alternative memory models can give high performance [DKC93, GGH91a, GGH92, KCZ92, ZuB92]. This thesis addresses the problem of how to make the available performance potential usable for programmers and how to reason about models in a way that can result in better performance than that of previous models.

1.2. Summary of Contributions

The contributions of this thesis are as follows.

(1) *A programmer-centric approach for specifying memory models.*

The first contribution of this thesis is a programmer-centric approach for specifying memory models, and is based on the following observations. To adequately satisfy the first two of the 3P criteria of programmability and portability, a simple, uniform interface should be provided to programmers of all shared-memory systems. Sequential consistency is a desirable candidate for such a uniform interface, but is inadequate for the third criteria of performance. Is there a way to get both high performance *and* sequential consistency? We have illustrated example program fragments for which the use of common performance-enhancing optimizations violates sequential consistency. Nevertheless, there are other program fragments where applying various optimizations does not violate sequential consistency and provides significant performance benefits. Thus, a way to get high performance with sequential consistency is for the system to *selectively* apply optimizations to only those parts of the program where the optimization would not violate sequential consistency. Unfortunately, for general programs, it is difficult for currently practical systems to determine which parts of the program are safe for applying a given optimization. This thesis, therefore, proposes that the programmer identify to the system the safe parts of a program that can allow different optimizations.

Specifically, we propose that a memory model be specified as a contract between the system and the programmer where the programmer provides some information about the program to the system, and the system uses the information to provide both sequential consistency and high performance. If the programmer provides incorrect information, then the system may violate sequential consistency. If the programmer does not provide enough information, then the system cannot provide high performance. We call this method of specification as the *sequential consistency normal form (SCNF)* method.

It follows that an SCNF memory model is simply a specification of the information that the system can use to provide high performance without violating sequential consistency. Different SCNF models differ in the information they can exploit (and consequently in the optimizations they can employ). Thus, rather than viewing different SCNF specifications as different models of memory, an SCNF specification may be viewed as the layer of abstraction that specifies how the programmer can get both high performance and sequential consistency on modern systems.

When viewed as a method of specifying memory models, the SCNF method improves upon the hardware-centric method with respect to the 3-P criteria as follows. First, the SCNF specification allows programmers to continue reasoning with sequential consistency, the most intuitive model. Second, it allows a uniform interface (i.e. sequential consistency) for the programmer across all shared-memory systems, enhancing portability. Finally, the information provided by programmers allows optimizations that lead to high performance (without a violation of sequential consistency). Note that the SCNF models allow optimizations similar to those for the hardware-centric models, including the optimization of non-atomic writes. Unlike many hardware-centric models, however, programmers of SCNF models need never be aware of these optimizations and can always

ties of the framework described above. Specifically, it is not recommended for programmers and does not provide insight for new, useful memory models.

program with the view of figure 1.1.

(2) *New programmer-centric models that allow more implementations than old hardware-centric models.*

The second contribution of this thesis demonstrates the effectiveness of the SCNF approach by applying it to the optimizations of several commercially implemented and several academic models. We develop four SCNF models that unify the IBM 370 model [IBM83], the SPARC V8 models of total store ordering and partial store ordering [SUN91], the Alpha model [Sit92], weak ordering [DSB86], processor consistency [GLL90], two flavors of release consistency (RCsc and RCpc) [GLL90], and lazy release consistency [KCZ92]. The four SCNF models — data-race-free-0, data-race-free-1, PLpc1, and PLpc2 — exploit strictly increasing amounts of information from the programmer regarding the behavior of memory operations in the program. They use this information to allow increasingly many optimizations, including those of the above hardware-centric models, without violating sequential consistency. The intuition that motivates the SCNF models is similar to that of the hardware-centric models; however, their programmer-centric nature allows programmers to continue reasoning with sequential consistency and at the same time allows for a superset of the optimizations of the hardware-centric models. The SCNF models also enhance portability by unifying several hardware-centric models and other not-yet-defined systems; a program written for an SCNF model can be run on all of the unified systems without violating sequential consistency. Figure 1.4 shows some of the systems unified by the different SCNF models. A rectangle labeled with an SCNF model encloses the systems unified by the SCNF model. Specifically, programs written for PLpc2 can be run on *all* of the commercially implemented hardware-centric models and the academic models mentioned above without violating sequential consistency, and usually exploiting the full performance potential of the hardware-centric models.

Since the four SCNF models exploit strictly increasing amounts of information, any program written for a more aggressive model can clearly run on a system implementing a less aggressive model; the less aggressive system can simply ignore the extra information in the program for the more aggressive model. Furthermore, a key feature of the above SCNF models is that they do not require perfect information, but allow programmers to be conservative in their specifications. This allows a program written for a less aggressive model to be run on a more aggressive system (potentially with the performance of the less aggressive model) by simply specifying the conservative options for the additional information exploited by the more aggressive model. Thus, the four models provide a spectrum of exploitable information and performance; the programmer can choose to be at any point on the spectrum irrespective of the specific system the program is written for.

The new models require the programmer to distinguish certain memory operations on which optimizations can be applied without violating sequential consistency. The data-race-free-0 model, for example, requires distinguishing operations that are never involved in a race (called data operations) from those that may be involved in a race (called synchronization operations). The other models additionally require distinguishing different race operations based on certain characteristics. As discussed above, an important aspect of these models is that they do not require perfect information and allow the programmer to be conservative in distinguishing a memory operation. For data-race-free-0, for example, if the programmer does not know whether an operation will be involved in a race, the operation can be conservatively distinguished as a synchronization operation. The more accurate the information, the higher the performance potential.

(3) *A framework to characterize and explore the design space of memory models.*

The third contribution is a framework that formally characterizes and exposes the design space of SCNF memory models, and its use to explore the design space.

The framework.

With the SCNF approach, a memory model is simply a method of procuring information from the programmer to allow optimizations without violating sequential consistency. Thus, the key to determining the design space is to determine the nature of useful information and corresponding optimizations that can be exploited while still retaining sequential consistency. The framework formalizes this relationship between information and optimizations, thereby characterizing and exposing the design space of memory models. This largely eliminates the complexity and ad hoc nature formerly present in the design process of memory models.

The development of the framework involves identifying an aggressive, sufficient condition for sequential consistency and then using it to characterize when an optimization will not violate sequential consistency. The

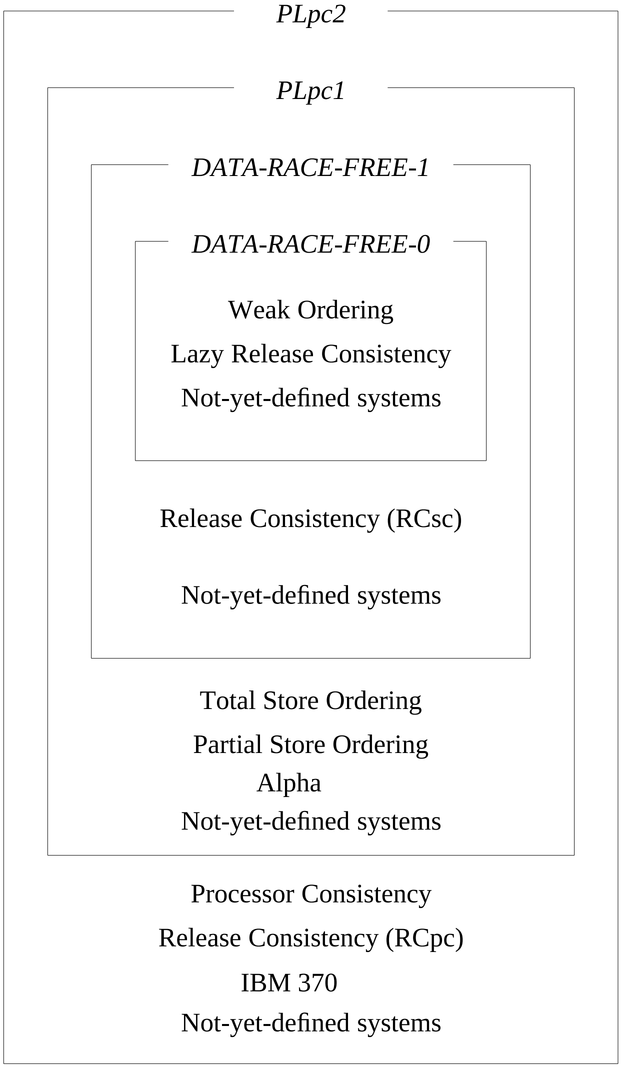


Figure 1.4. SCNF models unify hardware-centric systems.

condition used constrains certain paths called the *ordering paths* in a graph derived from an execution. It requires a certain subset of these paths called the *critical paths* to be executed ‘‘safely’’ by the system. We deduce that an optimization will not violate sequential consistency if it will not affect the safe execution of the critical paths. Thus, useful information is that which identifies the non-critical paths and useful optimizations are those that can exploit such information to execute non-critical paths aggressively.

A major part of the complexity of this work is to ensure that information about non-critical paths from only sequentially consistent executions is sufficient; this is important since we do not want the programmer to reason about executions that are not sequentially consistent. For this purpose, we develop a pre-condition, called the *control requirement*, that is obeyed by most current systems. We prove that for systems that obey the control requirement, information from sequentially consistent executions is indeed sufficient. The control requirement is an important contribution of this work since much previous work addressing aggressive optimizations does not adequately address this requirement, and is therefore incomplete. Although the requirement is obeyed by most currently practical systems, formalizing it is essential to determine the correctness of any system (and specifically, future, more aggressive systems).

Exploring the design space.

We use our framework to develop several new models by examining several different optimizations and information about program behavior. In particular, we examine the optimizations of out-of-order execution of memory operations, in-order execution with pipelining in the interconnection network, eliminating one phase of update protocols required by sequential consistency, and eliminating acknowledgement messages. We show that each of these optimizations can be applied safely to more cases than allowed by previous models, and determine the information that will allow this, thereby leading to new models. We consider specific synchronization constructs such as locking, producer-consumer synchronization, and barriers and show that for many common uses of these synchronization constructs, the above optimizations can be applied more aggressively than before. We characterize these common usages, again resulting in new and more aggressive memory models.

Characterizing the design space.

We finally use the relationship between optimizations and information to deduce the key characterization of a memory model that determines its performance and programmability. The key to a memory model is the ordering paths that it executes safely. We call such ordering paths the *valid paths* of the model. We define a generic memory model in terms of its valid paths, and define the programmer and system constraints for such a model in terms of these paths.

(4) *A technique to detect data races on data-race-free systems*

The fourth contribution concerns detecting data races in a program with the data-race-free-0 and data-race-free-1 models. These models guarantee sequential consistency only if the program does not contain data races (i.e., all operations that may be involved in a race are distinguished as synchronization). It is possible, however, that due to the presence of a bug, a program may contain some data races. It is desirable that even while debugging such a program, the programmer be able to reason with the interface of sequential consistency. We show how current techniques for detecting data races in sequentially consistent systems can be extended to data-race-free systems in a manner that allows programmers to continue reasoning with sequential consistency even while debugging programs that are not yet data-race-free.

Static techniques for data race detection in sequentially consistent systems can be directly applied to data-race-free systems as well. Dynamic techniques, on the other hand, may report data races that could never occur on sequentially consistent systems. This can complicate debugging because programmers can no longer assume the model of sequential consistency. We show how a post-mortem dynamic approach can be used to detect data races effectively even on data-race-free systems. The key observation we make is that many practical systems preserve sequential consistency at least until the first data races (those not affected by any others). We formalize this condition and show that for an execution on a system that obeys this condition, we can either (1) correctly report no data races and conclude the execution to be sequentially consistent or (2) report the first data races that also occur on a sequentially consistent execution (within a few limitations). Our condition is met by many currently practical implementations of the data-race-free models, and so our technique can exploit the full performance of these systems.

1.3. Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 discusses related work. Chapter 3 describes the SCNF approach for specifying memory models, and also develops terminology used throughout the thesis for defining SCNF memory models. Chapter 4 describes the data-race-free-0 memory model. Chapter 5 describes implementations of the data-race-free-0 memory model, and also develops a methodology for describing implementations of SCNF models. Chapter 6 discusses the data-race-free-1, PLpc1, and PLpc2 models. Chapter 7 develops the framework to explore the design space of SCNF memory models, and uses the framework to define several new models and to characterize the design space. Chapter 8 describes the technique to identify data races in data-race-free systems. Chapter 9 gives conclusions. The appendices formalize certain concepts introduced in the above chapters and provide correctness proofs for the results of the above chapters.

Chapter 2

Related Work

This chapter briefly describes related work on memory models, and also motivates models that impose weaker system constraints than sequential consistency (referred to as *relaxed* memory models). Section 2.1 discusses sequential consistency and motivates relaxed memory models. Section 2.2 discusses relaxed memory models. Section 2.3 discusses various formalisms used to define relaxed memory models. Section 2.4 summarizes performance studies of relaxed models. Section 2.5 describes other correctness criteria that are stronger than or similar to sequential consistency.

For our work on debugging on relaxed models (Chapter 8), the only other related work is by Gharachorloo and Gibbons [GhG91]. Since this work is related only to Chapter 8, we postpone its discussion to that chapter.

This chapter uses the terms *preceding* and *following* to indicate program order.

2.1. Sequential Consistency

Section 2.1.1 gives the original definition of sequential consistency by Lamport, Section 2.1.2 illustrates how this definition differs from cache coherence, Section 2.1.3 illustrates optimizations restricted by sequential consistency, Section 2.1.4 discusses various implementations of sequential consistency, and Section 2.1.5 examines why high performance implementations of sequential consistency are practically difficult and motivates relaxed memory models.

2.1.1. Definition

Sequential consistency was first defined by Lamport [Lam79] as follows.

> **Definition 2.1:** [A system 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.

Application of the definition requires a specific interpretation of the terms *operations* and *result*. Chapter 3 will formalize these notions; for now, we assume that *operations* refer to reads and writes of individual memory locations, and *result* refers to the values returned by all the read operations in the execution and possibly the final state of memory. Thus, sequential consistency requires that all memory operations should appear to have executed atomically in some total order, where (1) the total order is consistent with the program order of each process and (2) a read returns the value of the last write to the same location ordered before it by this total order.[2]

2.1.2. Sequential Consistency vs. Cache Coherence

Early bus-based systems with caches used cache coherence (or cache consistency) as the notion of correctness [CeF78, RuS84]. The definition, given by Censier and Feautrier, requires that a read should return the value deposited by the *latest* write to the same address [CeF78]. However, in systems with write buffers and general interconnection networks, the *latest* write to a location is not well-defined [DSB86]. Sequential consistency makes the notion of the *latest* write more precise; further, it also explicitly requires that operations of a single process appear to execute in the program order of the process. Similar to sequential consistency, Rudolph and Segall independently formalized the notion of returning the value of the *latest* write as requiring a total order on memory

[2] To model the final state of memory, a program can be considered to include reads of all memory locations after the rest of its execution.

operations where a read returns the value of the last write to the same location ordered before it by this total order [RuS84]. They do not, however, explicitly mention the requirement of program order.

For general interconnection networks, an often accepted definition of cache coherence requires that only writes to the same location appear to be seen by all processors in the same order [GLL90]. This thesis uses the term cache coherence, or simply coherence, to imply the above notion (a formal definition appears in Section 5.1.3).[3] This notion of coherence does not ensure sequential consistency since (informally) sequential consistency (a) requires all processors to see all writes to *all* locations in the same order [Col84-92, DuS90], and (b) requires all operations of a single process to be seen in program order. Figure 2.1 shows two program fragments that illustrate the difference between cache coherence and sequential consistency due to (a) above. In both programs, processors P_1 and P_2 update two locations and processors P_3 and P_4 read the values of those locations. In the executions depicted, P_3 and P_4 see the updates in opposite order for both programs. However, in the first program, the updates are to the same location and so the depicted execution is prohibited by cache coherence. In the second program, the updates are to different locations, and so the depicted execution does not violate cache coherence. Both executions, however, violate sequential consistency. The non-sequentially consistent executions of the programs in Figure 1.2 discussed in Chapter 1 (also repeated in figure 2.2 below) are examples where the program order requirement (condition (b) above) of sequential consistency is violated, but cache coherence is maintained. Thus, sequential consistency implies cache coherence, but cache coherence does not imply sequential consistency.

P_1	P_2	P_3	P_4
X = 1	X = 2	r1 = X	r3 = X
		r2 = X	r4 = X

Result: r1 = 1, r2 = 2, r3 = 2, r4 = 1

(a)

Initially X = Y = 0

P_1	P_2	P_3	P_4
X = 1	Y = 2	r1 = X	r3 = Y
		r2 = Y	r4 = X

Result: r1 = 1, r2 = 0, r3 = 2, r4 = 0

(b)

Figure 2.1. Sequential consistency vs. cache coherence.

Cache coherence does not allow the result in part (a), but allows the result in part (b). Sequential consistency does not allow either result.

2.1.3. Implementations That Disobey Sequential Consistency

Chapter 1 illustrated some of the implementation restrictions imposed by sequential consistency. We continue to use the examples of Chapter 1, repeated in figure 2.2, to illustrate more restrictions. As discussed in Chapter 1, systems that allow processors to reorder memory operations, or use write buffers to allow reads to bypass preceding writes, or overlap memory requests on the interconnection network can result in either or both of the non-sequentially consistent executions represented in figure 2.2 [DSB86, Lam79]. Analogously, compilers that reorder memory operations or allocate shared-memory locations in registers can violate sequential consistency [MPC89]. These violations are possible irrespective of whether systems have caches. In the presence of caches and a general interconnection network, sequential consistency imposes further constraints on hardware as follows.

Consider a cache-based system with a general interconnection network where there are no write buffers, processors issue memory operations in program order, and a processor waits for its memory request to be serviced

[3] There is a subtle issue concerning whether a write should be seen at all by a processor; the formal definition of Section 5.1.3 clarifies this issue.

by memory (or its cache) before issuing the following operation. Such a system can violate sequential consistency if a processor issues a memory operation before the effects of a preceding write have propagated to the caches of other processors. For example, for the code in part (a), assume that P_2 has X and Y initially in its cache. If P_2 returns the new value of *flag* while it still retains the old values of X and Y in its cache, then it is possible that P_2 will return these old values when it reads X and Y. Thus, typically, after a processor P_i executes a write to a line that may be present in other processor caches, the memory system needs to send invalidation or update messages to those caches and the caches need to acknowledge the receipt of these messages. Processor P_i can proceed with its subsequent operations only after the acknowledgements have been received (the acknowledgements may be collected directly by P_i or by the memory system). Thus, sequential consistency with a general interconnection network typically incurs additional network traffic and longer write latencies.

```
     Initially X = Y = flag = 0                 Initially X = Y = 0

  P1         P2                              P1         P2

  X = 14     while (flag != 1) {;}           X = 1      Y = 1

  Y = 26     r1 = X                          r1 = Y     r2 = X

  flag = 1   r2 = Y

       Result: r1 = r2 = 0                      Result: r1 = r2 = 0

              (a)                                      (b)
```

Figure 2.2. Violations of sequential consistency.

The program fragment in figure 2.1(b) illustrates even more network traffic due to sequential consistency as follows. Consider a cache-based system using an update-based coherence protocol [ArB86]. Assume all processors execute their memory operations in program order and one-at-a-time (waiting for acknowledgements as described above). It is still possible to violate sequential consistency if P_1's update reaches P_3 before P_2's update and P_2's update reaches P_4 before P_1's update. The violation in this case occurs because writes in the system are executed non-atomically: P_3 is allowed to return the value of the write of X before P_4 has seen the write, and P_4 is allowed to return the value of the write of Y before P_3 has seen the write. Sequential consistency requires writes to appear to execute atomically; i.e., no processor should see the value of a write until all processors have seen it. This typically forces update-based cache coherence protocols to employ two phases when updating caches due to a write. The first phase of the protocol (as also required for the previous examples) involves sending updates to the processor caches and receiving acknowledgements for these updates. In this phase, however, no processor can return the value of the updated location. The second phase begins when all the acknowledgement messages of the first phase have been received (by either the memory system or the writing processor), and involves sending back an acknowledgement to all the updated processor caches. A processor can use the updated value from its cache only after receiving an acknowledgement of the second phase. (For a system employing such a protocol, see [WiL92].)

Thus, sequential consistency, in general, restricts the use of several optimizations that can hide and tolerate memory latency and involves additional network traffic.

2.1.4. Implementations that Obey Sequential Consistency

In their seminal work, Dubois, Scheurich, and Briggs have analyzed the problem of imposing sequential consistency in many different types of hardware systems [DSB86, DSB88, DuS90, ScD87, Sch89]. To reason with non-atomic memory operations in a cache-based system, they define the notion of an operation being *performed with respect to a processor*, *performed*, and *globally performed*. Informally, a write operation is performed with respect to a processor when it has been observed by the processor, i.e., no future read of the processor to the same location can return the value of a previous write. A read operation is performed with respect to a processor when no future write of the processor can affect the value returned by the read. A write or a read operation is performed when it is performed with respect to all processors. A write is globally performed when it is performed. A read is globally performed when it is performed and when the write whose value it reads is performed. Thus, a write or a read is globally performed when the value written or read is observed by all processors.

Scheurich and Dubois state a sufficient condition for sequential consistency [ScD87, Sch89] for general systems (including systems with caches and systems with general interconnection networks that do not guarantee the ordering of any message pairs). The condition is satisfied if all processors issue their memory operations in program order, and no memory operation is issued by a processor until the preceding operation in program order has been globally performed. We note that for cache-based systems, the above condition is sufficient only if cache-coherence is also maintained (otherwise, the execution in figure 2.1(a) is possible). Thus, this condition requires a processor to execute its memory operations one-at-a-time, and writes to appear atomic.

Other researchers have proposed conditions for more restricted systems, which also obey the condition by Scheurich and Dubois. Lamport gives conditions for shared-memory systems with general interconnection networks, but no caches [Lam79]. Rudolph and Segall have developed two cache coherence protocols for bus-based systems. They formally prove the protocols obey their correctness criteria mentioned in Section 2.1.2; if we assume they implicitly considered processors that execute operations in program order, the protocols guarantee sequential consistency [RuS84]. The RP3 system [BMW85, PBG85] is a cache-based system, where processor memory communication is via an Omega network, but the management of cache coherence for shared writable variables is entrusted to the software. For sequential consistency, a process must wait for an acknowledgement from memory for its preceding miss on a shared variable before it can issue the following operation to another shared variable.

Several conditions for implementing sequential consistency that allow a processor to issue a shared-memory operation without waiting for its preceding operation to be globally performed have also been proposed. Shasha and Snir have proposed a scheme that uses static software support [ShS88]. Their scheme statically identifies a minimal set of pairs of memory operations within a process (called critical pairs), such that the only delays required are for the second element in each pair to wait for the first to complete. Although this work assumes that individual memory operations are atomic, we believe that if the term ‘‘complete’’ above is interpreted to mean ‘‘globally performed’’ and if cache coherence is maintained, then the results also extend to general systems where writes are not executed atomically. However, the algorithm used depends on detecting conflicting data operations (by the same processor and by different processors) at compile time and so its success depends on global data dependence analysis techniques, which may be pessimistic. Nevertheless, this work serves as a foundation for our framework to explore the design space of SCNF memory models (Chapter 7), and is discussed in more detail in Section 7.6.

Several implementations have been proposed that do not depend on software support to allow a processor to issue a memory operation while its preceding operation is not yet globally performed. Collier has proved that a system where all writes are performed with respect to all processors in the same order is (in most cases) equivalent to a system where all writes are executed atomically [Col84-92]. Thus, a system where writes obey the above condition and where all operations of a single processor are performed with respect to other processors in program order is sequentially consistent [Col84-92, DuS90]. Several researchers have proposed implementations that allow multiple outstanding writes of a processor, and essentially exploit Collier’s result as follows.

Collier himself proposes an implementation using a ring network where one processor in the network is used to serialize all writes [Col84-92]. In this system, a writing processor sends a write request to the serializing processor which then sends an update message along the ring. The writing processor can proceed with another operation as soon as it gets this update message, although the processors between the writing processor and the serializer have not yet seen the update. Landin et al. propose a scheme that employs the same principle for race-free (or acyclic) networks. Such a network preserves the ordering of certain transactions, much like the ring network in Collier’s work. Landin et al. identify a root node for every data item such that the sub-tree at the node contains all the copies of the data [LHH91]. They show that it is sufficient for a writing processor to wait only until the root node receives its request and sends an acknowledgement. Afek et al. propose a scheme called lazy caching for a cache-coherent bus-based system [ABM89, ABM93]. They show how a processor can proceed with a memory operation even if preceding writes are only buffered at the queues of other processors, but not necessarily performed with respect to those processors. Again, this scheme preserves sequential consistency by ensuring that all writes are seen in the same order by all processors. Scheurich proposes a similar scheme, but more informally, in his thesis [Sch89].

Other schemes that allow overlapping or reordering a processor's memory operations, but do not depend on specific network properties are as follows. Adve and Hill allow a writing processor to proceed once ownership of the requested line is obtained (other processors could still have a stale copy of the line); sequential consistency is maintained by ensuring that the effects of the subsequent operations of the writing processor are not made visible to any other processor until the write is globally performed [AdH90a]. Gharachorloo et al. describe the use of non-binding hardware prefetching and speculative execution for overlapped memory accesses with sequential consistency in cache-based systems [GGH91b].

All of the above schemes that allow a processor to overlap or reorder its memory accesses without software support, however, either require complex or restricted hardware (e.g., hardware prefetching and rollback for [GGH91b] and restricted networks for [Col84-92, LHH91]) or the gains are expected to be small (e.g., [AdH90a, LHH91]). Further, the optimizations of these schemes can be exploited by hardware (or the runtime system software), but cannot be exploited by compilers.

A relevant scheme from distributed message passing systems is Jefferson's virtual time scheme using the time warp mechanism [Jef85]. This approach allows a processor to execute its operations optimistically; the runtime environment detects any consistency violations and rolls back the relevant processes to a consistent state. The optimistic scheduling in [Jef85] has parallels with the speculative execution scheme of [GGH91b]; however, rollbacks in [GGH91b] are local while those in [Jef85] may have global effects. Similar ideas have also been used for parallelizing sequential programs written in mostly functional languages [Kni86, TiK88]. The parallel tasks are executed optimistically, using runtime (hardware or software) support to detect dependence violations and effect rollbacks.

Work related to compiler optimizations includes that by Shasha and Snir [ShS88] and Midkiff et al. [MPC89]. Although Shasha and Snir motivate their work for hardware optimizations (as discussed above), they suggest that compilers can apply optimizations that reorder memory operations to non-critical pairs of memory operations. The original algorithm given in [ShS88], however, is for straightline code that does not include branch operations. The paper suggests extensions, but no detailed algorithm is given for more general programs. Midkiff et al. have developed the work by Shasha and Snir to determine when a compiler can reorder program-ordered operations or schedule them in parallel without violating sequential consistency. While this work is applicable to general programs, it also depends on a global data dependence analysis.

Attiya and Welch [ACF93] and Lipton and Sandberg [LiS88] have derived bounds for the response time of memory operations on sequentially consistent implementations. (Attiya and Welch also derive bounds for the model of linearizability discussed in Section 2.5.)

Finally, Collier [Col84-92], Shasha and Snir [ShS88], and Landin et al. [LHH91] describe a sufficient system condition for sequential consistency in graph theoretic terms that we will use for the framework of Chapter 7.

2.1.5. Why Relaxed Memory Models?

We have seen that using uniprocessor optimizations such as write buffers and overlapped execution can violate sequential consistency. Therefore, general implementations of sequential consistency require a processor to execute its memory operations one at a time in program order, and writes are executed atomically. Sequential consistency, however, does not *require* memory operations to be executed as above; it simply requires that the execution *appear* as if operations were executed in program order and atomically. As discussed in Section 2.1.4, many schemes appear in the literature where memory operations of a processor are not executed one at a time or atomically. However, as also discussed in Section 2.1.4, these schemes either require restricting the network, or using complex hardware, or aggressive compiler technology. Even so, the performance gains possible are not known. For these reasons, there seems to be an emerging consensus in the industry to support relaxed memory models. Some commercial systems that support such models are the SPARC V8 and V9 architectures, the DEC Alpha, the Cray XMP, the CRAY T3D, and the Sequent Balance and Symmetry systems. However, before we discuss relaxed memory models, let us try to informally examine why it is difficult to exploit common optimizations with sequential consistency.

Although Section 2.1.3 showed examples where common optimizations violate sequential consistency, there are many other examples where such optimizations do not violate sequential consistency. Thus, it would be possible to implement sequential consistency with high performance if the system could determine the parts of the program where performance enhancing optimizations are safe and apply the optimizations selectively to only the safe parts. The reason that general, high performance sequentially consistent implementations have been difficult is that determining when it is safe to apply an optimization involves considering several interactions that are difficult to ascertain statically or track dynamically, as illustrated below.

Let us first focus on hardware. Consider, for example, the code fragment in figure 2.3. It shows processor P_1 executing two writes to locations *A* and *B*. When can P_1's second operation be executed before its first write completes (assuming a general interconnection network that does not preserve ordering of operations)? Clearly, there is a danger of violating sequential consistency if another processor reads *A* and *B* in the opposite order, as P_2 does in the figure. Thus, this indicates that to execute two operations in parallel, the system needs to keep track of any other processor that might access the same two locations while the two parallel operations are incomplete. However, this is not sufficient. Consider processors P_3 and P_4 in the figure. Processor P_3 writes to a third location *C* and then reads the initial value of *A*. Processor P_4 reads the new value of *B* and then reads *C*. For sequential consistency, P_4 should return the new value of *C*. However, if P_3 executed its read before the write to *C* completed, it is possible for P_4 to return the initial value of *C*. To prohibit this, the system needs to keep track of the interaction between three processors (P_1, P_3, and P_4) involving three memory locations (*A*, *B*, and *C*). Analogous code fragments that involve interactions between all processors in the system are possible.

Initially A = B = C = 0

P_1	P_2	P_3	P_4
A = 1	r1 = B	C = 1	r4 = B
B = 1	r2 = A	r3 = A	r5 = C

Assume r3 = 0, r4 = 1

Figure 2.3. Interactions to consider for sequential consistency.

Thus, to exploit the full parallelism present in the code of a single processor and yet retain sequential consistency, an implementation needs to keep track of several interactions, which is practically difficult. As we will see, relaxed models have the potential to exploit more parallelism with more practical hardware than sequential consistency. Our methodology for specifying relaxed memory models exploits the intra-process parallelism by requiring the programmer to give information to the system that allows the system to focus on fewer interactions, and give both sequential consistency and high performance with simpler hardware.

We also note that researchers have suggested techniques to tolerate the long memory latencies that are incurred with sequential consistency. These techniques include software or hardware based prefetching [GGH91b, MoG91] and the use of multiple contexts [ALK90, WeG89]. While these techniques help to improve the performance of sequentially consistent systems, they can also be used to aid the performance of relaxed systems [GHG91].

Observations similar to the above also apply to the compiler. Compilers need to perform a global data dependence analysis to determine when it might be safe to reorder memory operations without violating sequential consistency. Such analysis is often too conservative; the presence of procedures and pointers makes the analysis more difficult.

2.2. Relaxed Memory Models

This section briefly describes relaxed memory models proposed by other researchers. As indicated by figure 1.3 in Chapter 1, there are several such models that are related to each other in various ways. Consequently, determining a satisfactory classification to present all of these models and their relationships in a coherent manner has been difficult. We (somewhat arbitrarily) identify three memory models — weak ordering, processor consistency, and release consistency — that represent the keys ideas of many of the other models. We classify all the remaining models based on how close they are (in terms of optimizations they allow) to the above models. Each of the first three sub-sections below discusses one of the above three models and the other models that are closest to it. The fourth and final sub-section discusses the models that are considerably different from the above three models.

Our classification is sometimes arbitrary because many models combine ideas from others in different categories. For example, release consistency itself combines the optimizations of weak ordering and processor consistency.

This section presents the definitions of the various models in their original form. Subsequently, we have (jointly with others) proposed a uniform terminology and framework to express these models [GAG93]; this methodology is discussed briefly in Section 2.3 and in more detail in Chapter 7.

A goal of this section is to demonstrate the hardware-centric nature of many of the current models and the large variety of interfaces they present to the programmer. The rest of the thesis will show how a programmer-centric view can be used to specify and unify several of the following models, and return the programmer back to the familiar world of sequential consistency.

2.2.1. Weak Ordering And Related Models

The *weak ordering* model was proposed by Dubois, Scheurich and Briggs [DSB86, DuS90, Sch89]. It is based on the intuition that the ordering of memory operations is important only with respect to synchronization (as opposed to data) operations. It requires programmers to distinguish between data and synchronization operations, and requires the system to recognize this distinction. The model is defined as follows [DuS90]. (The definition implicitly assumes that uniprocessor dependences are obeyed.)

> **Definition 2.2:** In a multiprocessor system, memory accesses are *weakly ordered* if (1) accesses to global synchronizing variables are strongly ordered, (2) no access to a synchronizing variable is issued by a processor before all its [preceding] global data accesses have been globally performed, and (3) no access to global data is issued by a processor before its [preceding] access to a synchronizing variable has been globally performed.

Strong ordering referred to in (1) is defined as the hardware quality that guarantees sequential consistency without software aid [DuS90] (see Section 2.2.4). Weak ordering potentially provides higher performance than sequential consistency by allowing memory operations between consecutive synchronization operations of a processor to be executed in parallel and non-atomically. Synchronization operations act like local barriers that ensure all operations preceding the synchronization will globally perform before any operation following the synchronization is issued. Analogously, compilers can reorder memory operations between two consecutive synchronization operations, and can also allocate such locations in registers (with a little care [GAG93]). This is in contrast to the processor consistency based models of the next sub-section, where most of the allowed optimizations can be exploited only by the hardware (or runtime system software). The authors of weak ordering also give informal constraints for programmers. For example, in [DuS90], they state ‘‘if a shared variable is modified by one process and is accessed by other processes, then the access to the variable must be protected, and it is the responsibility of the programmer to ensure mutual exclusion of each access to the variable by using high-level language constructs such as critical sections.’’ Further, they state that critical sections should be implemented using hardware recognizable synchronization primitives.

The *Alpha* memory model is specified in terms of partial orders on memory operations and constraints on those partial orders [Sit92]. Although the specification methodology for the Alpha model is very different from that of weak ordering, the two models are semantically similar. The Alpha processor provides memory barrier (MB) instructions, and effectively ensures that two memory operations of a processor separated by an MB instruction appear to execute in program order. The MB instruction could be viewed as a synchronization memory operation of weak ordering; however, there is one important difference. As an example, a synchronization write operation of weak ordering to location *S* can be used to communicate the completion of its preceding operations to other processors that later read the location *S*. The MB instruction does not access a memory location and so by itself cannot be used for the above purpose; instead, it must always be used in conjunction with *another* memory operation. The semantics of a synchronization operation of weak ordering can be simulated by the Alpha by a memory operation that is immediately preceded and immediately followed by MB instructions.

The *RP3* system (described in Section 2.1.4) [BMW85, PBG85] provides a *fence* instruction similar to the MB instruction of Alpha, and can also be viewed as an implementation of weak ordering. The CRAY XMP also provides a similar instruction called the complete memory reference or CMR instruction [CRA82].

Bisiani, Nowatzyk, and Ravishankar propose a slightly relaxed version of weak ordering for the *PLUS* system [BNR89, BiR90]. The system allows a processor to proceed on read-modify-write synchronization operations before the requested value is returned, if it is known that the returned value does not ‘‘affect’’ the subsequent operations, or if the subsequent operations can be undone if the returned value necessitates it.

Attiya et al. describe the *hybrid consistency* model as a formalization of models such as weak ordering [AtF92]. They use the formalism of a simplified I/O automaton to define their models [LyT88] (briefly described in Section 2.3). The model classifies memory operations as strong and weak. Informally, the model requires that ‘‘(a) strong operations appear to be executed in some sequential order at all processes, and (b) if two operations are invoked by the same process and one of them is strong, then they appear to be executed in the order they were invoked’’ [AtF92]. Thus, strong operations are analogous to the synchronization operations of weak ordering, and weak operations are analogous to the data operations of weak ordering. However, as the authors point out in another paper, the formalism used above does not easily express aggressive optimizations such as non-blocking reads, out-of-order issue, and speculative execution [ACF93]. The authors have also proposed implementations of hybrid consistency, assuming each processor has a copy of the entire memory and that all writes can be broadcast atomically [AtF92, Fri93]. They also prove lower bounds on response times in hybrid consistent systems [AtF92, Fri93].[4]

2.2.2. Processor Consistency And Related Models

The *processor consistency* model was originally identified by Goodman [Goo89]. The model ensures that write operations of a *given* processor are observed in the same order by all processors in the system; however, writes by different processors may be seen in different orders by different processors. For example, processor consistency allows the execution of figure 2.1(b) where processors P_3 and P_4 observe the writes of P_1 and P_2 in different order. Goodman cites the VAX 8800 as an example of a commercial processor that guarantees processor consistency, but not sequential consistency. Goodman also hypothesizes that processor consistency is adequate for most programs.

Processor consistency was later refined by Gharachorloo et al. to explicitly specify the constraints on read operations as follows [GLL90]. (The definition implicitly assumes uniprocessor dependences and cache coherence.)

> **Definition 2.3:** [A system is processor consistent if] (1) before a load is allowed to perform with respect to any other processor, all [preceding] load accesses must be performed, and (2) before a store is allowed to perform with respect to any other processor, all [preceding] accesses (loads and stores) must be performed.

[4] To allow some of the optimizations disallowed by the original formalism of hybrid consistency, Attiya et al. have extended the formalism and redefined hybrid consistency [ACF93]. However, their new conditions seem to imply that all operations of a processor must appear to occur in ‘‘almost’’ program order, and so seem much stronger than the original definition. As one example, consider a program where a processor executes two strong writes to different locations, and another processor reads the same locations in opposite order using weak accesses. Overlapping or reordering weak (i.e., data) accesses for this program seems to violate the new definition of hybrid consistency.

Processor consistency can potentially provide higher performance than sequential consistency because it allows all reads to bypass all preceding writes of a processor (as long as the read and write are not to the same location), thereby allowing the aggressive use of write buffers in a system. Processor consistency also does not require writes to be atomic.

It was recently discovered that definition 2.3 does not allow a read to return the value of its own processor's preceding write from the write buffer until the write is serialized at memory, and a revision that eliminates the above constraint has been published [GGH93]. This revision involves modifying the definition of ‘‘perform,’’ and allows certain executions not allowed by definition 2.3. The relevant results of the rest of this thesis are applicable to both the above definition and the revised version of the processor consistency model.

The SPARC V8 architecture defined the *total store ordering* and *partial store ordering* models using a partial-order based formalism [SUN91]. These models are best understood when the system is viewed as a set of processors with a common memory module (as in figure 1.1), but where each processor has a write buffer. The following first describes the key features of total store ordering using the original formalism, and then compares it to processor consistency. The model ensures the existence of a partial order on all memory operations, where the partial order obeys a set of axioms. This partial order reflects the ordering of the operations at the common memory module and has the following key properties: (1) the order is total on all writes, (2) if a read precedes an operation in program order, then it also does so in the above partial order, (3) two program-ordered writes appear in program order in the above partial order, and (4) a read returns the value of the latest write that is either before it by the above partial order or that is before it by program order. The partial order also obeys uniprocessor dependencies, and does not order any write between the read and write of a read-modify-write.

Total store ordering is similar to definition 2.3 of processor consistency with the following exceptions concerning the atomicity of writes. Total store ordering requires that a write by a processor P_i should appear to be performed with respect to all other processors in the system at the same time (this is implied by (1) above); this makes total store ordering more strict than definition 2.3 (and the revised definition of processor consistency [GGH93]). Total store ordering allows a processor to read the value of its own write from its write buffer before the write is sent out to the memory system (implied by (4)). This makes total store ordering less strict than processor consistency when compared to definition 2.3, but not when compared to the revised definition [GGH93].

The partial store ordering model is similar to total store ordering except that it ensures two writes of the same processor to different locations will appear in program order only if they are separated by a special STBAR (store barrier) instruction [SUN91]. Thus, partial store ordering allows writes of a processor to be executed in parallel.

The authors of the SPARC V8 models state that programs that use single-writer-multiple-readers locks to protect all accesses to writable shared data will be portable across systems that obey total store ordering, partial store ordering, and sequential consistency (the lock routines need to be written to run correctly for partial store ordering).[5]

The SPARC V9 architecture introduces an additional model called *relaxed memory order* [SUN93]. This model guarantees uniprocessor dependencies. To enforce any other program order, it provides a range of barrier instructions (like STBAR) that order various pairs of memory operations. The model ensures that two memory

[5] The SPARC V8 manual also states that programs that use write locks to protect write operations but read without locking are portable across total store ordering, partial store ordering, and sequential consistency if writes of a processor are separated by the STBAR instruction. However, without further interpretation, this seems incorrect, as illustrated by the following program.

Initially X = Y = 0

P_1	P_2
lock L1	lock L2
A=1	B=1
r1=B	r2=A
unlock L1	unlock L2

The above program seems to obey the specified constraint; however, it is possible for the reads of A and B to return 0 on a total store ordering system, while such a result is not possible on a sequentially consistent system.

operations of a processor will appear to execute in program order if the operations are separated by the appropriate barrier instruction.

The *IBM 370* memory model [IBM83] can be viewed as a combination of a restricted total store ordering and the weak ordering models, although it was defined before either. Like the other models in this section, the 370 guarantees that operations of a single processor will appear to execute in program order except for a write followed by a read to a different location. Additionally, the 370 also seems to guarantee that writes will appear to execute atomically; thus, a read cannot return a value from its processor's write buffer until the write has been performed.[6] The 370 also provides serialization instructions similar to the MB instruction of Alpha. Before executing a serialization operation, a processor completes all operations that precede the serialization operation. Before executing any nonserialization operation, a processor completes all serialization operations that precede that non-serialization operation. Some of the serialization instructions also access memory; therefore, memory operations from such instructions are equivalent to synchronization operations of weak ordering.

Landin et al. propose various schemes for race-free interconnection networks that exploit the transaction orderings preserved by the network [LHH91]. One of the schemes is more aggressive than definition 2.3 of processor consistency, but preserves the semantics of definition 2.3 [GAG93]. Other schemes are variants of processor consistency.

Lipton and Sandberg have proposed the *pipelined RAM* model [LiS88, San90]. The definition assumes that each processor is associated with a copy of its memory. It states that a read returns the value of the accessed location in its processor's copy, and a write updates its processor's copy and sends out update messages for all other memory copies. (This is similar to Collier's system abstraction described in Section 2.3). The above definition does not place any constraints on the order in which memory operations are seen by different processors, and seems too weak for programmers (in the absence of hardware primitives more powerful than simple reads, writes, and read-modify-writes). However, discussions of the pipelined RAM system seem to imply that writes by the same processor are intended to be seen in the same order by all other processors, and each processor must execute its operations in program order [San90]. This seems to obey the aggressive version of processor consistency proposed by Landin et al. [LHH91], but is more aggressive than definition 2.3 since it allows writes of a single processor to be pipelined.

Ahamad et al. describe a formalization of Goodman's definition of processor consistency and the pipelined RAM using the formalism of execution histories (see Section 2.3), and discuss certain programming properties of processor consistency [ABJ93]. Some of the authors of this work have subsequently developed the work to include other models also, but do not provide formal proofs of equivalence between their formalizations and the original definitions of the models [KNA93].

2.2.3. Release Consistency And Related Models

The *release consistency* model combines and extends weak ordering and processor consistency [GLL90].[7] The model exploits differences between different types of synchronization operations to provide higher performance potential than either weak ordering or processor consistency [GLL90]. The model classifies all shared-memory operations into special and ordinary, all special operations into syncs and nsyncs, and all sync operations into releases and acquires. This classification of memory operations will be discussed in more detail in Chapter 6. Intuitively, ordinary accesses refer to synchronized data accesses; special accesses refer to synchronization or asynchronous data accesses; releases are write synchronizations used to order ordinary data operations (e.g., write due to an unlock); acquires are read synchronizations used to order ordinary data operations (e.g., read due to a lock); nsyncs are asynchronous data accesses or synchronization accesses not used to order ordinary data operations (e.g., set operation of a test&set). The model is defined as follows.

[6] We say the 370 *seems* to guarantee atomicity because the manual does not discuss non-atomic writes, but also does not explicitly mention that it enforces writes to be atomic. If writes in the 370 are intended to be non-atomic, then the 370 model can be viewed as a combination of processor consistency and weak ordering.

[7] The remarks about the revision of processor consistency in Section 2.2.2 apply to release consistency as well; specifically, the revised version uses the new definition of *perform*, but the revision does not affect our results.

Definition 2.4: [A system is *release consistent* if] (1) before an ordinary load or store access is allowed to perform with respect to any other processor, all [preceding] acquire accesses must be performed, and (2) before a release access is allowed to perform with respect to any other processor, all [preceding] ordinary load and store accesses must be performed, and (3) special accesses are processor consistent with respect to one another.

Compared to weak ordering, release consistency provides a higher performance potential mainly by allowing memory operations following a release synchronization to be overlapped with the operations that precede the release and the release itself, by allowing memory operations preceding synchronization to be overlapped with operations following the acquire and the acquire itself, by allowing a read synchronization operation to bypass previous write synchronization operations, and by allowing a write synchronization operation to be non-atomic. Compared to processor consistency, release consistency provides higher performance potential mainly by allowing all operations between consecutive synchronization operations to be overlapped.

A variant of release consistency where special accesses are sequentially consistent is also proposed [GLL90]. This model is abbreviated as *RCsc*, while the model with processor consistent special accesses is abbreviated as *RCpc*. Gharachorloo et al. identify a set of software constraints for which a system that obeys RCsc appears sequentially consistent. Programs that obey these constraints are called properly labeled (PL) programs.

Keleher et al. have proposed an implementation of release consistency, called *lazy release consistency*, for software-based shared virtual memory systems [KCZ92]. However, the implementation is a relaxation of release consistency since it does not require performing a memory access before the following release. Instead, it requires performing the access only at the next acquire to the same location as a following release, and only with respect to the acquiring processor. Petersen and Li have proposed alternative implementations of lazy release consistency using virtual memory support [PeL92a, PeL92b].

Bershad et al. [BeZ91] have proposed a relaxation of release consistency for the Midway software-based shared virtual memory system [BeZ91]. This model, called *entry consistency*, requires programmers to associate each data operation with a lock variable that should protect the operation. Like lazy release consistency, entry consistency does not perform an operation at the following release of a lock; instead, it postpones it to the time when another processor acquires the same lock and then performs the operation only with respect to the acquiring processor. In addition, entry consistency only guarantees that operations associated with the lock variable will be performed (with respect to the acquiring processor) at the time of the acquire. Bershad et al. have also proposed a combination of release consistency and entry consistency for Midway [BZS92]. The specification of entry consistency, however, seems to be incomplete; Chapter 7 discusses this issue further.

Gibbons and Merritt have proposed a relaxation of release consistency (RCsc) that is a generalization of entry consistency [GiM92]. This relaxation allows a programmer to associate a release with a subset of the preceding operations. A release now needs to wait for only its associated data operations (and all preceding synchronization operations) to be performed. Gibbons and Merritt state and prove the correctness of a formal set of constraints on programs for which such a system appears sequentially consistent. Further, this work also allows program order to be specified as a partial order per process, which can simplify expressing certain types of parallelism for the programmer. The specification in this work also seems to be incomplete and is discussed further in Chapter 7.

Dubois et al. propose *delayed consistency* for a release consistent system where an invalidation is buffered at the receiving processor until a subsequent acquire is executed by the processor [DWB91]. This scheme alleviates the problem of false sharing; however, it does not strictly implement release consistency since a processor can execute a release before its preceding operations are performed at all other processors. Another form of delayed consistency, referred to as loose coherence for Munin [BCZ90] and also discussed in [DWB91], buffers updates of a processor until the following synchronization operation of the same processor. This retains the memory model of release consistency.

Zucker proposes implementations for systems using software cache coherence [Zuc92]. He shows how these implementations do not obey the release consistency model, but obey the related programmer-centric model of data-race-free-1 that will be developed later in this thesis.

The *buffer consistency* model for the Beehive system differs from release consistency mainly by not requiring reads to be performed before the following release [LeR91, ShR91].

Note that the Alpha simulates the different types of operations of release consistency (RCpc) more efficiently than those of weak ordering. A write immediately preceded by an MB is a release, a read immediately followed by an MB is an acquire, and all other operations are data operations. Analogous observations hold for the RP3 system.

The *VAX* memory model differs from the above models by avoiding explicit restrictions on the order of execution of specific memory operations. Instead, the VAX architecture handbook states the following [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 structure. Seven instructions are provided to permit interlocked access to a control variable.’’ We have adopted the VAX approach for our work, as described in Chapter 3.

2.2.4. Other Relaxed Models

Collier has developed a general framework to express a variety of memory consistency models [Col84-92]. He uses an abstraction of a shared-memory system where each processor is associated with a copy of the shared-memory, a write consists of multiple sub-operations where each sub-operation updates one memory copy, and a read consists of a sub-operation that returns the value of the location in the memory copy of the reading processor. We adopt this abstraction for specifying system requirements for our work (Chapter 5). Based on the above abstraction, Collier defines architectures as sets of rules, where each rule is a restriction on the order in which certain memory sub-operations will appear to execute [Col84-92]. Using graph theoretic results, he has proved equivalences and inequivalences of several of these sets of rules or architectures. Examples of the rules include *program order* which states that all sub-operations of an operation will appear to execute before any sub-operation of the following operation by program order. The rule of *write order* imposes the above restriction only on pairs of program-ordered writes. The rule of *write order by store* imposes the above restriction only on sub-operations of program-ordered writes that occur in the issuing processor’s memory copy. Different combinations of the various rules represent different memory models, many of which are weaker than sequential consistency. The issue of programming most of these models, however, has not been addressed.

Scheurich proposes a model called *concurrent consistency* [Sch89], which is defined to behave like a sequentially consistent system for all programs except those that explicitly test for sequential consistency or take access timings into consideration. The programs have not been further characterized. Scheurich states sufficient conditions for obeying concurrent consistency. Informally, these are (1) memory operations of a given processor are issued and performed in program order and (2) when a write operation of processor P_i is observed by processor P_j, all memory operations previously observed by processor P_i should also have been observed by processor P_j. No practical implementations of concurrent consistency that are not sequentially consistent are given. The above sufficient conditions were originally stated as the definition of *strong ordering* and claimed to be sufficient for sequential consistency [DSB86]. However, there are programs for which the conditions violate sequential consistency [AdH90a] and strong ordering was later redefined as the hardware quality of a multiprocessor which guarantees sequential consistency without further software aid [DuS90, Sch89].

Hutto and Ahamad have hierarchically characterized various weak models [HuA90]. They introduce a model called *slow memory* that only ensures that (1) a read will always return a value that has been written to the same location, (2) once a processor reads the value of a write by another processor, it cannot read values of older writes to the same location by the same processor, and (3) local writes are immediately visible. Thus, slow memory is weaker than all the models discussed so far. Although the solutions of some problems have been demonstrated on slow memory, programming general problems on such a model seems difficult. Hutto et al. also introduce the model of *causal memory* [AHJ90, HuA90] which ensures that any write that causally precedes a read is observed by the read. Causal precedence is a transitive relation established by program order or due to a read that returns the value of a write.

Bitar has proposed the *weakest memory access order* [Bit92]. He proposes a weakest order for the processor to issue memory operations and for the memory modules to execute memory operations. However, the order proposed by Bitar cannot be the ‘‘weakest’’ order because Chapter 7 shows weaker sufficient constraints. For the memory module, Bitar uses the theory of Shasha and Snir to theoretically identify the minimal operations that memory needs to execute in a specific order. However, he does not say how hardware can identify these operations. Chapter 7 discusses his work in more detail.

The memory model problem of shared-memory systems has analogues in distributed message passing systems. The approach taken by the ISIS distributed system is particularly relevant to our work [BiJ87, Mul89]. The analogous term for sequential consistency in this context is *synchronous behavior*, which imposes strict constraints on the ordering of message deliveries. Instead of imposing such constraints all the time, the ISIS system provides the programmer with a variety of primitives that guarantee different ordering constraints. It is the responsibility of the programmer to use those primitives in the parts of the program where their use will not violate the appearance of synchrony. Such a system is called a *virtually synchronous* system.

2.3. Formalisms For Specifying Memory Models

This section briefly summarizes the various formalisms used to specify the memory models discussed in the previous section.

Dubois et al. developed the notions of ‘‘perform with respect to,’’ ‘‘perform,’’ and ‘‘globally perform’’ to capture the non-atomicity of memory as described in Section 2.1.4 [DSB86]. Several memory models have been described using this terminology including weak ordering, processor consistency, and release consistency. One disadvantage of this terminology is that it prescribes constraints in real time and so is more restrictive than the other models described below. The other formalisms overcome this disadvantage by requiring that the real time order of operations only *appear* to meet some constraints. The second disadvantage of this terminology is that it seems inadequate to capture a subtle interaction between reads returning values from writes in their write buffers and cache coherence. This interaction prompted the revisions in processor consistency and release consistency (Section 2.2) [GGH93], and is discussed in detail in Chapter 7.

The formalism used by Collier is related to that of Dubois et al., but is not time-based. Collier uses a system abstraction where each processor is associated with a copy of shared-memory, a write consists of sub-operations that each update one copy of memory, and a read consists of a sub-operation that returns the value of the location in its processor’s memory copy. Collier defines various rules (or memory models) that constrain the order in which various sub-operations should appear to execute. We adopt Collier’s approach to describe system constraints, and explain it further in Chapter 5. The advantage of Collier’s approach is that it is not time-based. It, however, suffers from the disadvantage of not being able to easily capture the interactions discussed for the ‘‘performs’’ formalism of Dubois et al. above. Chapter 7 explains, however, how Collier’s formalism is more attractive to reason about implementations of programmer-centric models than other formalisms that eliminate the above problem.

The formalism developed for the SPARC V8 architecture was illustrated in the description of the total store ordering model in the previous section. This formalism requires that there be a partial order on all memory operations, where the order obeys certain axioms [SUN91, SFC91]. The advantage of this formalism is that it captures the write buffer interaction described above; the disadvantage, however, is that it does not adequately model the non-atomicity of writes.

Gibbons et al. and Afek et al. [ABM89, ABM93, GMG91, GiM92] use the formalism of an I/O automaton developed by Lynch and Tuttle [LyT88]. This formalism expresses the system in terms of a non-deterministic automaton. The formalism is powerful and precise, and does not suffer from the disadvantages of the above formalisms. However, compared to other formalisms, it is less straightforward to translate a model in this formalism into an implementation, or to use such a model to verify the correctness of programs.

Others have used I/O automata, but express the model as constraints on the ‘‘execution histories of processors’’ generated by the automaton, rather than as the transition functions of the automaton itself [ABJ93, AtF92, KNA93]. The histories in the cited works indicate the state of memory as viewed by a processor, and thus represent systems similar to Collier’s abstraction. Consequently, they suffer from the same limitation as Collier’s abstraction.

The Alpha memory model is described in terms of constraints on a partial order which is composed of program order, and an order between operation pairs to the same location where at least one of the pair is a write. Hutto et al. describe causal memory using similar partial orders. These methods do not adequately model non-atomicity and can be considered to be special cases of Collier’s formalism.

We have, jointly with others, proposed a uniform methodology to describe many of the models of Section 2.2 [GAG93]. The formalism used is a combination of Collier's formalism and the formalism developed for the SPARC V8 models, and eliminates the disadvantages of the above formalisms. The advantages of our formalism are that it is not time-based, it captures the write buffer interaction indicated above, and it captures non-atomicity of writes. Using this formalism, we develop a uniform specification methodology for specifying many of the above memory models. The key advantage of the methodology is that it eliminates some unnecessary constraints for many of the models without changing the semantics of the model, thereby allowing more aggressive implementations. Further, it translates many models into a common framework making it easier to compare them and translate them into implementations. We use a similar specification methodology in Chapter 7, and postpone a further discussion of this work until that chapter.

2.4. Performance Benefits of Relaxed Memory Systems

This section discusses studies that analyze the performance benefits of relaxed memory systems. We focus on studies that examine systems with non-bus interconnection networks and that use real programs as workloads. All of the following studies evaluate runtime optimizations. To the best of our knowledge, there have not been any studies evaluating performance gains for the relaxed models due to compiler optimizations.

Gharachorloo et al. compared the models of sequential consistency, processor consistency, weak ordering, and release consistency (RCpc) for a DASH-like architecture [LLG90] using an execution-driven instruction-level simulator [GGH91a]. The study assumed an invalidation-based cache coherence protocol and blocking reads. It showed that the relaxed models can hide most of the write latency and can perform upto 41% better than sequential consistency. It also showed that with blocking reads, processor consistency performed as well as weak ordering and release consistency for most cases. This result is surprising because processor consistency only allows reads to bypass writes, whereas weak ordering and release consistency also allow writes to be pipelined. Thus, potentially with processor consistency, the write buffer could get full faster and the processor could have to block often on writes. However, with blocking reads, much of the latency of writes is hidden behind the reads, and the write buffer does not block often. In one case, processor consistency even does better than weak ordering because weak ordering requires its synchronization reads to stall for previous write operations, whereas in processor consistency, reads are never stalled for previous writes.

Zucker and Baer have studied sequential consistency, weak ordering, and release consistency (RCpc) with non-blocking reads, but where a processor is stalled when the value of an outstanding read is needed [Zuc92, ZuB92]. Again an execution-driven instruction-level simulator was used. The architecture studied was a dance-hall system with processors connected to memory through an omega network. The study involved examining the benefits of the relaxed models with varying cache and line sizes. It showed gains of upto 35% over SC, which were highly correlated with the hit rate, which in turn was highly correlated with the cache and line size.

Neither of the above studies fully exploited the non-blocking reads allowed by weak ordering and release consistency. Gharachorloo et al. [GGH92] examine non-blocking reads on a dynamically scheduled processor with a trace-driven simulation. The results indicate that weak ordering and release consistency can hide most of the read latency, but only with large window sizes (from 64 to 256).

Researchers at Rice University have performed two studies to compare release consistency and lazy release consistency implementations on a software-based shared virtual memory system [DKC93, KCZ92]. A key determinant of performance for such systems is the number of messages and amount of data exchanged. The first study is a trace-based simulation that examines the above metrics using invalidation and update based protocols [KCZ92]. The study shows that the number of messages and data transferred in the lazy release consistency schemes is consistently lower than for release consistency, especially for programs that show false sharing and frequent synchronization. The comparison between update and invalidate protocols is dependent on the type of sharing. The second study is a more detailed, execution-driven simulation that additionally examines a hybrid protocol for lazy release consistency that combines the advantages of the invalidate and update protocols. The study shows that the lazy hybrid protocol outperforms the other lazy protocols, and shows significant improvement for medium-grained applications. However, the results suggest that lock acquisition time is a serious limitation for shared virtual memory systems.

The above studies were for relatively large systems with non-bus interconnection, and used parallel programs as workloads. The following studies examine smaller bus-based systems and/or use probabilistic workloads.

Baer and Zucker have compared sequential consistency and weak ordering on a bus-based system [BaZ91] using simulation. Torellas and Hennessy use analytical models to compare the same models on a DASH-like system [ToH90]. Both studies find little improvement with weak ordering.

Petersen and Li use trace-based simulation to study sequential consistency and lazy release consistency implemented using virtual memory hardware support and compare it with a snooping protocol on a bus-based system (they also give a few results for a crossbar system) [PeL92a, PeL92b]. They show that the lazy release consistency scheme is competitive with the snooping scheme in terms of performance, and recommend it because of its implementation simplicity and flexibility.

Lee and Ramachandran use a probabilistic workload to compare buffer consistency with sequential consistency [LeR91]. Their study does not show significant performance gains with buffer consistency.

2.5. Correctness Criteria Stronger Than or Similar to Sequential Consistency

This section discusses correctness criteria that are stronger than, or similar to, sequential consistency.

A common correctness criteria for concurrent databases is *serializability* [BeG81, Pap86], which requires that transactions should appear as if they are executed one at a time in some sequential order. This is very similar to sequential consistency. However, database systems serialize the effects of entire transactions (which may be several reads and writes), while this thesis is concerned with the atomicity of individual reads and writes. The concept of a transaction may be extended to our case as well and database algorithms applied, but practical reasons limit the feasibility of this application. In particular, since database transactions may involve multiple disk accesses, and hence take much longer than simple memory operations, database systems can afford to incur a much larger overhead for concurrency control.

Herlihy and Wing propose a model of correctness, called *linearizability*, for systems with general concurrent objects [HeW90]. This model is proposed using the formalism of execution histories, where a history consists of invocation and response events for each operation (which may be a high-level operation involving several reads and writes of a concurrent object). The model assumes *well-formed* histories where each processor's operations are serialized (i.e., the response of an operation appears before the invocation of the next operation). Analogous to sequential consistency, linearizability requires all (possibly high-level) operations to appear to execute in a total order. It further requires that if an operation completely precedes another in the execution history, then that precedence should be preserved in the equivalent total order. A key difference between linearizability and sequential consistency is that linearizability is a *local* property; i.e., if all objects in the system are individually linearizable, then the system is linearizable. However, the practical difference between sequential consistency and linearizability is unclear. Specifically, we are mainly interested in aggressive processors that can overlap and reorder their memory operations.[8] For such a processor, a well-formed history does not correspond to the real time order of execution of the processor's memory operations; therefore, the significance of preserving the precedences of the processor's operations in the history is unclear. For processors that do not overlap or reorder their memory operations, we are not aware of any sequentially consistent implementations that are not also linearizable (assuming individual memory locations are the concurrent objects and the reads and writes of such locations are the operations).

Many theoretical models of shared-memory have been proposed to simplify the design and analysis of parallel algorithms. The *parallel random access machine (PRAM)* model is one of the most well-known [FoW78]. The PRAM model assumes that processors proceed in lockstep, executing a read, compute, and write function of their next instruction in every cycle. A PRAM obeys sequential consistency because all memory operations of a PRAM execution can be serialized consistent with program order. A sequentially consistent system, however, need not be a PRAM since it can allow its processors to execute asynchronously at different rates,

[8] We say a processor overlaps its operations if it issues an operation before its preceding operation is performed (according to the definition of Dubois et al.).

possibly resulting in executions that would not occur if processors proceeded in lockstep. Alternatively, the possible total orderings of memory operations corresponding to PRAM executions of a program are a subset of the possible total orderings for sequentially consistent executions. The PRAM model simplifies performance analysis of parallel algorithms. However, it is not a satisfactory model of correctness since requiring real machines to exhibit such synchronous behavior is far too expensive. Various, increasingly asynchronous relaxations of the PRAM have been proposed, but they still do not describe practical systems well enough to be considered as the model of correctness.

Chapter 3

A Programmer-Centric Methodology for Specifying Memory Models

Section 3.1 motivates the need for a uniform, programmer-centric methodology for specifying shared-memory models. Section 3.2 proposes one such methodology called the sequential consistency normal form (SCNF). Section 3.3 discusses some common concepts and terminology that will be used to define SCNF memory models in the rest of this thesis.

3.1. Motivation for a Programmer-Centric Methodology

Chapter 2 discussed several relaxed memory models that have been proposed. These models are specified using various methodologies and present a large variety of system constraints. To determine which model to use for a system and how to specify a memory model, we need to evaluate the various models and specification methodologies based on three criteria — programmability, performance, and portability — as follows. The first criterion of programmability determines the ease with which programmers can reason with the model. The second criterion of performance determines how fast programs can execute on practical implementations allowed by the model. In general, there is a trade-off between ease-of-programming and performance. For example, sequential consistency presents a simpler interface to programmers than the other models; however, sequential consistency is less amenable to high performance implementations than the other models. Since there is a tradeoff, it is likely that there is no ideal model and there will always exist a range of desirable models in the future. In such a scenario, it is important that the models be specified in a manner that makes it simple to port programs written for one model to another model. The third criterion, portability, for evaluating a memory model and its specification methodology determines the ease of porting programs between models specified with the given methodology. We refer to the three criteria of ***p**rogrammability, **p**ortability, and **p**erformance* as the ***3P** criteria* for evaluating memory models.

An ideal specification methodology for memory models should satisfy the 3P criteria as follows. For the first criterion of programmability, the specification should present a simple and familiar interface to the programmer. For the second criterion of performance, the specification should not impose any constraints on the system that are not necessary to meet the intended semantics of the model. For the third criterion of portability, the specification should camouflage the underlying system optimizations in a manner that presents a uniform interface to the programmer across a wide range of systems.

Most models and their specification methodologies discussed in the previous chapter lack many of the above properties. First, many of these models have been motivated by uniprocessor hardware optimizations and are defined almost in terms of those optimizations. This *hardware-centric* nature of the models makes them difficult to reason with for programmers. For example, processor consistency requires programmers to be aware that writes may not be executed atomically, effectively requiring programmers to be aware of caches in the underlying hardware. Second, as we will see later, some of the specifications prohibit implementations that would not otherwise violate the intended semantics of the model. For example, the authors of weak ordering mention that programmers should ‘‘ensure mutual exclusion for each access to [a shared writable] variable by using high-level language constructs such as critical sections,’’ where critical sections are implemented using hardware-recognizable synchronization primitives [DuS90]. There are potentially higher performance implementations not allowed by weak ordering that would also allow correct execution of such programs. Finally, in general, the relationships between the different models are not clear and so in many cases, porting a program written for one model to another model requires reasoning about the program again with the optimizations of the new model. For example, although total store ordering and processor consistency are very similar, the subtle differences between them can result in different behavior for certain programs. Thus, when moving a program written from total store ordering to processor consistency, determining whether the program will give acceptable results on processor consistency requires reasoning about the program again with the different semantics of processor consistency. Thus, the current models and their specification methodologies do not meet the 3P criteria adequately and there is a need for a more uniform and less hardware-centric methodology to specify memory models.

The rationale behind many of the models in Chapter 2 is that their optimizations allow most common programs to work correctly (at most with a few modifications to the program). For example, the authors of weak ordering informally cite the programs that use mutual exclusion through constructs such as critical sections [DuS90] and the authors of total store ordering and partial store ordering cite programs that use locks as described in Section 2.2.2 [SUN91]. In general, however, to formally ensure that a program is correct, further interpretation of the above informal statements may be required and the programmer has to reason with the specific optimizations of the model.

Given that many models informally state how programs can be written to ensure ‘‘correct’’ behavior, and given that these models are defined so that they appear ‘‘correct’’ to common programs, these models motivate the following potentially better approach for defining memory models. System designers could fix a uniform base model that captures the programmer’s notion of ‘‘correctness’’ and then define memory models entirely in terms of programs for which the model will appear like the base model. With this approach, programmers can always reason with the base model, and system designers are free to perform any optimizations that do not appear to violate the base model for the specified set of programs. An obvious choice for the base model is that of sequential consistency since it is a natural extension of the uniprocessor model and the most commonly assumed notion of correctness in multiprocessors. These observations lead to a specification methodology called the sequential consistency normal form or SCNF described below.

3.2. Sequential Consistency Normal Form (SCNF)

The sequential consistency normal form (SCNF) method of specifying memory models (earlier called weak ordering with respect to a synchronization model [AdH90b]) is as follows.

> **Definition 3.1:** A memory model is in *sequential consistency normal form (SCNF)* iff it guarantees sequential consistency to a set of formally-characterized programs. Further, the program characterization should not consider non-sequentially consistent executions.

Note that a memory model in SCNF does not provide any guarantees for programs that do not obey the described characterization. This property may be a disadvantage for programmers when debugging programs that do not yet obey the characterization; Chapter 8 addresses this issue. This property is an advantage for system designers because it allows them to give higher performance by *fully* exploiting the known characteristics of the allowed programs, irrespective of the effect on any other programs. The following chapters illustrate this advantage.

A memory model specification in SCNF implies a contract between programmers and system designers by which programmers are expected to adhere to certain rules and system designers are expected to give sequential consistency to programs that obey those rules. The rules for the SCNF models proposed in this thesis allow the use of all algorithms and programming paradigms developed for sequential consistency; the rules simply require programmers to provide certain information about the memory accesses in the program. This information helps the system determine whether an optimization is safe to apply to a certain memory operation. Consequently, programmers are free to provide conservative information; this simply prohibits the system from applying aggressive optimizations but preserves correctness. The programmer, however, may not specify incorrect information. This is because in the presence of incorrect information, the system may apply optimizations to unsafe parts of the program and violate sequential consistency.

Thus, an SCNF memory model is simply a method for the system to get help from the programmer to identify parts of the program where optimizations may be applied without violating sequential consistency. For the specific models in this thesis, the programmer always has the choice of not providing any help to the system: the cost is only in performance, not correctness. Therefore, rather than viewing this thesis as proposing different memory models, programmers can view this work as demonstrating how to get sequential consistency with high performance on different systems.

When viewed as different memory models, SCNF specifications address the 3P criteria discussed earlier as follows. For programmability, SCNF keeps the programmer's model simple by expressing it in terms of sequential consistency, the most frequently assumed model. For performance, SCNF provides tremendous flexibility to system designers by requiring only that the system appear sequentially consistent to all programs that obey the given characterization; no unnecessary conditions on how the system should create this appearance are given. For portability, SCNF models allow programmers to always program to a single interface (sequential consistency). If the program requirements of the models are easy to obey (as for the models in the subsequent chapters), then porting programs across different models is straightforward.

Besides addressing the 3P criteria adequately, the SCNF method of specification also provides a unified scheme for specifying and comparing memory models. A new memory model is useful only if the requirements it imposes on programs are easier to obey than previous models, or if it gives sequential consistency with higher performance than previous models. As we shall see later, the SCNF method does not make any distinction between implementations of RCsc (without nsyncs), lazy release consistency, the VAX, and weak ordering since the information for which these implementations are known to provide sequential consistency is the same. Such a unification is desirable because for many programmers (those that want sequential consistency), these systems are similar.

A few other models have used approaches similar to that of SCNF. The closest is the VAX model [Dec81]. The VAX model does not explicitly impose any system constraints. Instead, the following restriction is mentioned for programmers [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 structure [through seven interlock instructions].’’ The SCNF method generalizes the VAX approach and makes it more formal by explicitly stating how the system will behave when the programs obey the required constraints (i.e., sequentially consistent). The concurrent consistency model [Sch89] ensures sequential consistency to all programs except those ‘‘which explicitly test for sequential consistency or take access timings into consideration.’’ The SCNF method also generalizes the concurrent consistency approach, but requires that the constraints on programs be formally verifiable by reasoning about the behavior of the program on sequentially consistent systems (the constraints of concurrent consistency seem ambiguous and do not seem to fulfill this condition). Finally, the RCsc model, which was developed in parallel with the SCNF approach, is accompanied by a formal characterization of programs, called properly labeled or PL programs, for which RCsc gives sequential consistency [GLL90]. Thus, like the SCNF methodology, RCsc does not require programmers to deal with the system constraints of RCsc. However, in contrast to the SCNF methodology, RCsc imposes constraints on the system that are not necessary to guarantee sequential consistency to PL programs (Chapter 6).

The following discusses some potential disadvantages of specifying a memory model in SCNF and argues that these are more than offset by the advantages discussed above. First, the SCNF specification does not provide a good model for programmers of asynchronous algorithms that do not rely on sequential consistency for correctness [DeM88]. This is because the only guarantee given by SCNF models is that of sequential consistency and only when the program obeys the specified characterization. Enforcing sequential consistency for asynchronous algorithms could result in lower performance than is possible with other hardware-centric models. With the other models, the programmer has the option of reasoning directly with a lower-level system specification. However, our belief is that reasoning with such specifications is difficult and not likely to be used by many programmers. Restricting system flexibility so that it can be used by a few programmers seems undesirable. Furthermore, although SCNF models do not give any guarantees for programs that do not obey the relevant constraints, individual implementations will be reasonably well-behaved for such programs. We recommend that for maximum performance, programmers of asynchronous algorithms deal directly with low-level (hardware-centric) specifications of the system implementation. This would entail some risk of portability across other implementations, but would enable future faster implementations for the other, more common programs.

A second disadvantage of SCNF is regarding debugging of programs that do not yet obey the required constraints. SCNF models do not provide any guarantees for such programs. Again, individual implementations would be well-behaved for such programs and programmers could debug using the low-level (hardware-centric) specifications of the individual implementations. Although this is no worse than debugging with the hardware-centric models, it defeats the goal of providing a simple interface to the programmer. Possible alternatives are to either provide a sequentially consistent mode for debugging, or to specify models so that it is possible to provide compile time or runtime support to determine whether and where a program violates the specified constraints. Chapter 8 addresses this issue for two SCNF models.

A third possible disadvantage of the SCNF specification is that the software characterization for which a model guarantees sequential consistency may be complex, and it may be difficult to verify if a program obeys the characterization. However, if our hypothesis that programmers prefer to reason with sequential consistency is true, then this complexity is actually a limitation of the proposed model, rather than of the SCNF approach. Requiring that the model be specified in SCNF explicitly exposes this limitation.

Finally, it may be difficult for system designers to translate the SCNF specification to an implementation. The specifications of the hardware-centric models allow easier translations. However this disadvantage is offset by the design flexibility afforded by the SCNF specification. Further, an SCNF specification can always be accompanied by a more hardware-centric (and probably more restrictive) specification that is easier to translate into an implementation, although system designers will not be restricted to obeying that specification.

3.3. Concepts and Terminology For Defining SCNF Models

This section clarifies some intuitive concepts and terminology needed for defining SCNF models. Section 3.3.1 clarifies the dichotomy between the static and dynamic aspects of a system and Section 3.3.2 formalizes several key concepts.

3.3.1. Dichotomy Between Static and Dynamic Aspects of a System

Figure 3.1 illustrates a typical system consisting of a *static compile−time system* and a *dynamic runtime system*. Typically, a programmer writes a high-level language program that is transformed by the compile-time system (consisting of software such as the compiler) into a low-level language program that can be run by the runtime system. The runtime system (usually consisting of hardware and possibly some software) runs the low-level language program for some input data, performing the specified state changes in the program, and resulting in one of several possible executions of the program. To ensure that a program produces correct results on a system, programmers need to reason about all executions of their program that are possible on the system for the relevant inputs, the compile-time software designers need to reason about all possible executions of the high-level and low-level programs for all possible inputs, and the runtime system designers need only consider the current execution of the low-level program with the current input.

The SCNF memory models of this thesis require the programmer to reason about the behavior of memory operations of *all* sequentially consistent executions of a program (for relevant inputs). More specifically, many of the memory models we propose require that the *dynamic* memory operations in a sequentially consistent execution of the program be distinguished from one another on the basis of certain rules. The programmer, however, can only distinguish the *static* memory operations specified by the program. To use the models directly, programmers must be able to determine how to distinguish the static operations so that the dynamic operations will obey the required rules. It is possible to make this task easier by converting our low-level specifications into higher-level specifications that explicitly determine the above mapping for the programmer. We give a few examples of such high-level specifications, but mainly focus at the lowest level.

Requiring the programmer to provide information about the behavior of *all* possible sequentially consistent executions of a program may sound complex. However, writing a correct program in any case requires (at least conceptually) reasoning about all executions of the program possible on the given system; restricting this reasoning to sequentially consistent executions only decreases the number of executions that the programmer is required to reason about (compared to other hardware-centric models).

Finally, the static part of the system may itself consist of several sub-components resulting in several system levels. As mentioned in Chapter 1, each level needs to assume a memory model, and these models can be different. The software system at any static level needs to ensure that a program *Prog* that is correct for the model at its level gets transformed into a program that is correct for the model at the next level and that will produce the same results as the input program *Prog*.

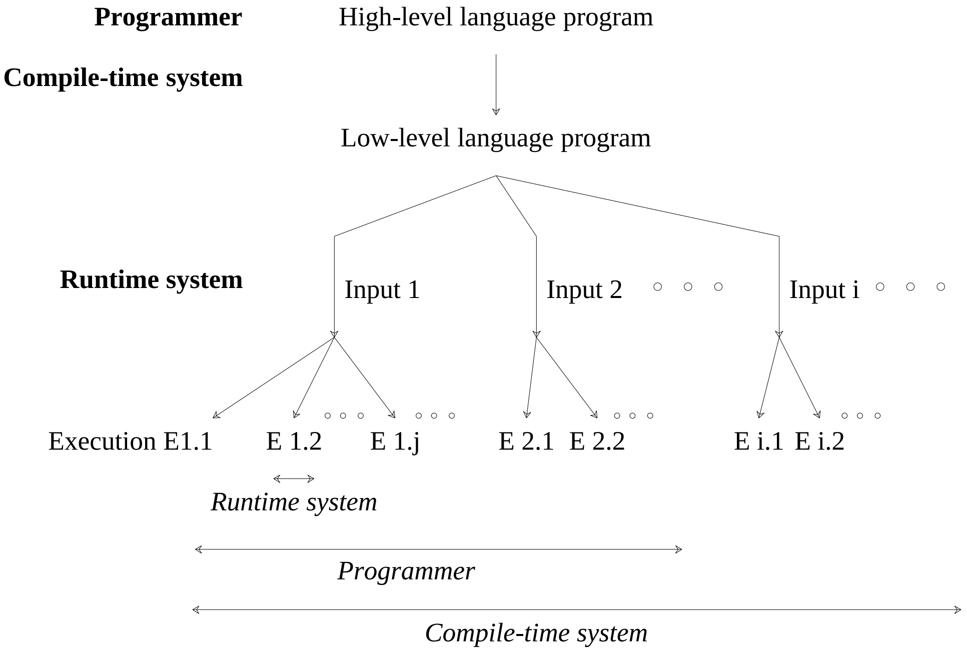


Figure 3.1. Dichotomy Between Static and Dynamic Aspects of a System.

Assume Input 1 and Input 2 are the only two inputs that the programmer will use.

3.3.2. Terminology for Defining SCNF Models

This section gives formal abstractions for the common concepts of a system, a program, a sequentially consistent execution of a program, and a sequentially consistent system. These concepts are fairly intuitive. Some of the formal definitions, however, are complex, but necessary for the formal interpretation and proofs of other definitions and results. Figures 3.2, 3.3, and 3.4 give the formal definitions. For the casual reader, the following provides a sufficient, informal summary.

A *system* is modeled as consisting of *n* processes, a set of atomically accessible shared-memory locations, and external input and output interfaces. The only externally alterable and observable states of a system are those of the external input and output interfaces respectively. A *program* specifies the initial state of a system and a sequence of (static) instructions that modify the state of the system. An instruction also specifies (static) shared-memory read and write operations, a partial ordering on these operations, and a partition of these operations into atomic subsets. This work allows an atomic subset to consist of either a single read, or a single write, or an atomic read-modify-write, where the read-modify-write consists of a read and a write to the same location and the read is ordered before the write. A program is run on a system, and the *result* of a run is the set of states acquired by the output interfaces during the run.

A *system is sequentially consistent* if the result of every run of a program on the system is the result of a sequentially consistent execution of the program, where a sequentially consistent execution and its result are defined as follows. A *sequentially consistent execution* of a multiprocessor program consists of a partially ordered set of (dynamic) instances of the instructions in the program, where the instances specify changes to the state of the system, and (dynamic) read and write operations on shared-memory. The partial order on the instances (and the corresponding order on the shared-memory operations) is called the *program order* (denoted $\xrightarrow{po}$). We adopt the approach of Gibbons and Merritt [GiM92] and allow the program order to be partial even over instruction instances of the same process. We will use the terms *preceding* and *following* to indicate program order. A sequentially consistent execution must satisfy two properties formalized in figure 3.4 and informally discussed

Definition 3.2: A *system* consists of *n* (*n* ≥ 1) processes with a set of state variables associated with each process that can be read and changed only by a run of a *program* (defined below), a set of shared-memory locations where each location can be atomically read and written only by a run of a program, a set of external input interfaces whose state can be read only by a run of a program and changed only by an external observer, and a set of external output interfaces whose state can be changed only by a run of a program and observed only by an external observer. The *result* of a run of a program on a system is the set of states of the external output interfaces during the run of the program.

Definition 3.3: A *program* specifies the initial state of a system and an ordered set of (static) instructions per process.

Each (static) instruction specifies zero or more of the following components:

(1) reads of the current state of some state variables of its process, reads of the states of some of the external input interfaces, and reads of the values of some shared-memory locations,

(2) changes to some of the state variables of its process and some of the external output interfaces, and writes to some shared-memory locations,

(3) one or more sets of *next* instructions of its process, or a *merge* instruction of its process, and

(4) a partial ordering for its shared-memory reads and writes, and a partition of its shared-memory reads and writes into atomic subsets. An atomic subset can be either a single read, or a single write, or an atomic read-modify-write, where the read-modify-write consists of a read and a write to the same location and the read is ordered before the write.

All of the above specifications (e.g., the shared-memory locations to be accessed) are functions of the values returned by various reads in (1), assuming no cyclic dependences within the functions (e.g., the location to be read cannot be a function of the value returned by the read.

Every process has at least one *initial* instruction and obeys Condition 3.7 in figure 3.3.

Definition 3.4: A (dynamic) *instance i′ of an instruction i* specifies the reads of some fixed state variables of its process, the writes of some fixed state variables of its process, and zero or more of the following components: states for some fixed external output interfaces, reads of some fixed external input interfaces and their values, reads of some fixed shared-memory locations and their values, and writes to some fixed shared-memory locations and the values, instances of one set of next instructions of *i* or an instance of a merge instruction of *i*, a partial ordering of its shared-memory reads and writes, and a partition of its shared-memory reads and writes into atomic subsets as specified in part (4) of definition 3.3.

Figure 3.2. Definition of a sequentially consistent execution and system (Cont.).

below.

In a sequentially consistent execution (informally), (1) the instruction instances of a single process must represent an execution from a correct uniprocessor given the values returned by the shared-memory reads of the sequentially consistent execution, and (2) shared-memory operations are executed in program order, and each read-modify-write is executed without any intervening writes to the same location. Property (1) is formalized as Condition 3.11 in figure 3.4. Property (2) is formalized by Condition 3.12 in figure 3.4. Specifically, Condition 3.12 requires a total order on all memory operations of a sequentially consistent execution, called the *execution order* (denoted by $\xrightarrow{xo}$), such that it is consistent with program order, a write to the same location as a read *R* and write *W* of a read-modify-write cannot be ordered between *R* and *W*, and a read returns the value of the last write

Definition 3.5: *Static operations*, *operations*, or *memory operations* refer to shared-memory reads and writes specified by instructions of a program; these terms include the specification of the shared-memory location to be accessed and the specification of the value to be written (if the operation is a write).

Dynamic operations refer to shared-memory reads and writes specified by instances of instructions; these terms include the shared-memory location accessed and the value written (if the operation is a write). The terms *operations* and *memory operations* are overloaded to also refer to dynamic operations.

Definition 3.6: A *sequentially consistent execution* of a program *Prog* consists of the following components.

(1) A (possibly infinite) set, *I*, of instances of the instructions of program *Prog* that obeys the uniprocessor correctness condition given in Condition 3.11 in figure 3.4.

(2) A set, *O*, of the memory operations specified by the instruction instances in *I*.

(3) A set, *V*, of the values returned by the read operations in *O*.

(4) A total order on the operations in *O*, called the execution order, denoted $\xrightarrow{\text{xo}}$, that obeys the execution order condition given in Condition 3.12 in figure 3.4.

Definition 3.7: *Program Order (* $\xrightarrow{po}$*):* For instruction instances i_1 and i_2 in an execution, $i_1 \xrightarrow{\text{po}} i_2$ for the execution iff (1) i_2 is the next instruction instance or merge instruction instance specified by i_1, or (2) there exists i_3 such that $i_1 \xrightarrow{\text{po}} i_3$ and $i_3 \xrightarrow{\text{po}} i_2$. For memory operations O_1 and O_2, $O_1 \xrightarrow{\text{po}} O_2$ iff O_1 and O_2 are from instruction instances i_1 and i_2 respectively, and either (1) $i_1 \xrightarrow{\text{po}} i_2$ or (2) $i_1 = i_2$ and O_1 is ordered before O_2 by i_1. The terms *preceding* and *following* will denote program order.

Condition 3.8: For any program, if i_1 and i_2 are instruction instances of the same process, and i_1 and i_2 are not related by program order, then i_1 or i_2 cannot read a state variable or memory location changed by the other instruction instance.

Definition 3.9: The *result* of an execution is the set of states of the external output interface specified by the instruction instances that specify such states.

Definition 3.10: A *system is sequentially consistent* if the result of every run of every program on the system is the result of a sequentially consistent execution of the program.

Figure 3.3. Definition of a sequentially consistent execution and system (Cont.).

Condition 3.11: *Uniprocessor correctness condition:* The set of instruction instances *I* of an execution must obey the following conditions.

(1) An instruction instance *i* is present in *I* iff either (a) *i* is the next instruction instance of another instruction instance present in *I*, or (b) *i* is an instance of an initial instruction of its process and *i* is the only instance of that initial instruction that is not the next instruction instance of another instruction instance in *I*. If *i* satisfies (b), then it is called an initial instruction instance.

(2) The program order relation is acyclic. Specifically, an instruction instance is the next instruction instance of at most one instruction instance, an initial instruction instance cannot be a next instruction instance of any instruction instance, and a merge instruction instance for an instruction instance *i* must be after *i* by program order.

(3) If *i* is an initial instruction instance, then the value returned by *i* for its read of a state variable of its process is the initial value of that state variable as specified by the program. If *i* is not an initial instruction instance, then the value returned by *i* for its read of a state variable of its process is the value specified by the last instruction instance ordered before *i* by program order that modified that state variable.

(4) The components other than the values of the state variables and memory locations read by an instruction instance *i* are determined by applying the functions specified by the instruction of *i* to the values of the state variables read by *i* and the values returned by the various reads of *i*.

(5) The value read from an external input interface by an instruction *i′* must be a value written by the external observer or an initial value.

Condition 3.12: *Execution order condition for sequentially consistent executions:*

The execution order, denoted $\xrightarrow{\text{xo}}$, of a sequentially consistent execution, *E*, is a total order on the set of memory operations, *O*, of *E* that obeys the following.

(1) A read in *O* returns the value of the write in *O* that is to the same location as the read and is the last such write ordered before the read by the execution order, if such a write exists. If there is no such write, then the read returns the initial value of the location.

(2) If two operations O_1 and O_2 in *O* are from an atomic read-modify-write, then there is no write to the same location as O_1 and O_2 ordered between O_1 and O_2 by the execution order.

(3) The number of operations ordered before any given operation by the execution order is finite.

(4) For operations O_1 and O_2 in *O*, if $O_1 \xrightarrow{\text{po}} O_2$, then $O_1 \xrightarrow{\text{xo}} O_2$.

Figure 3.4. Definition of a sequentially consistent execution and system.

to the same location ordered before it by the execution order (or the initial value if there is no such write). Intuitively, the execution order represents the order in which memory operations appear to execute in the system to the programmer.[9]

The *result* of a sequentially consistent execution is the set of states of the external output interface specified by the instruction instances of the execution. Note that given the uniprocessor correctness condition (Condition 3.11), it follows that the values returned by the various shared-memory reads in a sequentially consistent execution uniquely determine the result of the execution.[10] Our definition of result does not specify the sequence in which the external output interface states change; similarly, the formalism does not consider the order of changes to the external input interface. Although these notions may be too unconstrained for practical I/O, we deliberately avoid formalizing I/O any further since we are mainly concerned with the behavior of the memory system. It should be possible to independently incorporate a reasonable notion of I/O to the above formalism without affecting the results of this thesis.

Often, we will use the term *operations* or *memory operations* to refer to both static shared-memory operations (specified by the static instructions in a program) and to dynamic shared-memory operations (specified by the dynamic instruction instances of an execution); the reference made should be clear from the context.

Finally, the formalism of this section is to define SCNF models which will be used by programmers. Therefore, the formalism is centered around sequential consistency, the only view seen by programmers of SCNF systems. Chapter 5 extends the above formalism for designers of SCNF systems who need to reason with non-sequentially consistent behavior as well.

[9] We assume a relatively weak notion of atomic read-modify-writes to be consistent with other work [GAG93]. A stronger notion would require all operations of an atomic subset of an instruction to appear together in the execution order. Note, however, that even with our notion for read-modify-writes, for every sequentially consistent execution E_1, there is another sequentially consistent execution E_2 such that E_2 has the same result as E_1 and a read and a write of a read-modify-write appear together in the execution order of E_2.

[10] The observation that the values returned by shared-memory reads determine the result of an execution is key because it will allow us to determine the correctness of a general (non-sequentially consistent) system based on the values returned by its reads. We do not develop the formalism to describe general systems until Chapter 5; therefore, we cannot yet formally describe the meaning of the values of the reads of such a system. Nevertheless, Chapter 4 informally looks at general systems and executions, and informally uses the notion that the values of the shared-memory reads capture the result.

Chapter 4

An SCNF Memory Model: Data-Race-Free-0

This chapter defines an SCNF model — data-race-free-0 [AdH90b] — that is based on the intuition of weak ordering. It shows that although data-race-free-0 and weak ordering are based on the same intuition, the SCNF nature of data-race-free-0 provides a cleaner interface to the programmer than weak ordering, and allows for more implementations with potentially higher performance for some common programs than weak ordering.

Section 4.1 defines the data-race-free-0 model. Section 4.2 discusses programming with data-race-free-0, specifically comparing it to sequential consistency. Section 4.3 discusses implementations and the performance potential of data-race-free-0. Section 4.4 compares data-race-free-0 with weak ordering on the basis of programmability, portability, and performance.

A uniform description of the wide range of implementations allowed by data-race-free-0 requires developing a formalism to describe such implementations. The formalism and the details of the implementations are important for system designers, but their complexity is not necessary for users of data-race-free-0. For this reason, Section 4.3 in this chapter informally discusses only the key features of data-race-free-0 implementations. The next chapter develops the necessary formalism and uses it to describe data-race-free-0 implementations in more detail. Also, this chapter discusses implementations only for the runtime system since the additional performance gains with data-race-free-0 (as compared to weak ordering) are mainly significant for the runtime system; the next chapter will discuss compiler optimizations as well.

4.1. Definition of the Data-Race-Free-0 Memory Model

Section 4.1.1 and 4.1.2 respectively motivate and define the data-race-free-0 model (first presented in [AdH90b]). Section 4.1.3 describes the support needed in programming languages for data-race-free-0, and how the definition of data-race-free-0 can be adapted to the available support.

4.1.1. Motivation for Data-Race-Free-0

The problem of maintaining sequential consistency manifests itself when two or more processors interact through memory operations on common variables. In many cases, these interactions can be partitioned into memory operations that are used to order events, called *synchronization*, and the other more frequent operations that read and write *data*. A sequentially consistent system is usually forced to assume that each operation could be either a synchronization or a data operation, and so usually imposes a delay on every operation and executes every operation atomically. If, however, the system could distinguish between synchronization and data operations, then it may be possible to restrict actions for ensuring sequential consistency to only the synchronization operations, and achieve higher overall performance by completing data reads and writes faster.

For example, refer to figure 4.1. The figure shows process P_1 writing two data locations, *A* and *B*, which are later read by process P_2. The two processes use the location *Valid* to synchronize with each other so that P_2 reads the data only after P_1 updates it. Specifically, process P_1 uses its write of *Valid* to indicate to P_2 that previous data operations of P_1 are complete. Process P_2 uses its reads of *Valid* to determine when it can issue its data reads of *A* and *B*. A correct (and sequentially consistent) execution of such a program is one where P_2's reads of *A* and *B* will return the new values written by P_1. This correct answer can be obtained even if the data operations of each processor are done in parallel, as long as the data writes complete before the write of *Valid*, and the data reads do not begin before the read of *Valid* returns the new value of *Valid*.

The above considerations motivate an alternative programmer's model where the programmer explicitly indicates to the system whether an operation is data or synchronization. The system can then treat the two types of operations differently, typically executing the data operations faster for higher performance. The model of weak

```
P1                          P2

A = 100;                    while (Valid != 1) {;}

B = 200;                    ... = B;

Valid = 1;                  ... = A;
```

Figure 4.1. Motivation for Data-Race-Free-0.

ordering by Dubois, Scheurich and Briggs [DSB86, DSB88, Sch89] is such an alternative model, where memory operations of a processor are not guaranteed to execute in program order unless they are separated by synchronization, and writes may appear to execute at different times to different processors.

In general, to formally determine whether an operation should be distinguished as synchronization or data, the programmer of a weakly ordered system needs to reason with the conditions of weak ordering and deduce if distinguishing the operation in some way will result in a correct output from the program. Thus, to determine how to distinguish an operation on a weakly ordered system, the programmer must be aware that memory operations are not necessarily executed in program order and that a write may not be seen at the same time by all processors. A model based on the same motivation as weak ordering, but using the SCNF approach, can provide a higher level of abstraction where the programmer need only consider the sequentially consistent executions of programs, where memory operations are guaranteed to execute in program order and atomically. Reasoning with sequential consistency involves reasoning about fewer interactions between program instructions, and a familiar programming interface. The data-race-free-0 model is based on this motivation and defined next.

4.1.2. Definition of Data-Race-Free-0

The key feature of the definition of data-race-free-0 is the definition of when an operation should be distinguished as synchronization or data. This definition should be such that (1) programmers can distinguish the operations from reasoning about sequentially consistent executions of the program, and (2) the distinctions should allow for higher performance than with sequential consistency (without violating sequential consistency).

Intuitively, the distinguishing characteristic between data and synchronization operations is that in the execution order of any sequentially consistent execution, conflicting data operations are separated by conflicting synchronization operations. (Two operations conflict if they access the same location and at least one is a write [ShS88].) Referring back to figure 4.1, consider the write and read of *B*. In every sequentially consistent execution of this program, the write and read of *B* will always be separated by the write and read of *Valid* in the execution order. Data-race-free-0 requires that a program distinguish its operations as data and synchronization based on the above intuition. Operations distinguished as data can then be executed aggressively. Operations distinguished as synchronization will be executed conservatively; therefore, a programmer is free to (conservatively) distinguish an operation as synchronization. Distinguishing an operation conservatively will prohibit the high-performance optimizations with respect to this operation, but will result in correct (sequentially consistent) executions. A program distinguishes its operations correctly if sufficient operations are distinguished as synchronization. Intuitively, sufficient operations are distinguished as synchronization if all conflicting data operations are ordered by synchronization operations. A program that distinguishes its operations correctly is called a data-race-free-0 program and is formalized below. Note that any program can be converted into a data-race-free-0 program by simply distinguishing all operations as synchronization operations (this could be done by one single pragma or annotation).

There are several ways to formalize data-race-free-0 programs. The following method was used in our original definition of the data-race-free models [AdH90b, AdH93]. This method uses a happens-before relation to indicate when two memory operations are ordered by intervening synchronization operations. The relation is strongly related to the happened-before relation defined by Lamport for message passing systems [Lam78], and the approximate temporal order relation defined by Netzer and Miller for detecting races in shared-memory parallel programs [NeM90].

> **Definition 4.1:** Two memory operations *conflict* if they access the same location and at least one of them is a write [ShS88].
>
> **Definition 4.2:** *Synchronization-order-0 ($\xrightarrow{so0}$):* Let X and Y be two memory operations in a sequentially consistent execution. $X \xrightarrow{\text{so0}} Y$ iff X and Y conflict, X and Y are distinguished as synchronization operations to the system, and $X \xrightarrow{\text{xo}} Y$ in the execution.
>
> **Definition 4.3:** *Happens-before-0 ($\xrightarrow{hb0}$):* The happens-before-0 relation is defined on the memory operations of a sequentially consistent execution as the irreflexive transitive closure of program order and synchronization-order-0; i.e., ($\xrightarrow{\text{po}} \cup \xrightarrow{\text{so0}}$)+. (Program order or $\xrightarrow{\text{po}}$ was defined in figure 3.3.)
>
> **Definition 4.4:** *Race:* Two operations in a sequentially consistent execution form a *race* or a *data race* iff they conflict and they are not ordered by the happens-before-0 relation of the execution.
>
> **Definition 4.5:** *Data-Race-Free-0 Program:* A program is data-race-free-0 iff for every sequentially consistent execution of the program, all operations can be distinguished by the system as either data or synchronization, and there are no data races in the execution.
>
> **Definition 4.6:** *Data-Race-Free-0 Model:* A system obeys the data-race-free-0 memory model iff the result of every run of a data-race-free-0 program on the system is the result of a sequentially consistent execution of the program.

Figure 4.2 gives alternative definitions of a data race that can be used in definition 4.5. Appendix A shows that these definitions can be used in Definition 4.5 to define the same set of programs as data-race-free-0. (The proofs assume a very general set of mechanisms to distinguish operations; the assumptions are given in Section 5.1.3 in connection with implementations of SCNF models.) Alternative 1 is based on the work by Gharachorloo et al. [GLL90]. It defines a race in a sequentially consistent execution as two conflicting operations that occur consecutively in the execution order. We recommend this definition for programmers since it only involves the concept of the execution order of a sequentially consistent execution. Definition 4.4 above, however, makes it easier to determine how data-race-free-0 can be implemented (Chapter 5). Alternative 2 is analogous to definition 4.5, but makes more aggressive implementations explicit (Chapter 5). Alternative 3 is presented for completeness since a similar methodology will be used for later models. Alternative 4 is presented because it will be used later to prove the correctness of Alternative 2. Note that definition 4.4 above and alternative 2 assume that all memory operations are distinguished as either data or synchronization and then give a way to determine if an operation distinguished as data forms a data race. In contrast, alternatives 1, 3, and 4 define the notion of a race irrespective of how memory operations are distinguished.

It follows that a program is data-race-free-0 if for any sequentially consistent execution of the program, an operation that is involved in a race (as defined by any of the alternatives) is distinguished as a synchronization operation. Operations that are not involved in a race may be distinguished either as data or synchronization; however, the more operations that are distinguished as data, the greater the potential for higher performance. For example, figure 4.3 shows the code discussed earlier and depicts a sequentially consistent execution of the code. The notation *op,X* denotes a shared-memory operation *op* on location *X*. Assume the vertical order of the operations to be the execution order. The write and read on Valid occur consecutively; therefore, they form a race by Alternative 1 and should be distinguished as synchronization for this execution. The conflicting operations on A and B do not occur consecutively in this execution and so they do not form a race in this execution and can be distinguished as data for this execution. The figure shows the program order relation and the synchronization-order-0 relation assuming the above distinctions. It shows that all conflicting operations are ordered by the happens-before-0 relation and so by definition 4.4 also, there are no data races in the execution. Converting the program

Alternative 1.

Definition 4.7: Two operations in a sequentially consistent execution form a *race* iff they conflict and no other operation is ordered between them by the execution order of the execution.

Alternative 2.

Definition 4.8:

Synchronization-order-0+ ($\xrightarrow{\text{so0+}}$): Let X and Y be two memory operations in a sequentially consistent execution. $X \xrightarrow{\text{so0+}} Y$ iff $X \xrightarrow{\text{so0}} Y$, X is a write, and Y is a read that returns the value of X in the execution.

Happens-before-0+ ($\xrightarrow{hb0+}$): The happens-before-0+ relation is defined on the memory operations of a sequentially consistent execution as the irreflexive transitive closure of program order and synchronization-order-0+; i.e., ($\xrightarrow{\text{po}} \cup \xrightarrow{\text{so0+}}$)+

Definition 4.9: Two operations in a sequentially consistent execution form a race iff they conflict and they are not ordered by the happens-before-0+ relation for the execution.

Alternative 3.

Definition 4.10: *Conflict Order ($\xrightarrow{co}$):* Let X and Y be two memory operations in a sequentially consistent execution. $X \xrightarrow{\text{co}} Y$ iff X and Y conflict and X is ordered before Y by the execution order of the execution.

Definition 4.11: The *program/conflict graph* for a sequentially consistent execution E is a graph where the (dynamic) operations of the execution E are the vertices and there is an edge labeled $\xrightarrow{\text{po}}$ from vertex X to vertex Y in the graph iff $X \xrightarrow{\text{po}} Y$ for E, and there is an edge labeled $\xrightarrow{\text{co}}$ from vertex X to vertex Y in the graph iff $X \xrightarrow{\text{co}} Y$ for E.

Definition 4.12: Two operations in a sequentially consistent execution form a *race* iff they conflict and there is no path between the operations in the program/conflict graph of the execution such that the path has at least one program order arc.

Alternative 4.

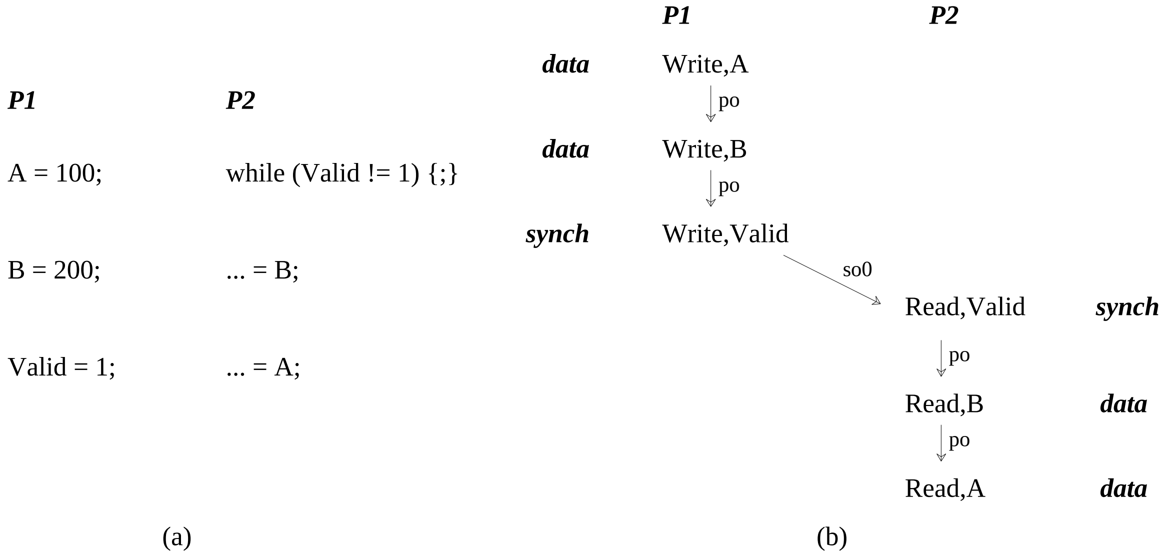
Definition 4.13: *Causal-Conflict Order ($\xrightarrow{cco}$):* Let X and Y be two memory operations in a sequentially consistent execution. $X \xrightarrow{\text{cco}} Y$ iff $X \xrightarrow{\text{co}} Y$, X is a write, and Y is a read that returns the value written by X in the execution.

Definition 4.14: Two operations in a sequentially consistent execution form a *race* iff they conflict and there is no path between the operations in the program/causal-conflict graph of the execution such that the path has at least one program order arc. The *program/causal-conflict graph* is similar to the program/conflict graph with conflict order replaced by causal-conflict order.

For each alternative, a data race is a race where at least one of the operations is distinguished as a data operation.

Figure 4.2. Alternative definitions of data races.

into a data-race-free-0 program, however, requires a mechanism in the programming language to distinguish memory operations so that all sequentially consistent executions of the program do not have data races.



```
P1              P2

A = 100;        while (Valid != 1) {;}

B = 200;        ... = B;

Valid = 1;      ... = A;
```

po = program order, so0 = synchronization-order-0

Figure 4.3. A sequentially consistent execution without data races.

In practice, the mechanisms provided in the programming language could make data-race-free-0 easier or more difficult to use for the programmer, and could affect the performance of a data-race-free-0 system. The next sub-section describes possible mechanisms for distinguishing memory operations, the resulting effect on data-race-free-0, and examples of data-race-free-0 programs that use these mechanisms.

4.1.3. Distinguishing Memory Operations

Data-race-free-0 requires a mechanism to distinguish memory operations for every component of the system mentioned in Chapter 1; e.g., the hardware and the compiler. Thus, the high-level programming language and the low-level machine language must provide support to make these distinctions. The data-race-free-0 model does not place any restrictions on the mechanisms that can be used, but the support provided can make the model easier or harder to reason with for programmers. The following first discusses four mechanisms for high-level programming languages and then discusses analogous mechanisms for machine languages. The section concludes with a note on the interaction between the high and low level mechanisms. Each of the mechanisms described could be used in isolation, or in combination with the other mechanisms. Each mechanism also leads to a higher-level specification of data-race-free-0 than definition 4.6. This section is primarily meant to illustrate the support possible in programming languages and the higher-level specifications for data-race-free-0 that can result depending on this support; this section should not be treated as an exhaustive description.

Mechanism 1 for high-level languages.

A simple mechanism is for the language to provide additional type declarations, called *sync_type* and *data_type*. For such a system, the definition of a data-race-free-0 program translates as follows. For any sequentially consistent execution of the program, if an operation accessing location *i* could form a race in that execution, then the corresponding static operation in the program should refer to a variable of type *sync_type*. For example, figure 4.4(a) shows the data-race-free-0 version of the program of figure 4.3. Figure 4.3 showed a sequentially consistent execution where the operations on Valid form a race; therefore, the variable Valid is declared as *sync_type*. The conflicting operations on A and B do not form a race in the execution shown and can never form a

```
P1                              P2

data_type int A,B;              data_type int A,B;
synch_type int Valid;           synch_type int Valid;


A = 100;                        while (Valid != 1) {;}


B = 200;                        ... = B;


Valid = 1;                      ... = A;
```

(a)

```
  P1                              P2

data = ON                       synchronization = ON
  A = 100;                        while (Valid != 1) {;}
                                data = ON
  B = 200;                        ... = B;
synchronization = ON
  Valid = 1;                      ... = A;
```

(b)

```
P1              P2              Implementation of Lock

Lock(L)         Lock(L)             while (Test&Set(L)) {;}

Access data     Access data     Implementation of Unlock

Unlock(L)       Unlock(L)           Unset(L)
```

(c)

Figure 4.4. Mechanisms for distinguishing operations and data-race-free-0 programs.

race in any sequentially consistent execution since they are always separated by the write and read of Valid. Therefore, the variable Valid can be declared as *data_type*. The above mechanism is simple to provide; however, it may not allow every operation to be distinguished arbitrarily. For example, it may not be possible to use the same variable for different types of operations in different parts of the program, and to declare the type of every field of every non-scalar data structure individually. Particular care needs to be taken for pointers, and parameters passed to procedures and library routines must be of consistent types. Specifically, if the parameter type assumed by the caller is *sync_type* or points to a *sync_type*, then the corresponding type assumed by the called procedure should be the same. For library routines, this might require supporting different flavors that accept different parameter types.

Also, it may be valuable to have default types. The advantage of having *sync_type* as default is that old programs will run correctly without any change, and conservative programmers need not do anything special to get correct programs. The disadvantage is that programmers who want any performance gains need to use special declarations; since most variables in programs are of type *data_type*, a default of *sync_type* will require many uses of the special declaration. Having *data_type* as default has complementary advantages and disadvantages.

Mechanism 2 for high-level languages.

Another simple mechanism is to provide annotations in the programming language. For example, an annotation such as *data = ON* (or *synchronization = ON*) would imply that memory operations from all (statically) subsequent instructions in the program until the (statically) next annotation in the program are data (or synchronization). With this mechanism, the definition of data-race-free-0 programs translates to the following. For any instruction *i* in the program, if a memory operation from an instance of *i* in a sequentially consistent execution could form a race in the execution, then the (statically) last annotation before the instruction in the program should be a *synchronization = ON* annotation. Figure 4.4(b) shows the corresponding data-race-free-0 program for the code in figure 4.3; an annotation of *synchronization = ON* precedes the operations on Valid and a single *data = ON* annotation suffices for the operations on A and B in each process. This mechanism does not have the disadvantages described for the previous mechanism; however, the simplest form of this mechanism provides only one annotation per instruction and so even if only one of the operations of an instruction is to be distinguished as synchronization, all operations from the instruction have to be distinguished similarly. Also, for calls to procedures and library routines, care must be taken to ensure that the annotations in the called routines are consistent with their usage assumed by the caller. Assuming the annotation of *data = ON* as the default, the number of annotations per program should be roughly proportional to the number of synchronization or race operations, which generally form a small part of the overall program size (assuming the programmer is not too conservative). The advantages and disadvantages of using either type of annotation as default are analogous to those for the defaults for mechanism 1.

Mechanism 3 for high-level languages.

The third mechanism exploits the use of synchronization library routines for common synchronization constructs (e.g., locks, barriers). Programmers often restrict synchronization in the program to calls to such routines. A useful mechanism for such a system would be one where operations from these library routines can be considered as synchronization while others as data. The definition of data-race-free-0 for a system that provides synchronization routines for locks, unlocks, and barriers would be as follows. Treat the lock, unlock, and barrier routines as single atomic operations with the appropriate semantics; assume the locks and unlocks specify a lock location to be locked or unlocked, and the barriers specify a barrier location. Define the synchronization-order-0 relation for a sequentially consistent execution as ordering locks and unlocks to the same location, and barriers to the same location in the order in which they happen in the execution order of the execution. Define the happens-before-0 relation and data-race-free-0 programs as before. The advantage of the above method is that it does not need any additional mechanisms for high-level programmers. The writers of the library routines still need some mechanism to make the different memory operation types evident to lower levels of the system. The disadvantage is that synchronization operations are restricted to only the routines provided by the system. As an example, consider figure 4.4(c) which shows two processors accessing data in a critical section implemented using library routines called lock and unlock. If the system specifies that the routines can be interpreted as synchronization, then the program is data-race-free-0. A low-level implementation of the lock and unlock routines is shown next to the high-level code. The lock is implemented using a read-modify-write instruction called Test&Set that atomically reads (tests) the lock variable and writes (sets) it to the value 1 (see Section 3.3.2 for the definition of a read-modify-write). The unlock is implemented using an Unset instruction that clears the accessed location to 0. This implementation is correct if hardware recognizes operations from the Test&Set and Unset instructions as synchronization operations.

Mechanism 4 for high-level languages.

The above mechanisms require programmers to consider low-level operations like individual reads and writes, locks, and barriers. Many languages specify higher level paradigms for synchronization and restrict programmers to using these paradigms; e.g., monitors, fork-joins, task rendezvous, doall loops. Such paradigms are naturally suited to data-race-free-0. For example, if the only way to express parallel tasks and synchronization is through doall loops, then the data-race-free-0 requirement simply becomes that if two iterations of a given doall loop access the same variable, then the variable cannot be written in either iteration. Similarly, for a language that provides only monitors, the data-race-free-0 restriction is that shared-memory accesses should be made only in the appropriate monitor. (The compiler ensures that the doalls and the monitors are translated appropriately so that the low-level program is correct.) Programmers using the above languages already obey the specified restrictions of data-race-free-0 simply because the above restrictions make parallelism easier to manage than unrestricted sequential consistency. In these cases, the implicit model for the high-level programmers was already data-race-free-0 and compilers of the above languages could already perform the optimizations of data-race-0. Specifying the system model explicitly as data-race-free-0, however, allows all parts of the system to perform the high-performance optimizations possible for data-race-free-0 programs.

Mechanisms for machine languages.

Mechanisms for distinguishing memory operations in the machine language are analogous to those for high-level programming languages. For example, memory operations can be distinguished based on the locations accessed. Alternately, analogous to the annotations of high-level languages, all operations could be assumed to be data by default unless their instructions were preceded by special prefix instructions; e.g., memory barrier instructions [SUN91, Sit92]. Another general mechanism that does not require adding extra instructions is to provide an extra bit for the op codes of all instructions that access memory to indicate whether the operations of the instructions are synchronization or data. An adequate number of synchronization and data instructions would need to be provided. The Am29000 processor, for example, provides *option bits* in its load and store instructions that could be used for this purpose. Another possibility is to use any unused high bits of the virtual memory address space to indicate whether the access is synchronization or data. A more restrictive version of the above is motivated by current hardware design that often provides a few special instructions to aid synchronization; e.g., Test&Set and Unset instructions in figure 4.4(c). Thus, all synchronization operations could be restricted to only these instructions. Finally, like the special library routines in programming languages, hardware may provide special instructions for fast high-level synchronization that do not use memory reads and writes; e.g., barriers of Thinking Machines CM-5 and locking in Sequent Balance. The synchronization-order and happens-before relations could then be modified to use these hardware operations for ordering, rather than memory reads and writes.

Interaction between mechanisms for different levels.

The mechanisms for distinguishing memory operations provided at the different levels of the system need to be compatible with each other. Specifically, the compiler needs to translate a high-level data-race-free-0 program into a low-level data-race-free-0 program. Therefore, if the low-level does not provide mechanisms to distinguish any general memory operation as synchronization or data, then the high-level should not provide such a mechanism either or the compiler should be able to determine an appropriate mapping for transforming the programs correctly. Similarly, if the high-level languages restrict synchronization operations to be certain types of operations, then there may be no need for the hardware to provide general mechanisms.

4.2. Programming With Data-Race-Free-0

This section discusses programming with data-race-free-0, specifically compared to sequential consistency. A comparison with respect to weak ordering appears in Section 4.4.

Data-race-free-0 allows programmers to program with the familiar interface of sequential consistency, as long as programmers obey certain rules. Therefore, the ease of programming with data-race-free-0 depends on how easy it is to obey the rules of data-race-free-0. This in turn depends on the support provided in the specific programming language for distinguishing memory operations. To isolate the effects of specific programming language implementations from the data-race-free-0 model, below we first consider programming with a general language where any memory operation in the program can be distinguished as synchronization or data. We consider more restrictive languages later in the section.

In a system where any memory operation can be distinguished as data or synchronization, data-race-free-0 does not impose any restrictions on how programs are written (compared to sequential consistency). Any program that is correct for sequential consistency can be run on a data-race-free-0 system and will generate the same instructions and memory operations as on a sequentially consistent system. Thus, programmers can use any synchronization mechanism, algorithm, or programming paradigm that they are familiar with. The only constraint data-race-free-0 imposes is that memory operations of this program should be distinguished as synchronization or data such that there are no data races in any sequentially consistent execution of the program.

Figure 4.5 captures the programmer's model for such a system. The programmer initially writes a program for a sequentially consistent system. Then for every static operation specified in the program, the programmer must determine whether any dynamic instance of the static operation can form a race in any sequentially consistent execution. The simplest definition of a race to use is definition 4.7 in figure 4.2.[11] If the programmer can determine that the operation will never form a race in any sequentially consistent execution, then the operation should be distinguished as data. If the programmer can determine that the operation will form a race in some sequentially consistent execution, then the operation should be distinguished as synchronization. If the programmer does not know whether the operation forms a race or not, or does not want to do the work to know, then the programmer can distinguish the operation as synchronization. The presence of this ''don't-know'' option is central to the data-race-free-0 model and the SCNF models of the subsequent chapters. It ensures that for writing a correct program, the programmer does not have to expend any more effort than on a sequentially consistent system; all memory operations can be conservatively distinguished as synchronization. It is only for higher performance that the programmer needs to expend more effort; furthermore, this effort can be spent incrementally by examining operations in the most performance critical parts of the program first and examining other parts later.

1. Write program assuming sequential consistency

2. For every memory operation specified in the program do:

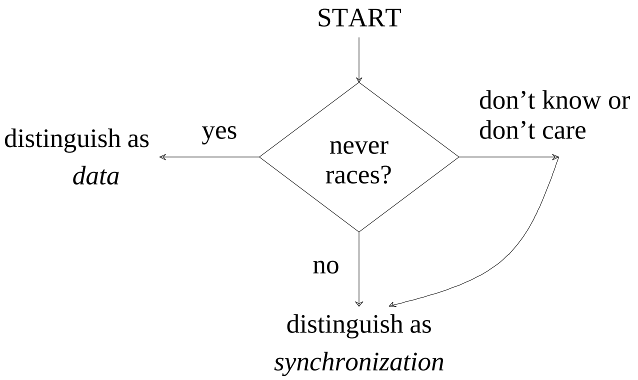


Figure 4.5. Programming with data-race-free-0.

[11] Definitions other than definition 4.7 can be used for figure 4.5 as well. Definitions 4.4 and 4.9 assume that memory operations are already distinguished to determine when there is a race. To use these definitions, the programmer should assume all operations except the considered operation are distinguished as synchronization, the considered operation is distinguished as data, and then determine if the considered operation will form a race by the above definitions. Henceforth, when we say an *operation forms a race*, we mean it in the above sense for the above definitions.

Although programmers have the option of being conservative, we can expect that they will not invoke this option too often. This is because the knowledge of whether an operation will be involved in a race or not is critical to the correct operation of the program even on a sequentially consistent system. A more important problem, however, is that programmers might believe they have adequate information to distinguish an operation correctly, but in reality there is a bug in the program that makes this information incorrect. This may result in non-data-race-free-0 programs, the system will no longer appear sequentially consistent, and the programmer can no longer use sequential consistency while debugging. In general, data-race-free-0 systems require additional support for debugging. Chapter 8 discusses how techniques for detecting data races on sequentially consistent systems can be used to detect data races on data-race-free-0 systems in a manner that allows the programmer to continue reasoning with sequential consistency. Gharachorloo et al. also propose hardware support that informs the programmer of when there might be data races in the program [GhG91]; however, further support is required to allow programmers to reason with sequential consistency to eliminate these data races.

Note that although incorrect distinctions made by the programmer can lead to non-sequentially consistent behavior, the responsible bug in the program will usually also be a bug on a sequentially consistent system for the following reason. A programmer makes an incorrect distinction only when a race operation is distinguished as data. This is an indication that the programmer expects two conflicting operations to be ordered by synchronization on a sequentially consistent system, but in reality they are not. This implies a misunderstanding of the program behavior on sequentially consistent systems and a bug that would usually manifest itself on a sequentially consistent system also; there is abundant literature on detecting data races on sequentially consistent systems motivated by the observation that data races are often symptoms of bugs in a program (see Chapter 8). Thus, program bugs due to the data-race-free-0 model are also often likely to be bugs when assuming the sequential consistency model.

The above discussion assumes programming languages that allow any operation to be distinguished as data or synchronization. The discussion also applies to languages where distinctions are made at the level of variables or instructions (such as described by mechanisms 1 and 2 in the previous section). For such languages, it may not be possible to always distinguish an operation as data if other operations to the same variable or in the same instruction need to be distinguished as synchronization. Although this could affect performance, such languages do not restrict programming techniques and the ease of programming is comparable to the most general language as discussed above.

Systems where only limited types of operations can be distinguished as synchronization (such as described by mechanisms 3 and 4) sacrifice programmers' flexibility. This is not a limitation of the data-race-free-0 model, but is a limitation of the specific implementation of data-race-free-0. Restrictions by systems such as those described by mechanism 4, however, are widely accepted as desirable programming paradigms even with sequentially consistent systems. Since those restrictions are at a higher level and since most high-level programmers already obey those restrictions, it follows that data-race-free-0 does not impose any additional constraints on programmers of such systems (but allows the hardware to exploit the constraints already assumed by high-level programmers).

To test our arguments above, we examined a set of programs from the SPLASH benchmark suite (BarnesHut, MP3D, and LocusRoute) [SWG92], and a program (Polyroots) for computing the roots of a large degree polynomial with integer coefficients of arbitrary precision [NaT92]. The programs are written in C and use the Argonne National Laboratory macro package to provide most synchronization and sharing primitives. Although most synchronization is explicitly identifiable through the macros, some programs also use normal reads and writes for synchronization (as in figure 4.1), and some use operations that race for asynchronous data accesses. We (manually) added the annotations indicated in mechanism 2 of the previous section (as illustrated by figure 4.4(b)) to identify the above accesses as synchronization. We assumed the default annotation as *data = ON*. We found that it was usually not difficult to determine whether an operation might race and the don't-know option was not invoked. The number of annotations added to the programs varied. The following gives the number of additional *synchronization = ON* annotations; an equal number of following *data = ON* annotations are required. Polyroots did not need any additional annotations, BarnesHut required five additional synchronization annotations, and MP3D with the locking option required four additional synchronization annotations (for the non-locking option, the locks within ifdef statements indicate the points where synchronization annotations are needed). LocusRoute uses many asynchronous data accesses, and required about 40 synchronization annotations. Thus, for the programs that do not access data asynchronously, most races were already explicit calls to library routines.

4.3. Implementations of Data-Race-Free-0

One aspect of implementing data-race-free-0, support for distinguishing memory operations, has already been discussed. This section discusses runtime system optimizations allowed by data-race-free-0. (Chapter 5 discusses compiler optimizations.) This section motivates four possible implementations of data-race-free-0. The first implementation is allowed by weak ordering, while the other three implementations are not allowed by weak ordering. The second implementation is based on release consistency (RCsc). As discussed earlier, this section is intended to convey only the key intuition for the four implementations; therefore, it uses several terms (e.g., execution of a program on general hardware) in the intuitive sense. Formalizations of all the terms, and details of the implementation proposals appear in the next chapter. Recall that we use the terms *preceding* and *following* to relate to program order.

Figure 4.6(b) depicts an execution of the program in Figure 4.6(a), where processor P_1 executes data operations, including a write of *A*, and then does a synchronization write on *Valid*. Processor P_2 executes a synchronization read on *Valid* until it returns the value written by P_1, and then executes data operations, including a read of *A*. For the execution to be sequentially consistent, the only constraint is that P_2's read of *A* should return the value written by P_1's write of *A*. The following first shows how weak ordering meets this requirement, and then shows how data-race-free-0 allows more aggressive implementations. (This work is based on material in [AdH90b, AdH93].)

Weak ordering requires a processor to stall on every synchronization operation until its preceding operations complete. Thus, in figure 4.6(b), weak ordering would prohibit P_1 from issuing its write of *Valid*, and all following operations, until the data write on *A* completes. Similarly, P_2 is prohibited from issuing its read of *Valid*, or any other operation following the read, until all the operations preceding the read complete.

A more aggressive implementation (based on release consistency) requires P_1 to delay only its write of *Valid* until its preceding operations complete; data operations following the write of *Valid* can execute even while preceding data operations are not complete. This allows P_1 to progress faster than with weak ordering, but does not affect when its write of *Valid* is executed, and consequently, when P_2 sees that write.

The delay on the write of *Valid* imposed by the two techniques described above, however, is not necessary to maintain sequential consistency (as also observed by Zucker [Zuc91, Zuc92]). We discuss two aggressive implementation proposals for data-race-free-0 that do not impose the above delay, and are not allowed by weak ordering (and release consistency). The first of these implementations maintains sequential consistency by ensuring that when P_2 issues its read of *Valid*, P_1 will not allow the read to return the updated value of *Valid* until P_1's operations preceding the write of *Valid* complete [AdH90b]. This implies that P_1 need never delay its memory operations; specifically, its write of *Valid* executes earlier than with the implementations discussed above. This could also make P_2 progress faster since the earlier execution of the write of *Valid* implies that P_2's read of *Valid* could succeed earlier than with weak ordering.

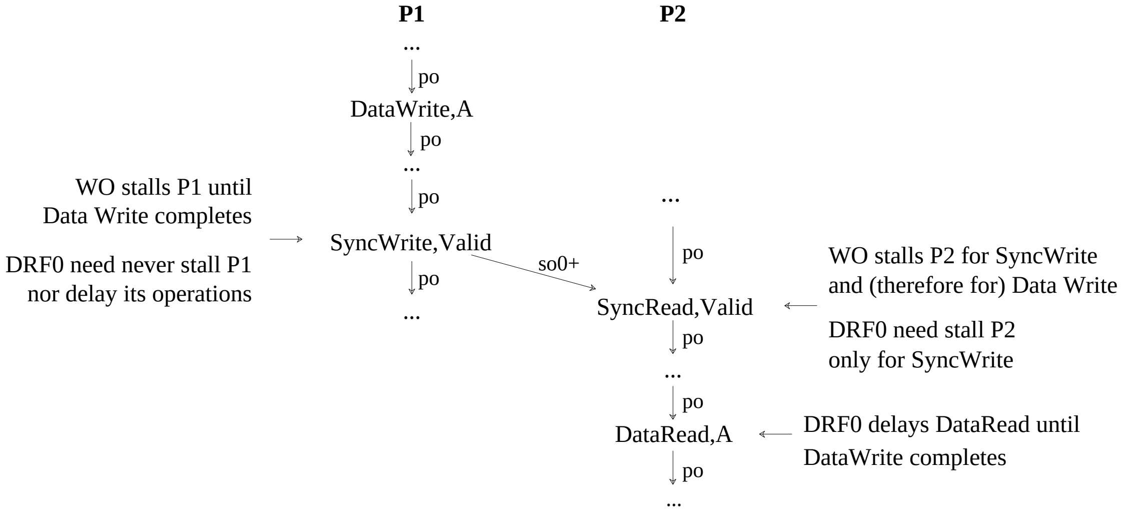
The final implementation proposal [AdH93] is even more aggressive. This proposal does not require P_2's reads on *Valid* to wait for P_1's data operations, thus allowing P_2's synchronization to succeed earlier than the previous implementation. This implementation obeys sequential consistency by requiring P_1's data write on *A* to complete before P_2 executes its data read on *A*. It achieves this by ensuring that (i) when P_2 executes its synchronization read on *Valid*, P_1 notifies P_2 about its incomplete write on *A*, and (ii) P_2 delays its read on *A* until P_1's write on *A* completes.

Thus, both P_1 and P_2 complete their synchronization earlier than with weak ordering. Further, P_2's operations following its synchronization that do not conflict with previous operations of P_1 will also complete earlier. Operations such as the data read on *A* that conflict with previous operations of P_1 may be delayed until P_1's corresponding operation completes. Nevertheless, such operations can also complete earlier than with weak ordering. For example, if P_2's read on *A* occurs late enough in the program, P_1's write may already be complete before P_2 examines the read; therefore, the read can proceed without any delay.

Other implementation proposals in the literature that also obey data-race-free-0 include an aggressive implementation of release consistency (without nsyncs and assuming data = ordinary, and synchronization = sync) that uses (1) a non-binding prefetch into the cache to overlap part of the execution of a synchronization write with

	P1	P2	
	◦ ◦ ◦	◦ ◦ ◦	
Data	A = 100;	while (Valid != 1) {;}	***Synchronization***
	◦ ◦ ◦	◦ ◦ ◦	
Synchronization	Valid = 1;	... = A;	***Data***
	◦ ◦ ◦	◦ ◦ ◦	

(a)



po = program order, so0+ = synchronization-order-0+
WO = weak ordering, DRF0 = data-race-free-0

(b)

Figure 4.6. Implementations of weak ordering and data-race-free-0.

the completion of preceding data operations, and to overlap part of the execution of data writes with the completion of a previous synchronization read, and (2) a rollback mechanism to let a processor conditionally execute its reads following its synchronization read before the synchronization read completes [GGH91b]. Our optimizations will benefit such implementations also because we allow a processor to issue and complete its synchronization writes earlier and consequently synchronization reads that need to return the value of this write can complete earlier; we also allow the data writes following a synchronization read to be issued and completed earlier, and the data reads following a synchronization read to be committed earlier. Another aggressive implementation called lazy release consistency has been proposed for software-based shared virtual memory systems [KCZ92]. This implementation does not obey weak ordering or release consistency, but does obey our sufficient conditions for data-race-free-0 described in the next chapter (assuming all read synchronizations are acquires and all write synchronizations are releases).

4.4. Comparison of Data-Race-Free-0 with Weak Ordering

This section compares data-race-free-0 with weak ordering on the basis of the 3P criteria — programmability, portability, and performance.

Programmability.

Data-race-free-0 provides a simple and formal programmers' interface: if programmers write data-race-free-0 programs, a data-race-free-0 system guarantees sequential consistency. This allows programmers to reason only with sequential consistency, a natural extension from uniprocessors and the most commonly (and often implicitly) assumed interface for multiprocessors. Further, Section 4.2 has argued that the knowledge and effort required of a programmer to write data-race-free-0 programs is often also required for writing programs for sequential consistency.

In contrast to data-race-free-0, the definition of weak ordering provides a more difficult and unfamiliar interface for programmers, requiring programmers to reason with memory operations that may be executed out of program order and writes that may be executed non-atomically. The authors of weak ordering do state that programs have to obey certain conditions (Section 2.2.1); however, further interpretation and reasoning with the weak ordering definition is required to formally deduce whether a program obeys the required conditions and how the hardware will behave for programs that obey these conditions. Data-race-free-0 formally defines both of the above aspects: a program should not result in a data race in any sequentially consistent execution and a data-race-free-0 system appears sequentially consistent to such programs.[12]

Thus, data-race-free-0 provides an interface that is easier to program than weak ordering.

Portability.

An important architectural feature is the ease with which programs written for the architecture can be ported to other architectures. Section 4.3 discussed several possible implementations for data-race-free-0. As motivated by the first implementation discussed in Section 4.3, data-race-free-0 allows all implementations of weak ordering (a more formal argument appears in Section 5.5). However, three out of the four implementations motivated in Section 4.3 (and discussed in more detail in Chapter 5) do not obey weak ordering. Figure 4.7 illustrates a program that would always run correctly on a weakly ordered system, but may not run correctly on the aggressive implementations motivated in Section 4.3. Describing the architectural interface for all of the above implementations as data-race-free-0, however, allows a program written for any of the above implementations to work correctly on all of the others. Thus, data-race-free-0 enhances portability. (The example in figure 4.7 may seem convoluted and unrealistic to some readers. Note, however, that system designers must ensure that *all* possible programs, no matter how unrealistic, are executed according to the system specification. Data-race-free-0 does not give any guarantees for such convoluted code, and so can afford more optimizations than weak ordering for more realistic code.)

Note that if memory models are defined using the hardware-centric approach of weak ordering, then the formal specifications of the four implementation proposals (motivated by Section 4.3) given in the next chapter and the lazy release consistency specification (assuming all synchronization reads are acquires and all synchronization writes are releases) comprise five different memory models (there exist programs that would give different results on the different models). Data-race-free-0, however, unifies all of the above models. We will also show in later chapters that the SCNF method also enhances portability between different SCNF models, primarily since SCNF models allow a conservative ''don't-know'' option, and since all models allow programmers to reason with the common interface of sequential consistency.

[12] As an informal example, consider a program where processors obtain tasks from a task queue protected by a lock, and then process the tasks. Assume that the task processing does not require additional synchronization with sequential consistency because in a sequentially consistent execution, two tasks that are processed concurrently do not execute conflicting operations. Further interpretation seems to be required, however, to unambiguously conclude whether such a program obeys the program constraints for weak ordering given in Section 2.2.1, since the task processing is not done in a critical section, but the program seems to be adequately synchronized. The data-race-free-0 definition unambiguously allows the conclusion that the program obeys the data-race-free-0 conditions since no sequentially consistent execution of the program exhibits data races.

	P1	P2	
data	A = 1	B = 1	***data***
synchronization	C = 1	D = 1	***synchronization***
data	r1 = B	r2 = A	***data***

Figure 4.7. Portability with weak ordering and data-race-free-0.

Assume the bold words next to an instruction indicate the way the operations from the instruction are distinguished to the system (by the programmer). In a sequentially consistent execution, the reads of A and B cannot both return 0. Weak ordering ensures this for the above program. However, the program is not data-race-free-0 and so the aggressive implementations motivated in Section 4.3 allow executions where both the reads return 0. If registers r1 and r2 represent external output interfaces, then it follows that the aggressive implementations do not appear sequentially consistent.

Performance.

As discussed above, data-race-free-0 allows all implementations of weak ordering. Additionally, data-race-free-0 also allows implementations that are not allowed by weak ordering. Weak ordering requires a processor to stall on a synchronization operation until all previous operations complete. The second method discussed for data-race-free-0 describes an implementation where synchronization reads do not have to wait for preceding operations to complete, and data operations do not have to wait for preceding synchronization writes to complete. The third and fourth methods do not require a processor to delay any synchronization operation for preceding operations to complete. Section 4.3 qualitatively indicated the increased performance potential with such implementations for some programs.

Thus, compared to weak ordering, data-race-free-0 provides more flexibility to system designers by allowing more implementations and has the potential for higher performance for some programs.

A potential disadvantage of data-race-free-0, however, is that it constrains programmers to identify all potential race operations as synchronization operations. Weak ordering may result in higher performance than data-race-free-0 if it were possible to identify a race operation as a data operation and still get sequentially consistent results. However, the following observation and conjecture lead us to believe that the above advantages of ease-of-programming, portability, and increased performance potential (through aggressive implementations) may offset this disadvantage for data-race-free-0.

The observation cited above is that a data-race-free-0 system can be accompanied by a hardware-centric specification that describes the specific implementation. Programmers who want the maximum performance possible can reason with this low-level system specification to determine the optimal way to distinguish memory operations. Such programmers will not benefit from the advantages of data-race-free-0, but others who prefer to reason with sequential consistency can continue to do so.

The conjecture cited above is that the performance increase through allowing data races may either not be significant or would be difficult to exploit because of the following reason. In most programs, race operations are used to order conflicting data operations. Therefore, irrespective of how the operations are distinguished, to get sequential consistency, a weakly ordered system must stall on the race operations until preceding operations that the race is meant to order are complete, and stall on a non-race operation until the preceding race that is meant to order it is complete (Chapter 7 provides more intuition for this). Data-race-free-0 gives one way of distinguishing memory operations that will ensure the above. Other ways that could give higher performance would require that there be intervening operations between a race and the data operation that it is supposed to order, and that programmers realize and be able to exploit this fact. This is difficult and the performance gains, especially compared to the more aggressive implementations of data-race-free-0, are not clear.

A disadvantage of data-race-free-0 similar to the above arises for programmers using asynchronous algorithms [DeM88] that do not rely on sequential consistency for correctness. Such programs would probably get better performance in the presence of data races. However, again the observation specified above leads us to believe that the advantages of data-race-free-0 for the many programmers who do prefer to reason with sequential consistency offset the disadvantage.

In conclusion, for many programmers, data-race-free-0 provides the advantage of a simpler interface, enhanced portability, and higher performance potential than weak ordering. For some programmers, weak ordering may offer a higher performance potential, and they may be willing to sacrifice the advantages of data-race-free-0 for this potential. Data-race-free-0 provides such programmers the choice of dealing with the low-level hardware-centric specification to get the higher performance, but at the loss of all the advantages of data-race-free-0.

Chapter 5

A Formalism To Describe System Implementations and Implementations of Data-Race-Free-0

Section 5.1 develops a formalism to describe implementations of SCNF models. Section 5.2 uses the formalism to describe a high-level sufficient condition for implementing data-race-free-0 that encompasses the implementations we can currently envisage. Section 5.3 describes low-level sufficient conditions based on Section 5.2 and describes hardware implementation proposals for those conditions. Section 5.4 discusses compiler optimizations. Section 5.5 uses the specifications of previous sections to argue that all implementations of weak ordering obey data-race-free-0.

5.1. A Formalism for Shared-Memory System Designers

Section 3.3 modeled a system as consisting of processes, input and output interfaces, and a *single* copy of shared-memory locations which could be atomically updated. Such an abstraction of a system is adequate for programmers of SCNF models who only need to deal with sequential consistency. Practically, for higher performance, systems allow multiple copies of shared-memory locations (e.g., in the presence of caches), and do not execute memory operations atomically. System designers need to reason with such features. To describe implementations that can use such features to system designers, our abstraction of a system must model this behavior. Section 5.1.1 extends the notion of a system to model this behavior, and defines a general execution that can describe the results of running programs on such systems. Section 5.1.2 shows how these abstractions of system and execution can be used to describe real system implementations. Section 5.1.3 gives an assumption made by all our implementations of SCNF models that rely on distinguishing memory operations, and extends some previous definitions to the more general notion of an execution.

5.1.1. A Formalism for Shared-Memory Systems and Executions

Based on Collier's work [Col84-92], we extend the definition of a system as follows.

Definition 5.1: A *system* is as defined by definition 3.2 in figure 3.2 with the following changes. (Denote the n processes of the system as P_1, P_2, ..., P_n).

(1) The single copy of shared-memory is replaced with n copies of shared-memory, denoted by M_1, M_2, ..., M_n, where memory copy M_i belongs to process P_i.

(2) In a run of a program on a system, a read operation R to a shared-memory location X specified by process P_i is comprised of a single atomic sub-operation $R(i)$, that reads the value of X in M_i.

(3) In a run of a program on a system, a write operation W to a shared-memory location X with value *Val* specified by process P_i is comprised of an atomic sub-operation $W(i)$ and zero or more atomic sub-operations from $W(1)$, $W(2)$, $\cdots$, $W(i-1)$, $W(i+1)$, $\cdots$, $W(n)$, where the sub-operation $W(k)$ updates the location X in M_k to the value *Val*.

Our previous work describes how the above formalism represents real systems as follows [AdH92]. ''Although real systems do not usually provide physical copies of the entire memory to any processor, a logical copy of memory can be assumed to be associated with every processor. For example, in a cache-based system, the logical copy of memory for a processor may be the union of the processor's cache and all the lines from the main memory that are not in the cache. Also, in a real system, some sub-operations may not be distinct physical entities. However, logically distinct sub-operations can be associated with every operation. For example, an update of main memory on a write constitutes the sub-operations of the write in the memory copies of the processors that do not have the line in their cache. Finally, in real systems, sub-operations may not actually execute atomically; i.e., one-at-a-time and instantaneously. However, in most systems, sub-operations *appear* to execute atomically. For example, the sub-operation involving an update of a processor cache may be spread over an interval of time and may occur in parallel with other sub-operations. However, one can identify a single instant of time at which the update takes effect such that other sub-operations take effect either before or after this time.''

The notion of sub-operations is similar to that of memory operations performing with respect to a processor defined by Dubois et al. [DSB86]. A write sub-operation, $W(i)$, corresponds to the write W performing with respect to processor P_i. A read sub-operation, $R(i)$, corresponds to the read R performing with respect to all processors. Our first use of Collier's work in [AdH92] assumed that a write always had n sub-operations. The relaxation that only some sub-operations need be included was first included in [GAG93] to represent more systems; e.g., systems with software-based cache-coherence where a write need not update the memory copies of all processors. Further, the work in [AdH92] implicitly assumes that in any execution, only a finite number of operations can be ordered before another operation by program order. Our formalism and that in [GAG93] do not make this assumption. Finally, the work in [GAG93] explicitly models features such as write buffers; Section 7.6 of Chapter 7 explains why we chose to exclude that feature.

With the above formalism for a system, an execution is defined similar to a sequentially consistent execution with four important exceptions. First, an execution also consists of a set of sub-operations. Second, the execution order is a total order on sub-operations rather than on operations since sub-operations are the atomic entities; the execution order condition is suitably modified to reflect this. Third, the execution order does not necessarily have to be consistent with the program order. Fourth, an instruction instance may be in the execution even if there are infinite instruction instances before it by program order. This was earlier implicitly prohibited by the uniprocessor correctness condition (Condition 3.11 in figure 3.4) which required an instruction instance to exist in an execution only if it was an initial instruction instance or a next instruction instance of some other instruction instance. We remove the above restriction for general executions so that a processor can now speculatively execute and commit instructions even when it is not known whether a preceding unbounded loop will terminate. This allows useful optimizations since usually either unbounded loops of programs are intended to terminate, or they are not followed by any instructions that can affect the result of the program.

The definition of a general execution and the corresponding execution order condition follows below. The relaxation of the uniprocessor correctness condition described above is intuitive, but involves more complex formalism and is described in Appendix B.

Definition 5.2: *Execution:* An *execution* of a program *Prog* consists of the following components.

(1) A (possibly infinite) set, *I*, of instances of the instructions of program *Prog* that obeys the modified uniprocessor correctness condition in Appendix B.

(2) A set, *O*, of memory operations, specified by the instruction instances in *I*.

(3) A set, *V*, of the values returned by the read operations in *O*.

(4) A set, O_s, of sub-operations corresponding to the operations in *O* as follows. For each read operation *R* by process P_i to location *X* in *O*, O_s contains a read sub-operation $R(i)$ to location *X*. For each write operation *W* by process P_i to location *X* with value *Val* in *O*, O_s contains a write sub-operation $W(i)$ to location *X* with value *Val*, and zero or more write sub-operations from $W(1)$, $W(2)$, ..., $W(i-1)$, $W(i+1)$, ..., $W(n)$ to location *X* with value *Val*.

(5) A total order on the sub-operations in O_s, called the execution order, denoted $\xrightarrow{xo}$, that obeys the execution order condition (Condition 5.3) below.

Condition 5.3: *Execution order condition:*

The execution order, denoted $\xrightarrow{xo}$, of any execution, *E*, is a total order on the set of memory sub-operations, O_s, of *E* that obeys the following.

(1) A read in O_s returns the value of the write sub-operation in O_s that is to the same location and in

the same memory copy as the read and is the last such write sub-operation ordered before the read by the execution order, if such a write exists. If there is no such write, then the read returns the initial value of the location.

(2) If two sub-operations O_1 and O_2 in O_s are from operations of a read-modify-write and O_1 and O_2 are to the same memory copy, then there is no write to the same location and same memory copy as O_1 and O_2 ordered between O_1 and O_2 by the execution order.

(3) The number of sub-operations ordered before any given sub-operation by the execution order is finite.

The definition of the result of an execution is the same as in figure 3.3, and the definition of the data-race-free-0 model is the same as in Definition 4.6. We say an execution gives a sequentially consistent result or appears sequentially consistent if the result of the execution is the result of a sequentially consistent execution.

5.1.2. Using The Formalism to Describe System Implementations

The definition of the data-race-free-0 model (or any SCNF model M) gives the necessary and sufficient condition for a system to obey data-race-free-0 (or model M). However, translating this definition into an implementation is difficult. This section discusses giving specifications that are easier to translate into implementations. This section focuses on implementing the runtime system; Section 5.4 will consider the static compile-time system.

The obvious translation from the definition of data-race-free-0 to a system implementation is to ensure that a run of a program on the system imitates a sequentially consistent execution. This does not, however, exploit the flexibility of data-race-free-0 which requires that a run of a program only *appear* sequentially consistent, and appear sequentially consistent *only to data-race-free-0 programs*. A way to make the translation simpler and provide high performance is to specify other, less stringent constraints on executions, which for data-race-free-0 programs, will give the same result as a sequentially consistent execution. Hardware (or the runtime system) can implement the memory model by directly meeting these constraints. Specifications that give such constraints correspond to the hardware-centric specifications of earlier hardware-centric models. Since we will use them for all aspects of system design (including software), we call them the *system-centric specifications* of the SCNF models.

Definition 5.4: A *system-centric specification* for an SCNF model specifies constraints on executions such that executions that obey the constraints give sequentially consistent results for programs that obey the model.

Hardware (or the runtime system) can implement a memory model by running programs exactly as specified by the system-centric specification of the model. There usually will be a tradeoff in how easily a system-centric specification can be translated to a system implementation and how much flexibility (or performance) it offers.

To translate a system-centric specification into a system implementation, runtime system designers need to determine a mapping between physical events and entities of the system, and the various components of an execution, such that the result of a run of a program that imitates an execution with this mapping is the result of the execution. For example, a read sub-operation can be assumed to occur when the requesting processor procures the line in its cache, or when the requested value arrives in the processor’s register, or when the requested line is dispatched to the processor from another processor’s cache or memory. A mapping for some write sub-operations was given in Section 5.1.1. A mapping for the execution order is the real time order of execution of the various sub-operations. Since these mappings are very specific to specific hardware, but fairly obvious for most systems, we do not give details of such mappings here. In our descriptions of system implementations, we use terms such as ‘‘a sub-operation executes in the hardware,’’ ‘‘a sub-operation executes before or after another sub-operation,’’ etc., implicitly assuming the above mappings. Similarly, we will use informal implementation specific terms such as ‘‘issuing an operation’’ or ‘‘committing an operation;’’ we expect their meanings to be clear for most hardware since the conditions they are expected to meet are formally described by the formal system-centric specifications.

System designers translating a system-centric specification must ensure that all instructions, operations, sub-operations, and values of reads generated in a run of a program (according to the mapping above) respectively form the sets I, O, O_s, and V of an execution such that I obeys the uniprocessor correctness condition, the real time order of execution of sub-operations obeys the execution order condition, and all the above components obey any other conditions in the system-centric specifications. However, note that system-centric specifications are only an aid to the system designers; they do not *require* that the system actually follow them exactly. It is sufficient if the system only *appears* to follow them, or appears to follow any other system-centric specification of the same model. Thus, hardware can execute sub-operations in any order in real time as long as it appears as if the order is the same as for some system-centric specification of the model. Similarly, hardware can actually execute more operations than specified by an execution as long as the effects of these extra operations are not seen. This is in contrast to many earlier specifications [AdH90b, DSB86, GLL90, ScD87] which impose constraints on the real time ordering of events.

We next give an assumption made by all system-centric specifications of all SCNF models that rely on distinguishing between different memory operations, some terminology used by such specifications, and an interpretation of ‘‘result’’ that we use for the rest of this thesis.

5.1.3. An Assumption and Terminology for System-Centric Specifications

The SCNF models that rely on distinguishing memory operations do not place any restrictions on how the operations are distinguished. However, to be able to prove the correctness of specific implementations, we need to make some assumptions. The assumption we make allows the practical mechanisms we can currently envisage (including those discussed in Chapter 4) and is as follows. We assume a system can distinguish an operation O only on the basis of the instruction or address of O, or on instructions (and their operations) preceding O's instruction by program order, or on non-memory instructions following O and in the same basic block as O, or on the write whose value O returns. Informally, this assumption requires that an operation be distinguished either based on its own ‘‘behavior’’ or based on the behavior of only certain ‘‘related’’ operations or instructions.

The system-centric specifications use the following terminology. A condition such as ‘‘$X(i) \xrightarrow{xo} Y(j)$ for all i, j’’ refers to pairs of values for i and j for which both $X(i)$ and $Y(j)$ are defined. The condition implicitly also requires that all such pairs be in the execution. Thus, $W(i) \xrightarrow{xo} R(j)$ for all i,j where R is issued by process P_k implies that $W(m)$ is in the execution for $m = 1,n$ and $W(m) \xrightarrow{xo} R(k)$ for $m = 1,n$, where n is the number of processes in the system. Similarly, ‘‘$X(i) \xrightarrow{xo} Y(i)$ for all i’’ refers to values of i for which both $X(i)$ and $Y(i)$ are defined.

The following definitions extend previous concepts to the more general notion of an execution, and formalize the notion of coherence (or cache coherence) that we will use throughout the thesis. The definitions assume $X(i)$ and $Y(j)$ are memory sub-operations corresponding to operations X and Y respectively.

Definition 5.5: Two memory sub-operations *conflict* if they are to the same location and same memory copy and at least one is a write.

Definition 5.6: For two memory sub-operations $X(i)$ and $Y(j)$ in an execution, $X(i) \xrightarrow{po} Y(j)$ iff $X \xrightarrow{po} Y$.

Definition 5.7: The *synchronization-order-0 ($\xrightarrow{so0}$) and happens-before-0 ($\xrightarrow{hb0}$) relations for any execution:*

Let X and Y be two memory operations in any execution E. $X \xrightarrow{so0} Y$ iff X and Y conflict, X and Y are distinguished as synchronization operations to the system, and $X(i) \xrightarrow{xo} Y(i)$ for some i in E.

The happens-before-0 relation is defined on the memory operations of an execution as the irreflexive transitive closure of program order and synchronization-order-0; i.e., ($\xrightarrow{po} \cup \xrightarrow{so0}$)+.

Definition 5.8: The *synchronization-order-0+* and *happens-before-0+* relations on the operations of any execution are defined exactly as the corresponding definitions for sequentially consistent executions.

Definition 5.9: *Cache coherence or coherence:* Two conflicting write operations, W_1 and W_2, in an execution are cache coherent or coherent iff either $W_1(i) \xrightarrow{\text{xo}} W_2(i)$ for all *i* or $W_2(i) \xrightarrow{\text{xo}} W_1(i)$ for all *i* in the execution.

We next give an interpretation of the *result* of an execution that we use throughout. To determine whether a system-centric specification is correct, we need to compare the result of the executions allowed by the specification to the results of sequentially consistent executions. Given the modified uniprocessor correctness condition (Appendix B), for finite executions, it follows that the values returned by the various shared-memory reads of the execution uniquely determine the result of the execution. Thus, to determine whether the results of two finite executions are the same, it is *sufficient* to only compare the reads of the executions and their values. Further, unless mentioned otherwise, for the program examples we use in the rest of this thesis, we implicitly assume that every instruction that reads shared-memory is followed by an instruction that modifies the output interface based on the value returned by the read. Thus, it follows that unless mentioned otherwise, for two finite executions of our example programs to have the same result, it is *necessary* to ensure that the reads and the values returned by the reads are the same in the two executions. Thus, for finite executions, we will henceforth assume that, unless mentioned otherwise, the reads and the values returned by the reads of the execution comprise the result of the execution.[13] The above observations also hold for infinite executions that obey the unmodified uniprocessor correctness condition (Condition 3.11); i.e., where only a finite number of instruction instances can be program ordered before any instruction instance. To show that the results of two infinite executions that do not obey the unmodified uniprocessor correctness condition are the same, it is sufficient to additionally ensure that the instruction sets of the two executions are identical.

5.2. A High-Level System-Centric Specification for Data-Race-Free-0

This section gives a high-level system-centric specification for the data-race-free-0 model. The advantage of this specification is that it encompasses all data-race-free-0 systems that we can currently envisage; the disadvantage is that it is not very simple to translate into an implementation. The next section will give low-level specifications that are easier to translate to implementations, and will describe implementations based on those specifications. Figure 5.1 summarizes the tradeoffs between performance potential and ease-of-implementation for the various specifications of data-race-free-0 given in this thesis. The definitions of data-race-free-0 and sequential consistency mark the ends of the spectrum; the system-centric specifications and implementations of this and the next sub-section represent intermediate points in the spectrum that increasingly sacrifice implementation flexibility and performance potential in favor of easy translations to actual implementations.

There are three aspects to the high-level system-centric specification proposed in this section: the data, synchronization, and control requirements. These are first intuitively motivated below (the description below is similar to that in [AdH93]).

A system-centric specification of data-race-free-0 is correct if the result of any execution of a data-race-free-0 program allowed by the specification is the same as that of a sequentially consistent execution of the program. From the discussion of Section 5.1.3, the above is true if all the instruction instances of the execution, the shared-memory reads in the execution, and the values returned by the shared-memory reads are the same as that of a sequentially consistent execution. The value of a read is the value from the conflicting write sub-operation that is ordered last before the read by the execution order. Thus, the value returned by a read depends on the order of the conflicting sub-operations in the execution order. Thus, a system-centric specification of data-race-free-0 is correct if for every execution *E* of a data-race-free-0 program that is allowed by the specification, (i) the instruction instances and read operations of *E* are the same as those of some sequentially consistent execution of the program, and (ii) the order of conflicting sub-operations in the execution order of *E* is the same as in the execution order of the above sequentially consistent execution. The three requirements of the system-centric specification given below together ensure the above.

The first requirement of the system-centric specification, called the *data requirement*, pertains to all pairs of conflicting sub-operations of a data-race-free-0 program, where at least one of the operations is a data operation.

[13] A key exception that will be made later is for unessential reads as defined in Chapters 6 and 7.

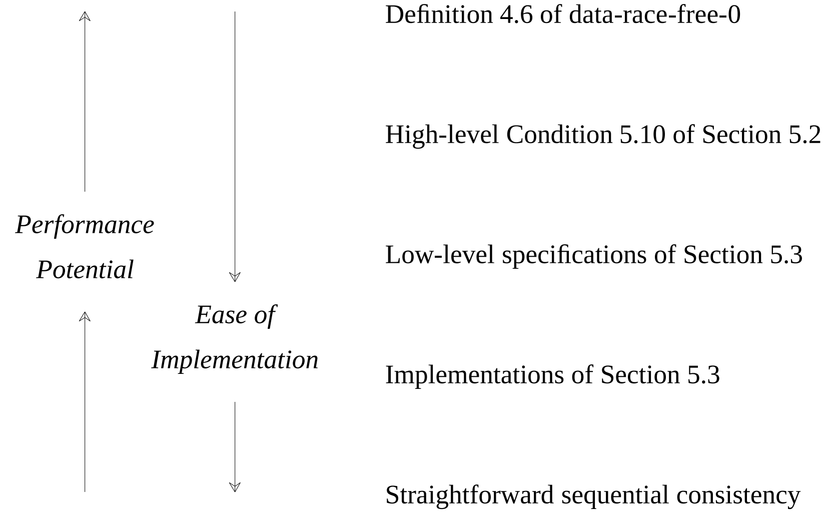

Definition 4.6 of data-race-free-0
High-level Condition 5.10 of Section 5.2
Performance Potential
Low-level specifications of Section 5.3
Ease of Implementation
Implementations of Section 5.3
Straightforward sequential consistency

Figure 5.1. Performance/ease-of-implementation tradeoffs in system-centric specifications.

In a sequentially consistent execution, the operations of such a pair of sub-operations are ordered by the happens-before-0+ relation of the execution (using definition 4.9 for a race), and the sub-operations maintain this order in the execution order. The data requirement is that if two operations are ordered by happens-before-0+, then their conflicting sub-operations should be in the execution and should be ordered similarly by their execution order. This requirement ensures that in figure 4.6, the data read of *A* will return the value of the write of *A*.

The second requirement of the system-centric specification, called the *synchronization requirement*, pertains to all pairs of conflicting synchronization sub-operations of a data-race-free-0 program, and is analogous to the data requirement. In a sequentially consistent execution, the operations of such a pair of sub-operations are ordered by the happens-before-0 relation of the execution, and the sub-operations maintain this order in the execution order. The synchronization requirement for a data-race-free-0 execution is that if two operations are ordered by happens-before-0, then their conflicting sub-operations should be in the execution and should be ordered similarly by their execution order. The need for this requirement is apparent from figure 4.6, if all operations are distinguished as synchronization. This requirement would ensure that the synchronization read of A would return the value of the synchronization write of A.

The data and synchronization requirements would suffice if they also guaranteed that for any execution, *E*, that obeyed these requirements, there is some sequentially consistent execution with the same instruction instances, the same happens-before-0+ relation, and the same happens-before-0 relation as *E*. In the absence of control flow operations (such as branches) and when all sub-operations of a write are guaranteed to appear in the execution, the above is automatically ensured; otherwise, a third requirement, called the *control requirement*, is needed to ensure the above. To see the effect of a write for which all sub-operations are not guaranteed to occur in the execution, consider figure 4.6. If the sub-operation of the write of *Valid* does not occur in P2's memory copy, then P2's loop on valid will never terminate. To see the effect of control flow operations, refer to figure 5.2. Assume all the operations in the figure are distinguished as data operations. In any sequentially consistent execution of the program, the reads of *X* and *Y* would always return the value 0, and therefore the writes on *X* and *Y* could never be in the execution. Thus, there cannot be a data race, the program is data-race-free-0, and executions of the program allowed by the system-centric specification should have the same result as a sequentially consistent execution. If only the data and synchronization requirements are imposed, then there is a possible execution that returns the value 1 for both the reads in the program. The control requirement prohibits such behavior. (Such a behavior would be possible on a real system only with a very aggressive implementation where P_0 could speculatively write *Y* before its read of *X* returned a value, and P_1 could speculatively write *X* before its read of *Y* returned a value. Although such implementations are unlikely, the formal system-centric specification needs to ensure that they are prohibited.)

```
                 Initially X = Y = 0
              P0                        P1
if (X == 1) {Y = 1;}      if (Y == 1) {X = 1;}
```

Figure 5.2. Motivation for the control requirement for data-race-free-0.

The formal system-centric specification for data-race-free-0 and the proof of its correctness follow. These are based on the work in [AdH92]. In the rest of this chapter, *data* (respectively synchronization) *operation* refers to an operation distinguished as data (respectively synchronization).

> **Condition 5.10:** *High-level system-centric specification for data-race-free-0:* An execution E of program *Prog* should satisfy the following conditions:
>
> (1) *Data* - If X and Y are conflicting operations, at least one of X or Y is a data operation, and $X \xrightarrow{\text{hb0+}} Y$, then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i.
>
> (2) *Synchronization* - If X and Y are conflicting synchronization operations, and $X \xrightarrow{\text{hb0}} Y$, then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i.
>
> (3) *Control* - If *Prog* is data-race-free-0, then there exists a sequentially consistent execution, E_s of *Prog*, such that (i) the set of instruction instances and read operations of E and E_s are the same, (ii) for two conflicting operations X and Y, such that at least one of them is a data operation, if $X \xrightarrow{\text{hb0+}} Y$ in E_s, then $X \xrightarrow{\text{hb0+}} Y$ in E, and (iii) for two conflicting synchronization operations X and Y, if $X \xrightarrow{\text{hb0}} Y$ in E_s, then $X \xrightarrow{\text{hb0}} Y$ in E.

Proof that Condition 5.10 is correct.

The control requirement ensures that for any execution, E, of a data-race-free-0 program, *Prog*, the instruction instances and read operations of E are the same as those for a sequentially consistent execution, E_s, of *Prog* and the $\xrightarrow{\text{hb0}}$ and $\xrightarrow{\text{hb0+}}$ relations on conflicting operations are the same as those for E_s. We know that the execution order of E_s orders all conflicting synchronization sub-operations in the same way as the $\xrightarrow{\text{hb0}}$ relation orders the corresponding operations. We also know that the execution order of E_s orders all pairs of conflicting sub-operations, where at least one is data, in the same way as the $\xrightarrow{\text{hb0+}}$ relation orders the corresponding operations. Therefore, the data and synchronization requirements ensure that the execution order of E orders all pairs of conflicting sub-operations of E in the same way as the execution order for E_s. Thus, E and E_s have the same sub-operations and their execution orders order conflicting sub-operations similarly. Therefore, all read sub-operations in E return the same value. It follows that the result of E is the same as that of E_s. □

5.3. Low-Level System-Centric Specifications and Implementations of Data-Race-Free-0

For each of the data, synchronization, and control requirements, we first give an alternative system-centric specification that enforces more constraints, but is easier to translate into an implementation than the requirement in Condition 5.10. We then describe an implementation proposal. The main difference between weak ordering and data-race-free-0 is in the data requirement; therefore, we focus on that requirement. Section 5.3.1 gives four specifications and implementation proposals for the data requirement motivated by Section 4.3 to illustrate the difference between weak ordering and data-race-free-0. Sections 5.3.2 and 5.3.3 give one system-centric specification and implementation for the synchronization and control requirements. For the sake of simplicity, the synchronization and control specifications given here are not very aggressive. (Parts of this work were originally presented in [AdH90b, AdH92, AdH93].)

The proofs that the new low-level system-centric specifications of the data and synchronization requirements specify executions that also satisfy the system-centric specification of Condition 5.10 are straightforward, but the corresponding proof for the control requirement is fairly complex. Chapter 7 develops a common framework and proof for a common system-centric specification for all SCNF models; in Appendix F, we use the common proof of Chapter 7 to prove the special case of data-race-free-0 discussed here.

Below, S, S_1, S_2, S_i, etc. denote synchronization operations; D, D_1, D_2, D_i, etc. denote data operations; Sw, Sw_1, Sw_2, Sw_i, etc. denote synchronization write operations; Sr, Sr_1, Sr_2, Sr_i, etc. denote synchronization read operations; X, Y and Z denote any type of memory operations. Also, $X(i)$ denotes the sub-operation of operation X. We also continue to use the terms *preceding* and *following* to relate to program order. We say an operation *completes* on a system when all its sub-operations have executed.

5.3.1. The Data Requirement

The following discusses four different methods for meeting the data requirement as motivated in Section 4.3. The first two methods have been presented before in the literature in the context of weak ordering and release consistency (RCsc); we present them here for completeness. The implementation proposals below assume a hardware cache-coherent system and the increase in performance potential is due to overlapping memory latencies with useful work. The system-centric specifications of data-race-free-0, however, allow for increased performance potential in other types of systems and in other ways as well; we briefly discuss these at the end of this section.

All the implementation proposals below assume an arbitrarily large shared-memory system in which every processor has an independent cache and processors are connected to memory through an arbitrary interconnection network. The proposals also assume a directory-based, writeback, invalidation, ownership, hardware cache-coherence protocol, similar in most respects to those discussed by Agarwal et al. [ASH88]. One significant feature of the protocol is that invalidations sent on a write to a line in read-only or shared state are acknowledged by the invalidated processors.

The cache-coherence protocol ensures that (a) all sub-operations of an operation are eventually executed, (b) conflicting sub-operations of a write operation execute in the same order in all memory copies, and (c) a processor can detect when an operation it issues is *complete*; i.e., when all the sub-operations of the operation have executed. For (c), most operations complete when the issuing processor receives the requested line in its cache and services the operation. However, a write (data or synchronization) to a line in read-only or shared state completes when all invalidated processors also send their acknowledgements. (Either the writing processor may directly receive the acknowledgements, or the directory may collect them and then forward a single message to the writing processor to indicate the completion of the write.)

Finally, all of the implementation proposals below first assume a process runs uninterrupted on the same processor. The last implementation proposal (method 4) indicates a general technique to handle context switches correctly.

Method 1: Weak Ordering

Low-Level System-Centric Specification for Data Requirement.

Condition 5.11: An execution obeys the data requirement if it obeys the following.

(a) *SyncWrite/SyncRead:* If $X \xrightarrow{po} Y$ and at least one of X or Y is a synchronization operation, then $X(i) \xrightarrow{xo} Y(j)$ for all i,j.

(b) *Sync-Atomicity:* If $Sw \xrightarrow{so0+} Sr \xrightarrow{po} Z$, then $Sw(i) \xrightarrow{xo} Z(j)$ for all i, j.

(c) *Uniprocessor-Dependence* - If $X \xrightarrow{po} Y$, and X and Y conflict, then $X(i) \xrightarrow{xo} Y(i)$ for all i.

Implementation Proposal.

The above specification is similar to that of weak ordering, and can be implemented as follows. The Uniprocessor-Dependence condition is met if the processor maintains local data dependences. Sync-Atomicity is met if a synchronization read miss is not serviced by memory when there are outstanding acknowledgements to the accessed location. SyncWrite/SyncRead can be met by adding the following features to a uniprocessor-based processor logic.

- A counter (similar to one used in RP3) per processor, initialized to zero.
- Modification of issue logic to stall on certain operations.

The counter of a processor is used to indicate the number of issued, but incomplete, memory operations of the processor. It is incremented on the issue of a memory operation and decremented when the operation completes (as detected by the cache-coherence protocol). To meet the SyncWrite/SyncRead condition, the issue logic examines all memory operations in program order, and stalls on a synchronization operation until the counter reaches zero. This ensures that before any synchronization sub-operation executes, all sub-operations of preceding operations execute, and no sub-operation of a following operation executes, thus obeying the required condition.

Method 2: Release Consistency (without nsyncs)

Low-Level System-Centric Specification for Data Requirement.

Condition 5.12: An execution obeys the data requirement if it obeys the following.

(a) *SyncWrite* - If $X \xrightarrow{\text{po}} Sw$, then $X(i) \xrightarrow{\text{xo}} Sw(j)$ for all i,j.

(b) *SyncRead* - If $Sr \xrightarrow{\text{po}} X$, then $Sr(i) \xrightarrow{\text{xo}} X(j)$ for all i,j.

(c) *Sync-Atomicity* - Same as for Method 1.

(d) *Uniprocessor-Dependence* - Same as for Method 1.

Implementation Proposal.

The above specification is similar to that for release consistency (RCsc), assuming no distinction between sync and nsync special operations. It differs from method 1 by distinguishing between write and read synchronization operations. Sync-Atomicity and Uniprocessor-Dependence can be met as for method 1. SyncWrite and SyncRead can be met by adding the following features to a uniprocessor-based processor logic.

- A counter (similar to method 1) per processor.
- Modification of issue logic to delay the issue of, or stall on, certain operations.

SyncRead can be met if the issue logic examines all operations in program order, and stalls after the issue of a synchronization read until the synchronization read completes. For SyncWrite, recall that the counter indicates the number of issued but incomplete operations of a processor. Therefore, the condition is met if the issue of a synchronization write operation is delayed until the counter of the processor reads zero. Data operations following the synchronization write can still be issued.

While a synchronization write is delayed waiting for preceding memory operations to complete, the operations following the synchronization write will also increment its counter. This requires the synchronization write to wait for the new operations as well, and it is possible that the counter may never reach zero. This problem can be alleviated by allowing only a limited number of cache misses to be sent to memory while a synchronization write is delayed for the counter to reach zero. A more dynamic solution involves having two counters and providing a mechanism to distinguish operations (and their acknowledgements) preceding a particular synchronization write from those following the write.

A simpler implementation, such as the DASH implementation of release consistency [GLL90], is possible by forsaking some of the flexibility of the above specification. The DASH implementation blocks on all read operations; therefore, a synchronization write needs to wait for the completion of only preceding write operations. On systems with write buffers, this can be ensured simply by stalling the write buffer before retiring a synchronization write until the counter reads zero. This allows reads following a synchronization write to be overlapped with preceding data writes, but serializes data writes separated by synchronization writes.

Method 3: First Aggressive Implementation For Data-Race-Free-0

The next two methods may seem complex and difficult to currently incorporate in hardware cache-coherent systems. However, this complexity is manageable for software-based shared virtual memory systems, as demonstrated by recent implementations based on similar ideas [KCZ92]. (These are further discussed at the end of this section.) The following method is based on the proposal in [AdH90b].

Low-Level System-Centric Specification for Data Requirement.

Condition 5.13: An execution obeys the data requirement if it obeys the following.

(a) *Post-SyncWrite* - If $X \xrightarrow{\text{po}} Sw \xrightarrow{\text{so0+}} Sr$, then $X(i) \xrightarrow{\text{xo}} Sr(j)$ for all i,j.

(b) *SyncRead* - Same as for Method 2.

(c) *Sync-Atomicity* - Same as for Methods 1 and 2.

(d) *Uniprocessor-Dependence* - Same as for Methods 1 and 2.

Implementation Proposal.

The above specification differs from that of Method 2 by not requiring the issue of a synchronization write to be delayed for preceding operations to complete. Instead, the next synchronization read to the same location by any processor (that might return the value of the above synchronization write) is stalled until the operations preceding the synchronization write complete. The SyncRead, Sync-Atomicity, and Uniprocessor-Dependence conditions are satisfied as for method 2. The Post-SyncWrite condition is met by adding the following features to a uniprocessor-based processor logic and the base cache-coherence logic mentioned earlier.

- A data counter per processor, initialized to zero.
- A synchronization counter per processor, initialized to zero.
- A bit called the *reserve* bit per cache line, initialized to reset.
- Modification of issue logic to delay the issue of, or stall on, certain operations.
- Modification of cache-coherence logic to allow a processor to retain ownership of a line whose reserve bit is set, and to specially handle requests to such a line.

The data and synchronization counters of a processor indicate the number of issued, but incomplete, data and synchronization operations respectively of the processor, in a manner analogous to the counters of the first two methods.

For the Post-SyncWrite condition, a synchronization write is not issued until all preceding synchronization operations complete (accomplished using the synchronization counter, and necessary to prevent deadlock), and all preceding data operations are issued. The reserve bits in the caches are used to ensure that after a synchronization write sub-operation is executed, a synchronization read sub-operation to the same location cannot execute until all operations preceding the synchronization write complete. A reserve bit of a line is set when a synchronization write is issued for the line and if the data counter is positive; i.e., preceding data operations are incomplete. All reserve bits are reset when the counter reads zero, i.e., when all outstanding data operations of the processor complete. (The reset does not necessarily require an associative clear: it can be implemented by maintaining a small, fixed table of reserved blocks as used for method 4 below, or a less aggressive implementation may allow only one reserved line at a time and stall on any following operations that need to reserve a line.) On a synchronization write, the writing processor procures ownership of the requested line, and does not give up the ownership until the reserve bit of the line is reset (the mechanism for achieving this is given in the next paragraph). Consequently, the cache-coherence protocol forwards subsequent requests to the line, including subsequent synchronization reads, to the cache holding the line reserved. The cache controller for this cache can now stall the synchronization reads until the reserve bit of the line is reset (i.e., its data counter reads zero). The cache controller can stall an external request by maintaining a queue of stalled requests to be serviced, or a negative acknowledgement may be sent to the processor that sent the request.

Table 5.1 gives the details of how the base cache-coherence logic can be modified to allow a processor P_i that does a synchronization write to retain ownership of the requested line. The only way a processor could be re-

Request	Reserve Bit of Requested Line Set?	Action
Requests by this processor		
Any	No	Process as usual.
Any read or write	Yes	Process as usual.
Cache line replacement	Yes	Stall processor until all reserve bits reset.
Requests from other processors forwarded to this processor		
Any	No	Process as usual.
Synchronization	Yes	Stall request until reserve bit of line reset.
Data	Yes	If read request, send line to other processor; if write request, update line in this processor's cache and send acknowledgement to other processor; request other processor to not cache the line; inform directory that this processor is retaining ownership.

Table 5.1. Modification to cache-coherence logic at processor.

quired to give up ownership of a line with the base protocol is if it needs to flush the line due to a conflict in its cache, or if it gets an external request for the line. We expect the former case of flushing to be rare and require that the processor stall in that case until it is ready to give up ownership. For the latter case of external requests, we have already discussed that for external synchronization read requests, the cache controller of the processor needs to stall the request to obey the Post-SyncWrite condition. For external synchronization write requests to the same line also, the cache controller must stall the request until it is ready to give up ownership. For all other external requests to the same line (i.e., data requests), the cache controller must perform a *remote service*. The remote service mechanism allows a cache controller to service the requests of other processors without allowing those processors to cache the line. The mechanisms of stalling operations for external synchronization writes and remote service for data operations are both necessary. This is because stalling data operations can lead to deadlock (as illustrated by figure 5.3), and servicing external synchronization writes remotely would not let the new synchronizing processors procure ownership of the line as required for the Post-SyncWrite condition.[14]

Again, as discussed in method 2, data operations following a synchronization write will also increment the data counter in the processor, and could slow progress by not letting the counter reach zero in reasonable time. To preclude such a case, the implementation needs to either put a bound on the number of data operations that can be issued while a line is reserved, or to have a more dynamic mechanism with two or more counters, as discussed for method 2.

The proposal described above never leads to deadlock or livelock as long as the underlying cache-coherence protocol is implemented correctly, and messages are not lost in the network (or a time-out that initiates a system clean-up is generated on a lost message). Specifically, the above proposal never stalls a memory operation indefinitely since (i) the proposal never delays the completion of issued data operations, and (ii) the proposal delays an operation of a processor only if its preceding synchronization operation is incomplete, or issued, data operations of other processors are incomplete.

[14] The remote service mechanism should be included in the implementation proposal presented in our previous work [AdH90b].

	P1	P2	
data	A = 1	B = 1	***data***
synchronization	B = 2	A = 1	***synchronization***

Figure 5.3. Need for remote service of data operations.

Consider an execution where the synchronization writes are allowed to get ownership of their lines before the preceding data writes. Then, if remote service is not provided for data writes, each process will stall the data write of the other process indefinitely, leading to deadlock.

Method 4: Second Aggressive Implementation For Data-Race-Free-0

The following method is based on the material in [AdH92, AdH93].

Low-Level System-Centric Specification for Data Requirement.

Condition 5.14: An execution obeys the data requirement if it obeys the following.

Let X and Y be conflicting memory operations such that at least one of X or Y is a data operation.

(a) *Post-SyncWrite* - If $Z \xrightarrow{po} Sw_1 \xrightarrow{so0+} Sr_1 \xrightarrow{po} Sw_2 \xrightarrow{so0+} Sr_2$, then $Z(i) \xrightarrow{xo} Sr_2(j)$ for all i,j.

(b) *SyncRead* - Same as for Methods 2 and 3.

(c) *Post-SyncRead* - If $X \xrightarrow{po} Sw \xrightarrow{so0+} Sr \xrightarrow{po} Y$, then $X(i) \xrightarrow{xo} Y(i)$ for all i.

(d) *Sync-Atomicity* - Same as for Methods 1, 2, and 3.

(e) *Uniprocessor-Dependence* - Same as for Methods 1, 2, and 3.

Implementation Proposal.

The above specification differs from Method 3 by not requiring a processor (say P_i) that issues a synchronization write (on say line *l*), to delay the next synchronization read (of any processor) until operations preceding the synchronization write complete. Instead, P_i can send the identity of its incomplete operations to the processor that does the subsequent synchronization read on line *l*. The reading processor ensures that it does not subsequently access a location for which P_i has a pending operation. SyncRead, Sync-Atomicity, and Uniprocessor-Dependence can be satisfied as in the earlier methods. The Post-SyncWrite and Post-SyncRead conditions involve adding the following features to a uniprocessor-based processor logic and the base cache-coherence logic mentioned earlier.

- Addition of three buffers per processor -- incomplete, reserve, and special (tables 5.2(a) and 5.2(b)).
- A synchronization counter (similar to method 3).
- Modification of issue logic to delay the issue of or stall on certain operations (table 5.3(a)).
- Modification of cache-coherence logic to allow a processor to retain ownership of a line in the processor's reserve buffer and to specially handle requests to such a line (almost similar to table 5.1 of method 3).
- A processor-to-processor message called ‘‘empty special buffer.’’ (table 5.3(b)).

To meet the Post-SyncWrite and Post-SyncRead conditions, the incomplete buffer and reserve buffer respectively are used analogous to the data counter and reserve bits of the previous implementation. A simple data counter that indicates the number of outstanding data requests no longer suffices because we need to also store the identity of the outstanding requests. Reserve bits could be used instead of the reserve buffer, but as men-

Buffer	Contents	Purpose
Incomplete	Incomplete data operations (of this processor)	Used to remember incomplete operations (of this processor) preceding a synchronization write (of this processor).
Reserve	Synchronization writes (of this processor) for which there are incomplete operations	Used to remember synchronization writes (of this processor) that may cause future synchronization reads (of other processors) to the same line to need special attention.
Special	Incomplete operations (of another processor) received on a synchronization read (by this processor)	Used to identify if an operation (of this processor) requires special action due to early completion of a synchronization read (of this processor).

(a) Contents and purpose of buffers

Buffer	**Insertions**		**Deletions**	
	Event	*Entry Inserted*	*Event*	*Entry Deleted*
Incomplete	Data operation issued	Address of data operation	Data operation completes	Address of data operation
Reserve	Synchronization write issued	Address of synchronization write operation	Operations preceding synchronization write complete, (i.e., deleted from incomplete buffer), and special buffer empties	Address of synchronization write operation
Special	Synchronization read completes	Addresses received on synchronization read	‘‘Empty special buffer’’ message arrives	All entries

(b) Insertion and deletion actions for buffers

Table 5.2. Second aggressive implementation of data-race-free-0 (Cont.).

tioned for method 3, the reserve buffer precludes the need for an associative clear of the reserve bits.

The *incomplete buffer* of a processor stores the addresses (and identity) of all the issued, but incomplete, data operations of the processor. A synchronization write is not issued until all preceding synchronization operations complete (accomplished using the synchronization counter, and necessary to prevent deadlock) and all preceding data operations are issued. Thus, the incomplete buffer contains the identity of all the operations preceding a synchronization write that are incomplete. To distinguish between data operations preceding and following a synchronization write, entries in the incomplete buffer may be tagged or multiple incomplete buffers may be used.

Operation	Address in Special Buffer?	Action
Data	No	Process as usual.
Synchronization write	No	Issue after all preceding operations are issued and all preceding synchronization operations complete.
Synchronization read	No	Issue after special buffer empties and stall until synchronization read completes.
Any	Yes	Stall or delay issue of only this operation until special buffer empties.

(a) Modification to issue logic

Event	Message
All incomplete buffer entries corresponding to a synchronization write deleted	Send ‘‘empty special buffer’’ message to processors that executed synchronization reads to the same line as the synchronization write while the line was in reserve buffer.

(b) New processor-to-processor message

Table 5.3. Second aggressive implementation of data-race-free-0.

To meet the PostSyncWrite and PostSyncRead conditions, after a processor executes a synchronization write operation, it prevents a subsequent synchronization read to the same line (by any processor) from completing until (a) all operations received by the writing processor on its synchronization reads preceding this write are complete, and (b) the writing processor transfers to the new reading processor the addresses of all incomplete operations preceding the synchronization write.

To accomplish the above, when a processor issues a synchronization write operation, it stores the identity of the operation in its *reserve buffer*. The entry is deleted from the reserve buffer only after the previously received operations mentioned in (a) complete and the operations preceding the write mentioned in (b) complete. As will be clear from the following, the identity of the former operations is stored in the special buffer, and as discussed before, the identity of the latter operations is stored in the incomplete buffer of the processor. Thus, the entry for a synchronization write remains in the reserve buffer until the entries in its processor’s incomplete buffer tagged for this write are deleted, and its processor’s special buffer empties. After a processor issues a synchronization write, it does not release ownership of the requested line until the identity of this operation is removed from the reserve buffer (the mechanism for this is described in the next paragraph). Thus, all synchronization reads for the line are sent to the processor and the processor can now stall the synchronization read until (a) and (b) above are satisfied. (a) is satisfied when the special buffer of the processor empties. To satisfy (b), the processor transfers the contents of its incomplete buffer tagged for the synchronization write to the processor that did the synchronization read.

The mechanism for retaining ownership of a line in the reserve buffer is the same as described in table 5.1 for retaining ownership of a line with its reserve bit set in Method 3 with the following two exceptions. First, the checks of the reserve bit of a line must be replaced with checks of whether the address of the line is in the reserve buffer. Second, an external synchronization read request to a line in the reserve buffer need not be stalled until the line is deleted from the reserve buffer, but it can be serviced remotely once the conditions described in (a) and (b) above are met.

Satisfying (a) above directly meets the Post-SyncWrite condition. For the Post-SyncRead condition, a processor must delay an operation following a read synchronization until the completion of any conflicting operation transferred to it on the read synchronization. For this purpose, a processor uses a *special buffer* to save all the information transferred to it on a read synchronization. If a following operation conflicts with an operation stored in the special buffer, the processor can either stall or delay only this operation, until it receives an ‘‘empty special buffer’’ message from the processor that did the relevant synchronization write. The latter processor sends the ‘‘empty special buffer’’ message when it deletes the entry for the corresponding synchronization write from its reserve buffer. For simplicity, a processor doing a synchronization read can also stall on the read until its special buffer empties to avoid the complexity of having to delay an operation for incomplete operations of multiple processors.

As for method 3, the above proposal never leads to deadlock or livelock because the proposal never delays the completion of issued data operations, and it delays an operation only if certain issued data operations of other processors are incomplete or a preceding synchronization operation is incomplete.

This completes all the implementation proposals for the data requirement, assuming that a process runs uninterrupted on a processor. To handle context switches correctly, a processor must stall before switching until the various buffers mentioned in the implementations empty, the counters read zero, and the various reserve bits are reset. Overflow of the buffers can also be handled by making a processor stall until an entry is deleted from the relevant buffer.

Alternative Implementations.

The implementation proposals so far have assumed hardware cache-coherent systems and benefit from increased performance potential by overlapping the latency of memory operations with useful work. The system-centric specifications, however, are well-suited for other types of systems and can benefit them in other ways as well.

Software-based shared virtual memory systems [CBZ91, Li89], for example, have different tradeoffs compared to hardware cache-coherent systems. First, more complex mechanisms can be used to ensure memory consistency in software-based systems since the implementation is in software. Second, reducing the number of messages is critical for performance in such systems [CBZ91, KCZ92]. For these reasons, our more aggressive system-centric specifications (Condition 5.10 and Methods 3 and 4) are less formidable to implement and can provide greater benefit on such systems. Munin [BCZ90, CBZ91] employs a release consistency (RCsc) based implementation (Method 2) in which messages for data writes are combined and sent at the following synchronization write (release). A more aggressive implementation not allowed by previous hardware-centric models, but allowed by the specification of Condition 5.10 is the lazy release consistency implementation [KCZ92]. This implementation executes a data write sub-operation of some processor in the memory copy of another processor (P_i) only if P_i executes a synchronization read (acquire) ordered after the data write by happens-before-0+. (Methods 2 and 3 of this section execute a data write in *all* processor’s copies if any processor executes an acquire ordered after the write by happens-before-0+.) This implementation satisfies the system-centric specification of Condition 5.10. It does not, however, satisfy the hardware-centric models of weak ordering or release consistency since it does not stall a synchronization write (release) for preceding operations to complete (this difference is observable by programmers since some programs can give results with lazy release consistency that are not allowed with release consistency).

The specification of Condition 5.10 is also well-suited for software-based cache-coherent systems [ChV88, Che90, CKM88]. Zucker explains how the hardware-centric models proposed so far cannot be used on software-based cache-coherent systems but our specifications lead to efficient implementations [Zuc92].

5.3.2. The Synchronization Requirement

The synchronization requirement is met by ensuring that synchronization operations interact like on a sequentially consistent system. The conditions for sequential consistency given by Dubois et al. [ScD87] motivate the following system-centric specification to meet the synchronization requirement.

Condition 5.15: An execution obeys the synchronization requirement if it obeys the following.

(a) If $S_1 \xrightarrow{\text{po}} S_2$, then $S_1(i) \xrightarrow{\text{xo}} S_2(j)$ for all i,j.

(b) If $Sw \xrightarrow{so0} Sr \xrightarrow{po} S$, then $Sw(i) \xrightarrow{xo} S(j)$ for all i,j.

(c) If Sw_1 and Sw_2 conflict, then either $Sw_1(i) \xrightarrow{xo} Sw_2(i)$ for all i or $Sw_2(i) \xrightarrow{xo} Sw_1(i)$ for all i.

A hardware implementation satisfies part (a) if a processor does not issue a synchronization operation until its preceding synchronization operation completes. Part (b) is the same as the Sync-Atomicity condition of method 1 of the data requirement, and can be satisfied similarly. Part (c) is satisfied by the cache coherence protocol.

5.3.3. The Control Requirement

The specification for the control requirement involves three intuitive, but hard to formalize concepts. Since Chapter 7 will formalize these concepts while developing a common framework for SCNF models, we use those concepts informally here. The first concept is that of a read controlling a memory operation. We (informally) say a read R *controls* memory operation X if (a) R and X are issued by the same process, and (b) the value that R returns determines if the instruction instance that generated X would be executed, or determines the address accessed by X, or determines the value written by X (if X is a write). For example, X may be in only one path of a branch whose outcome is decided by R, or X may access an address in an array whose index is returned by R. The second concept is that of determining whether a specific instruction instance in one execution occurs in another. For a program without loops or recursion, the above is straightforward to determine; for other programs also, the concept is intuitive but the formalization is slightly complex and will be given later. The third concept involves a loop in a program that does not terminate in any sequentially consistent execution, and instruction instances from the loop. The system-centric specification follows. It is mostly based on the work in [AdH92], and assumes that the high-level data and synchronization requirements are obeyed.

Condition 5.16: An execution E of program *Prog* obeys the control requirement if it obeys the high-level data and synchronization requirements of Condition 5.10, and the following.

(a) Let read R control an operation X in E. Then $R(i) \xrightarrow{xo} X(j)$ for all i, j in E.

(b) Consider any sequentially consistent execution, E_s, of program *Prog* and operations X and Y in E_s such that Y is in E, $X \xrightarrow{po} Y$ in E_s, and either X and Y conflict in E_s, or X is a synchronization read in E_s, or Y is a synchronization write in E_s, or X and Y are both synchronization operations in E_s. If the instruction instance i that issued operation X in E_s is not in E, or operation X does not access the same location in E as in E_s, then let R be a read in E that determined that i would not be in E (if i is not in E) or that X would access a different location in E (if X accesses a different location). Then $R(i) \xrightarrow{xo} Y(j)$ for all i, j in E.

(c) Let read R control operation X in E. Consider Y in E such that $X \xrightarrow{po} Y$ in E, and either X and Y conflict in E, or X is a synchronization read in E, or Y is a synchronization write in E, or X and Y are both synchronization operations in E. Then $R(i) \xrightarrow{xo} Y(j)$ for all i, j in E.

(d) If R is a synchronization read and $R \xrightarrow{po} X$ in E, then $R(i) \xrightarrow{xo} X(j)$ for all i, j in E.

(e) Let j be an instance of any instruction j' in E that writes shared-memory or writes to an output interface in E. If j' follows (in E) an instance L of a loop that does not terminate in some sequentially consistent execution of *Prog*, then the number of instances of instructions that are from the loop instance L and that are ordered by program order before j in E is finite.

(f) If a synchronization write operation, *Sw*, is in E, then there is a sub-operation $Sw(k)$ in E for all k from 1 to n where n is the number of processors in the system; i.e., all possible sub-operations of *Sw* are in E.[15]

Hardware can satisfy part (a) of the control requirement if a memory operation is not issued until it is

[15] Kourosh Gharachorloo pointed out that for a control read, we need to also consider a read that determines the value written by a write. Richard Zucker pointed out that for part (b) reads that control the value of a write X in E need not be considered if X's instruction instance does not execute in E or X accesses a different location in E. Parts (e) and (f) were

known that it will be (committed) in the execution and it is known what address it will access and what value it will write (if it is a write). Parts (b) and (c) are satisfied if an operation is not issued until the following is known about the preceding operations that are not yet issued or committed. First, a preceding unissued or uncommitted operation cannot conflict with the current operation and cannot be a read synchronization. Second, if the current operation is a synchronization operation, then all the preceding operations that will be in the execution are known. Two simpler, but more conservative, ways of satisfying parts (a), (b), and (c) are for a processor to block on all reads that could possibly control an operation, or to stall the issue of a memory operation until it is known that the memory operation will be (committed) in this execution and it is known which memory operations preceding this operation will be (committed) in this execution; specifically, speculative execution is prohibited.

Hardware can satisfy part (d) by stalling on a synchronization read until it completes. Hardware can satisfy part (e) by not speculatively executing an instruction that might follow an unbounded loop until it is known that the loop will terminate. The requirement is necessary only for those instructions that might write shared-memory or change the output interface. If hardware has information about whether a loop always terminates in a sequentially consistent execution, then the above requirement is needed only for loops that do not always terminate in sequentially consistent executions. Often, programmers write programs so that a loop always terminates (for every sequentially consistent execution) or is not followed by any other instructions; therefore, the above information may be worthwhile to transmit if hardware does aggressive speculation. Hardware can satisfy part (f) simply by ensuring that when a synchronization write operation is executed, all sub-operations of the write are executed. This is trivially satisfied by most hardware cache-coherence protocols.

5.4. Implementations for Compilers

This section discusses implementations of the static compile-time part of the system. We focus on the compiler, but the following discussion applies to any software that pre-processes the program. We can use the low-level system-centric specifications to reason about compiler implementations similar to the runtime implementations. However, note that the runtime system needs to only ensure that the current run of the program gives the correct output, while the compiler needs to consider all possible runs of the input program for all possible data sets. The reasoning used in this section has evolved from other joint work [AGG93, GAG93].

We consider compiler optimizations that involve reordering instructions of the program (e.g., loop transformations) and allocating shared-memory locations to registers. The optimization of register allocation does not have a direct analog in the runtime system, since this optimization results in eliminating certain operations from the original input program. Below, we first discuss how we can use the system-centric specification to reason about such an optimization. We expect this discussion will extend to other optimizations that involve elimination of memory operations (e.g., common sub-expression elimination).

To use the system-centric specifications for reasoning about register allocation, we model register operations as memory operations as follows. We assume two types of intervals in the instruction instances of a process (as ordered by program order) over which the compiler can allocate a memory location to a register. The first type of interval consists of reads to the location with no intervening writes to that location. In such an interval, the first memory read (by program order) needs to be retained as a memory read that writes the value returned into the register. Subsequent memory reads of the interval are replaced by reads that return values from the above register. Call the first (by program order) memory read of this interval as the start operation of the interval. The second type of interval begins with a memory write and may contain other reads and writes to the same location. In such an interval, the first (by program order) memory write is replaced by a write to a register, and subsequent reads and writes to the same location are replaced by accesses to that register. If the last (by program order) operation (to the register-allocated location) in this interval is a read, then an additional memory write called a flush write needs to be inserted at the end of the interval that writes back the value of the register to the register-allocated memory location. If the last operation (to the register allocated location) at the end of this interval is a write, then either an additional memory write may be inserted as above, or this write itself may be done to memory and treated as the flush write. Call the flush write as the end operation of the interval.

not present in previous versions of the control condition [AdH92] since those versions assumed a stricter model of an execution as discussed in Section 5.1.1. These parts were developed as part of other joint work [AGG93].

It follows that the register reads of the first type of interval can be modeled as memory reads that occur (in the execution order) just after the start read of this interval. Thus, an effect of the register allocation for such an interval is to reorder a register allocated read with respect to the operations (other than the start read) that precede it in the interval. The register operations of the second type of interval can be modeled as memory operations that occur (in the execution order) just before the end write of the interval. Thus, an effect of this type of register allocation is to reorder a register allocated operation with respect to the operations (other than the end write) that follow it in the interval. We can now use the system-centric specification to reason about register allocation by modeling register operations as memory operations as described above.

Consider the data requirement first. The more aggressive specifications cannot be easily exploited by the compiler; therefore, we consider the specification for method 2 (Condition 5.12). The first part states that if an operation *X* is followed by a synchronization write operation *Sw*, then *X* should be executed before *Sw*. To obey this part, the compiler must not reorder instruction instances where the second instance could generate a synchronization write in any execution of the original program.[16] This part does not impose any constraints for register allocation intervals of the first type since these only reorder an operation followed by a read. For register allocation intervals of the second type, there should not be any synchronization write between (by program order) the beginning of the interval and the end write in any execution of the original program.

The second part of the data requirement (Condition 5.12) is analogous and states that an operation that follows a synchronization read should be executed after the read. To obey this part, the compiler must not reorder instruction instances where the first instance could generate a synchronization read in any execution. For register allocation intervals of the first type, there should not be any synchronization read between (by program order) the start read of the interval and the end of the interval. For the second type of interval, a read that could be a synchronization read in any execution should not be replaced with a register operation.

The third part requires that if $Sw \xrightarrow{\text{so0+}} Sr \xrightarrow{\text{po}} Z$, then $Sw(i) \xrightarrow{\text{xo}} Z(j)$ for all i, j. If the compiler obeys the constraints for the second part discussed above, then it also obeys this part.

The fourth part requires maintaining the order of conflicting operations, which most uniprocessor compilers already maintain.

For the synchronization requirement, consider the low-level specification of Condition 5.15. The first part states that synchronization operations should not be reordered. Thus, the compiler should not reorder instruction instances that could generate synchronization operations in any execution of the input program. Further, for register allocation intervals of the first type, a synchronization operation should not be replaced with a register operation if there is another synchronization operation (to another location) preceding it in the interval. Analogously, for register allocation intervals of the second type, a synchronization operation should not be replaced with a register operation if there is another synchronization operation (to another location) following it in the interval. The second part of the requirement is already met by the above. The third part enforces coherence on synchronization writes and is usually not relevant to the compiler.

For the control requirement, the first three parts are obeyed if a control read is not reordered with respect to operations following it. Further, if an operation is replaced by a register operation using the first type of interval, then no read that controls this operation should be after (by program order) the start operation of this interval. If a read is replaced by a register operation with the second type of interval, then there cannot be an operation that the read controls before the end operation of the interval. (These requirements will be relaxed in Chapter 7.) For the part (d) of the control requirement, an instruction following a loop that may not terminate in some sequentially consistent execution should not be reordered with respect to the loop instructions. Part (e) of the control requirement is not usually relevant to the compiler, given the assumption that a write allocated in a register is always flushed.

This completes the requirements specified in the system-centric specification of the last section. In addition, the compiler must also obey the general requirements of an execution; i.e., the uniprocessor correctness condition (Condition 3.11 and Appendix B) and the condition that only a finite number of sub-operations are ordered before

[16] Although we require the compiler to obey our constraints for *any* execution, it is sufficient to consider only the executions possible on the runtime part of its system.

any other by the execution order (Condition 5.3). Traditional uniprocessor compilers obey the major part of the uniprocessor correctness condition constraints. However, additional care is required in the presence of register allocation in multiprocessors. The uniprocessor correctness condition is satisfied if an execution always has an end operation for every register allocation interval of the second type. Specifically, whenever an operation from an unbounded loop is allocated in a register, the corresponding end operation should also be in the loop. The condition for execution order described above is relevant to the compiler in the presence of register allocation. This condition can be met if whenever an operation from an unbounded loop is allocated in a register, then the start or end memory operation of the allocation interval is also in the loop.

The compiler must also ensure that any operation of the input program that could be distinguished as synchronization, is also distinguished as synchronization in the output program. How this is done is specific to the mechanisms provided in the hardware. In the simplest case, where the hardware provides special instructions for synchronization, the compiler need only ensure that it uses the special instructions for every synchronization operation indicated by the input program. However, note that this straightforward reasoning does not work if a synchronization operation is substituted by a register operation since hardware cannot recognize register operations as synchronization. For now, we satisfy the requirement by prohibiting synchronization operations from being allocated in registers. The compiler section in Chapter 7 indicates how this requirement may be met more aggressively.

Finally, note that for the third part of the control requirement, if the runtime system exploits the knowledge of whether a loop terminates in a sequentially consistent execution, then the compiler should ensure this information is also correctly translated.

5.5. All Implementations of Weak Ordering Obey Data-Race-Free-0

This section argues that data-race-free-0 allows all implementations of weak ordering. Consider two operations X and Y such that $X \xrightarrow{po} Y$, one of X or Y is synchronization, and the other is data. Weak ordering ensures that $X(i) \xrightarrow{xo} Y(j)$ for all i, j. Further, if X is a synchronization read and $W \xrightarrow{so0+} X$, then $W(i) \xrightarrow{xo} Y(j)$ for all i, j. Weak ordering also requires that synchronization operations appear sequentially consistent and uniprocessor data dependences be satisfied. These conditions fulfill the data and synchronization requirements for data-race-free-0 and part (d) of the control requirement. Weak ordering also states that uniprocessor control dependences should be satisfied. This notion is not formalized; we assume that it covers parts (a), (b), (c), and (e) of our control requirement. We assume part (f) of the control requirement is met by weak ordering since it requires synchronization operations to be sequentially consistent. Therefore, we claim that all implementations of weak ordering obey data-race-free-0.

Chapter 6

Three More SCNF Models: Data-Race-Free-1, PLpc1, and PLpc2

This chapter defines three SCNF models — data-race-free-1, PLpc1, and PLpc2. The data-race-free-1 model [AdH93] extends data-race-free-0 by allowing the programmer to provide more information about the synchronization operations in the program. Data-race-free-1 unifies the hardware-centric model of release consistency (RCsc) and other new implementations along with the systems unified by data-race-free-0.

The PLpc1 and PLpc2 models are based on our earlier programmer-centric model called PLpc, proposed jointly with others [GAG92]. PLpc unifies the models of total store ordering, partial store ordering, processor consistency, and release consistency (RCpc) along with the systems unified by the data-race-free models. The PLpc1 and PLpc2 models achieve the same goals, but they are specified differently from PLpc to enhance portability across the programmer-centric models and to allow programmers to analyze their programs more incrementally. A later section explains the need for defining the two models in more detail. PLpc1 unifies the models of total store ordering and partial store ordering along with the systems unified by the data-race-free models. PLpc2 further unifies processor consistency and release consistency (RCpc) with the above systems. Thus, a program written for PLpc2 works correctly on all the above-mentioned hardware-centric systems.

Other commercially implemented hardware-centric models not mentioned above are the IBM 370 and the Alpha. We show that with reasonable interpretations of some ambiguities in these models, PLpc1 can be viewed as unifying the Alpha and PLpc2 can be viewed as unifying the IBM 370 model as well.

Sections 6.1, 6.2, and 6.3 describe the data-race-free-1, PLpc1, and PLpc2 models respectively. Each of these sections is similarly organized with a sub-section each on the motivation for the model, the definition for the model, programming with the model, implementations of the model, and comparison of the model with the new hardware-centric models it unifies. Section 6.4 briefly discusses the PLpc model. Section 6.5 compares PLpc1 and PLpc2 with IBM 370 and Alpha. Section 6.6 concludes the chapter by using the SCNF models described so far to re-evaluate the SCNF methodology.

6.1. The Data-Race-Free-1 Memory Model

6.1.1. Motivation of Data-Race-Free-1

Figure 6.1 motivates the data-race-free-1 memory model, first presented in [AdH93]. Figure 6.1(a) shows an implementation of a critical section using locks, where the lock and unlock use the Test&Set and Unset instructions discussed in Section 4.1.3. Assume the data locations accessed before the critical section are different from the data locations accessed in the critical section. Figure 6.1(b) shows an execution of this code involving two processes, P1 and P2, where P1's Test&Set succeeds first. Data-race-free-0 requires the programmer to distinguish all the memory operations from the Test&Set and Unset instructions as synchronization operations. This implies that the straightforward data-race-free-0 implementations (methods 1 and 2) will not allow a processor to execute the write from the Set (and therefore the read from the Test) until its preceding operations complete. However, unlike the Unset, it is not necessary to delay the Set for sequential consistency. Data-race-free-1 exploits this difference between the synchronization writes of the Set and the Unset.

More specifically, many synchronization operations occur in pairs of conflicting write and read operations, where the read returns the value of the write, and the value is used by the reading processor to conclude the completion of all memory operations of the writing processor that preceded the write in the program. In such an interaction, the write synchronization operation is called a *release*, the read synchronization operation is called an *acquire*, and the release and acquire are said to be *paired* with each other. A synchronization operation is *un-*

```
/* code for critical setion */

data ops before critical section
while (Test&Set(s)) {;}
data ops in critical section
Unset(s)
```

(a)

```
     P1              P2
 Test&Set,s
                     ...
    ...
  Write,x
    ...          Test&Set,s
  Unset,s

                 Test&Set,s
                     ...
                   Read,x
                     ...
                  Unset,s
```

(b)

Figure 6.1. Motivation for data-race-free-1.

paired if it is not paired with any other synchronization operation in the execution. In figure 6.1, the write due to an Unset is paired with the Test that returns the unset value; the Unset write is a release operation and the Test read is an acquire operation because the unset value returned by the Test is used to conclude the completion of previous memory operations. The write due to a Set of a Test&Set and a read due to the Test of a Test&Set that returns the set value are unpaired operations; such a read is not an acquire and the write is not a release because the set value does not communicate the completion of any previous memory operations. Similarly, operations that form a race (and hence need to be distinguished as synchronization) but occur due to asynchronous accesses to data [DeM88] are unpaired synchronization in our terminology.

Of the synchronization operations, the system needs to enforce greater restrictions on paired operations since they are used to order data operations. For example, unlike the paired Unset in figure 6.1, the unpaired Set need not await the completion of preceding data operations. Thus, if hardware could distinguish an unpaired operation from a paired operation, it could complete the unpaired synchronization operations faster than the paired synchronization operations without violating sequential consistency. A data-race-free-1 system gives programmers the option of distinguishing *unpairable* synchronization operations from *pairable* synchronization operations. The system distinguishes a synchronization write and read as paired if they are distinguished as pairable by the programmer and if the read returns the value of the write in the execution; otherwise, the synchronization operations are distinguished as unpaired and can be executed faster.

The characterization of memory operations into pairable and unpairable operations is similar to the sync and nsync operations for properly labeled programs of release consistency [GLL90]; Section 6.1.5 discusses the differences.

6.1.2. Definition of Data-Race-Free-1

The previous section informally characterized synchronization operations as paired and unpaired based on the function they perform. This section gives the formal criterion for when the operations are distinguished correctly for data-race-free-1. Again, as for data-race-free-0, the programmer has the option of distinguishing operations conservatively; a synchronization operation can always be distinguished as pairable with all other synchronization operations since only unpairable operations are optimized. Intuitively, data-race-free-1 requires that sufficient operations be distinguished as synchronization (as for data-race-free-0), and sufficient synchronization operations be distinguished as pairable so that only paired operations are used to order data operations. These notions are formalized below.

Definition 6.1: *Synchronization-order-1 ($\xrightarrow{so1}$):* Let X and Y be two memory operations in an execution. $X \xrightarrow{so1} Y$ iff X and Y conflict, X is a write, Y is a read, Y returns the value of X in the execution, and *X* and *Y* are distinguished as pairable synchronization operations to the system. We say *X* is a release operation, *Y* is an acquire operation, and *X* and *Y* are paired with each other in the execution.

Definition 6.2: *Happens-before-1 (* $\xrightarrow{hb1}$*):* The happens-before-1 relation is defined on the memory operations of an execution as the irreflexive transitive closure of program order and synchronization-order-1; i.e., ($\xrightarrow{po} \cup \xrightarrow{so1}$)+

Definition 6.3: *Race:* Two operations in an execution form a *race* under data-race-free-1 iff they conflict and they are not ordered by the happens-before-1 relation of the execution. They form a *data race* under data-race-free-1 iff at least one of them is distinguished as a data operation.

Definition 6.4: *Data-Race-Free-1 Program:* A program is data-race-free-1 iff for every sequentially consistent execution of the program, all operations can be distinguished by the system as either data or synchronization, all pairs of conflicting synchronization operations can be distinguished by the system as either pairable or unpairable, and there are no data races (under data-race-free-1) in the execution.

Definition 6.5: *Data-Race-Free-1 Model:* A system obeys the data-race-free-1 memory model iff the result of every run of a data-race-free-1 program on the system is the result of a sequentially consistent execution of the program.

Mechanisms for distinguishing memory operations.

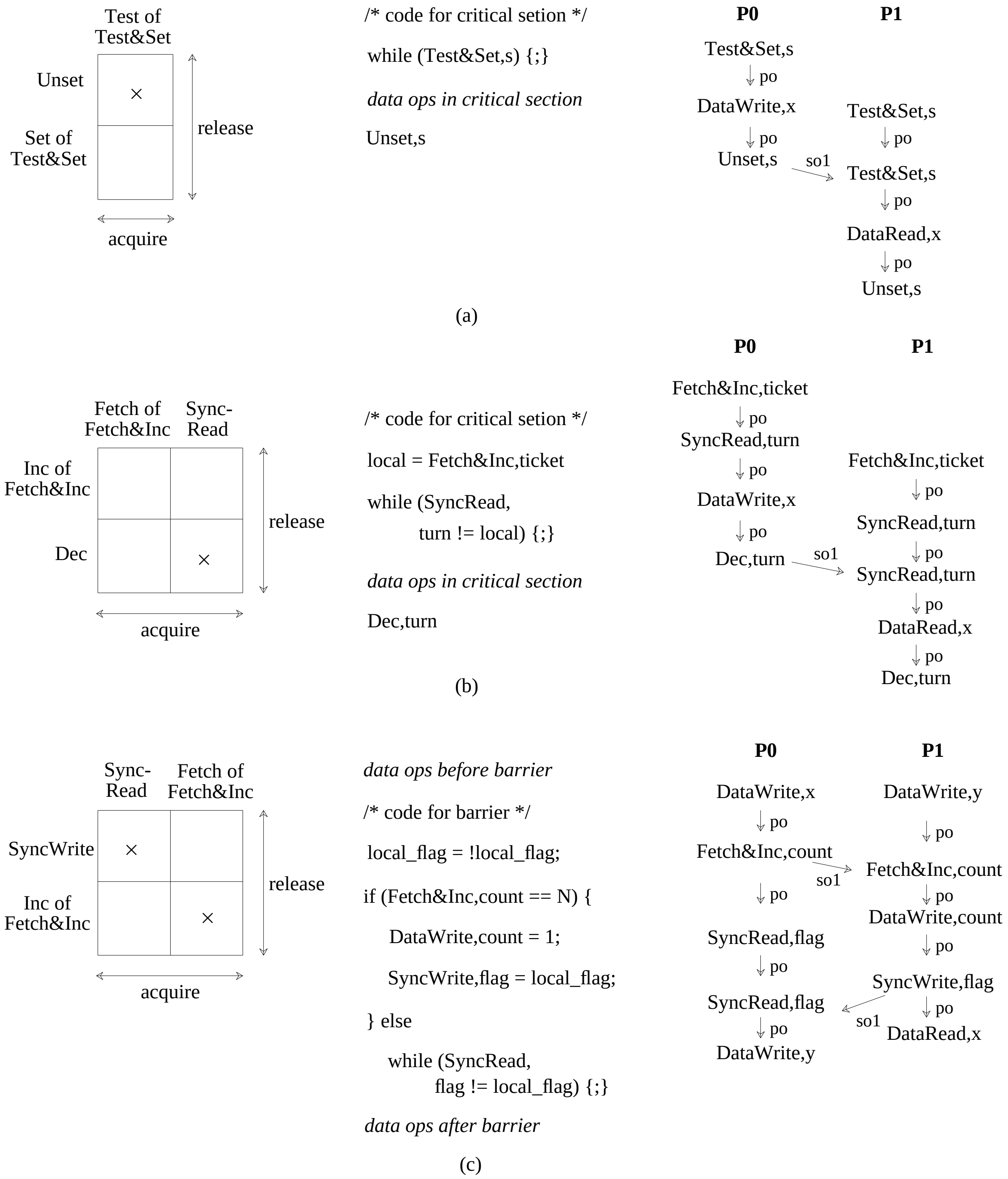
Data-race-free-1 requires a mechanism in the programming language for distinguishing data operations from synchronization operations, and for distinguishing the write/read synchronization operations that are pairable from those that are unpairable. For this purpose, the mechanisms to distinguish data and synchronization operations discussed for data-race-free-0 can be extended to distinguish multiple (>2) categories of operations. One of the categories can be used to distinguish data operations. The rest of the categories can be used to distinguish synchronization operations. In addition, the system can provide a static *pairable* relation on the different categories of synchronization operations that defines the operation categories that are pairable with each other.

A simple implementation of the above notion is to provide three distinct categories (data, unpairable synchronization, and pairable synchronization) and a trivial pairable relation that relates every pairable synchronization write to every pairable synchronization read. Figure 6.2 uses annotations for a program discussed for data-race-free-0. As before the operations on A and B can be distinguished as data and the operations on Valid are synchronization. The operations on Valid are also pairable synchronization, as indicated by the annotations.

```
   P1                          P2

data = ON                   pairable = ON
   A = 100;                    while (Valid != 1) {;}
                            data = ON
   B = 200;                    ... = B;
pairable = ON
   Valid = 1;                  ... = A;
```

Figure 6.2. High-level data-race-free-1 programs.



```
/* code for critical setion */

while (Test&Set,s) {;}

data ops in critical section

Unset,s
```

(a)

```
/* code for critical setion */

local = Fetch&Inc,ticket

while (SyncRead,
        turn != local) {;}

data ops in critical section

Dec,turn
```

(b)

```
data ops before barrier

/* code for barrier */

local_flag = !local_flag;

if (Fetch&Inc,count == N) {

    DataWrite,count = 1;

    SyncWrite,flag = local_flag;

} else

    while (SyncRead,
            flag != local_flag) {;}

data ops after barrier
```

(c)

po = program order, so1 = synchronization-order-1

Figure 6.3. Low-level data-race-free-1 programs.

Figure 6.3 illustrates specific pairable relations for three hardware systems that provide different instructions for data and synchronization (parts (a) and (c) are from [AdH93]). For each system, the figure shows the different synchronization operations and the pairable relation, along with programs and executions that use these operations. The table in each figure lists the read synchronization operations (potential acquires) horizontally, and the write synchronization operations (potential releases) vertically. A '×' indicates that the synchronization operations of the corresponding row and column are pairable. DataRead and DataWrite denote data operations.

Figure 6.3(a) shows a system with the Test&Set and Unset instructions, which are useful to implement a critical section, as discussed in Section 4.1.3. A write due to an Unset and a read due to a Test&Set are pairable. The figure shows code for implementing a critical section using these operations and a sequentially consistent execution with two processors executing the code. Assume the operations in the critical section do not access location *s*. In the execution shown, all pairs of conflicting operations, such that at least one is a data operation, are ordered by the happens-before-1 relation; this is true of all sequentially consistent executions of the code. Therefore, a program consisting of N (≥ 2) processors where each processor is executing the critical section code is data-race-free-1.

Figure 6.3(b) shows a system with the Fetch&Inc [GGK83], Dec, and SyncRead instructions, which are also useful for implementing a critical section, based on ticket locks [MeS91, ReK79]. The Fetch&Inc instruction is a read-modify-write that atomically reads and increments a memory location. The Dec instruction atomically decrements a memory location. The SyncRead is a read that is distinguished by hardware as synchronization. A write due to a Dec is pairable with a read due to a SyncRead. The figure shows code for implementing a critical section and a sequentially consistent execution with two processors executing the code. This implementation of critical sections (and others based on similar mechanisms) have more desirable properties than the implementation in figure 6.3(a) in terms of reduced contention and fairness [MeS91]. In the execution shown, all pairs of conflicting operations, such that at least one is a data operation, are ordered by the happens-before-1 relation; this is true of all sequentially consistent executions of the code. Therefore, a program consisting of N (≥ 2) processors where each processor is executing the critical section code is data-race-free-1 (assuming data operations in the critical section do not access locations *turn* and *ticket*).

Figure 6.3(c) shows a system with Fetch&Inc, SyncWrite, and SyncRead instructions that are useful to implement a barrier [MeS91]. SyncWrite is a synchronization write that updates a memory location to the specified value. A write due to a Fetch&Inc is pairable with a read due to another Fetch&Inc and a write due to a SyncWrite is pairable with a read due to a SyncRead. Also shown is code where N processors synchronize on a barrier [MeS91], and its execution for N = 2. The variable local_flag is implemented in a local register of the processor and operations on it are not shown in the execution. Again, the execution does not have data races and the program is data-race-free-1 (assuming the data operations before and after the barrier code do not access the locations *flag* and *count*.

6.1.3. Programming with Data-Race-Free-1

Programming with data-race-free-1 is similar to programming with data-race-free-0: it allows reasoning with sequential consistency as long as the programmer distinguishes all memory operations correctly. Figure 6.4 shows how memory operations can be distinguished correctly for data-race-free-1. Like data-race-free-0, data-race-free-1 allows programmers to distinguish data operations from synchronization operations; in addition, it also allows programmers to distinguish unpairable synchronization operations from pairable synchronization operations. As for data-race-free-0, any of the definitions of race used for data-race-free-0 can be used to distinguish between data and synchronization for data-race-free-1. For the distinction between pairable and unpairable synchronization, definitions 6.1-6.4 need to be applied (i.e., this distinction is correct if in every sequentially consistent execution, all conflicting pairs of operations, where at least one is distinguished data, are ordered by the resulting happens-before-1 relation of the execution).[17] Data-race-free-1 allows programmers to be conservative

[17] There is one subtle difference between data-race-free-0 and data-race-free-1 that affects the programmers' model. With data-race-free-0, there is always a unique set of memory operations that form a race, and are necessary and sufficient to distinguish as synchronization. Thus, the correct distinction of a memory operation is independent of how the other operations are distinguished. For pairable and unpairable operations, however, there is no such unique set that *must* be distinguished as pairable. Thus, the answer for the second question in figure 6.4 can depend on whether there are other operations distinguished as pairable that already order a data operation.

in making their distinctions by providing ‘‘don’t-know’’ options. As with data-race-free-0, data-race-free-1 does not restrict the use of any algorithms, synchronization primitives, or programming paradigms; it simply requires distinguishing operations correctly.

1. Write program assuming sequential consistency
2. For every memory operation specified in the program do:

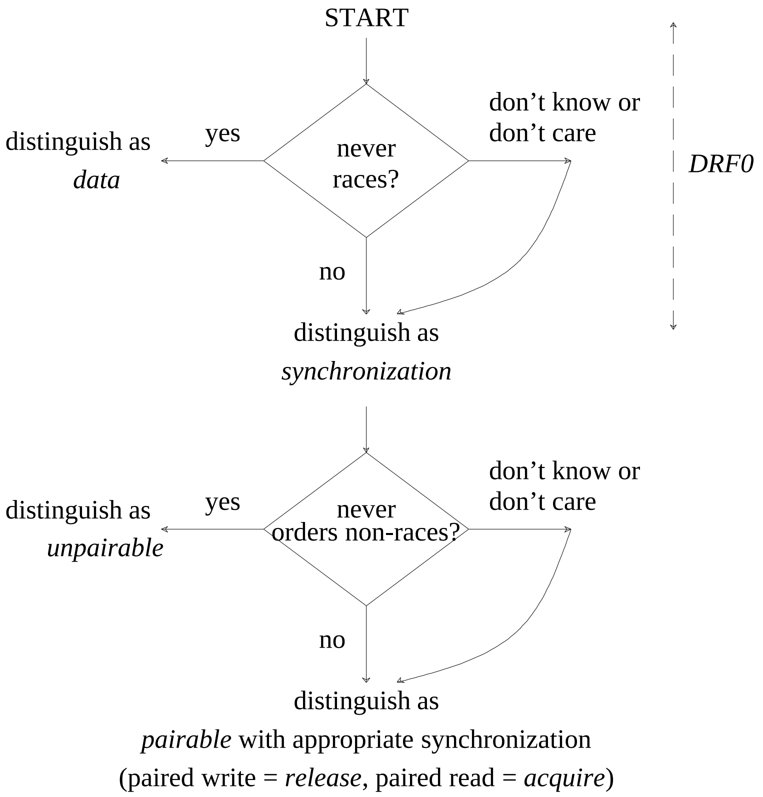

START
never races?
yes
distinguish as *data*
don’t know or don’t care
no
distinguish as *synchronization*
DRF0
never orders non-races?
yes
distinguish as *unpairable*
don’t know or don’t care
no
distinguish as *pairable* with appropriate synchronization
(paired write = *release*, paired read = *acquire*)

Figure 6.4. Programming with data-race-free-1.

Note that it is not necessary to use the data-race-free-1 model at all levels of the system. Instead, it is possible to specify the model for programmers of high level languages as data-race-free-0, and for the hardware as data-race-free-1. Thus, at the high-level, an operation must be distinguished as either synchronization or data. If (for example) some synchronization is through library routines, then the writers of the library routines can distinguish the operations in those routines as unpairable or pairable. Similarly, the compiler can translate high-level synchronization constructs into low-level constructs that exploit the flexibility of data-race-free-1. In such a system, only designers of synchronization library routines, compilers, and other system software designers need to see the added complexity due to data-race-free-1.

6.1.4. Implementing Data-Race-Free-1

The high-level system-centric specification for data-race-free-0 (Condition 5.10), consisting of the data, synchronization, and control requirements, is also a high-level system-centric specification for data-race-free-1. A more aggressive specification that exploits the additional information available in data-race-free-1 programs is one where the happens-before-0+ relation in the data-race-free-0 specification is replaced by the happens-before-1 relation of data-race-free-1. The corresponding low-level system-centric specifications and hardware implementations of data-race-free-0 need to be changed as follows.

For the data requirement, the low-level specification and hardware implementations of data-race-free-0 are valid for data-race-free-1 as well. The system can be more aggressive, however, by exploiting the information that data operations now are ordered only by paired operations. Thus, the data requirement need not restrict the execution of unpaired synchronization operations. Consequently, in each of the previous low-level specifications and hardware implementation descriptions of the data requirement, synchronization writes can be replaced by releases and synchronization reads can be replaced by acquires. Additionally, for the two aggressive hardware implementations (methods 3 and 4), when a processor stalls an external request to retain ownership of a reserved line, the external requests for unpaired synchronization operations can be treated as data requests; i.e., they can be remotely serviced. Specifically, in table 5.1, the row for synchronization applies only to paired synchronization while the row for data applies to data and unpaired synchronization. Also, for method 4, the issue logic can be modified so that unpaired synchronization operations are again treated as data operations; i.e., in figure 5.3(a), the first row applies to data and unpairable synchronization while the second and third rows apply only to pairable synchronization.

For the synchronization requirement, the low-level specification and hardware implementation of the synchronization requirement of data-race-free-0 are valid for data-race-free-1. Data-race-free-1 does not offer any higher performance alternatives for the synchronization requirement.

For the control requirement, again, the requirement for data-race-free-0 is sufficient; however, it can be relaxed for data-race-free-1 as follows. The condition of parts (b) and (c) need not be maintained if one of *X* or *Y* mentioned in those parts is a data operation and the other is an unpaired synchronization operation. Also, the condition of part (d) is required only if *R* is a pairable synchronization read. Hardware implementations for the control requirement of data-race-free-1 are analogous to the data-race-free-0 implementations.

The formal proofs of the above requirements can be derived from the framework of Chapter 7 similar to the derivation for data-race-free-0 in Appendix F.

The examples in figure 6.3 illustrate additional hardware performance gains made possible with data-race-free-1 over data-race-free-0. For the critical section code of part (a), since the Set of the Test&Set is distinguished as unpairable, the execution of the Test&Set can be fully overlapped with the execution of all the preceding data operations. Similarly, in the critical section code of part (b), since the Inc is unpairable, the execution of the Fetch&Inc can be fully overlapped with the execution of all the preceding data operations. These gains are more significant with straightforward implementations (based on methods 1 and 2 of data-race-free-0) that delay the execution of synchronization writes for preceding operations to complete. The barrier code of figure (c) does not provide any significant additional gains over data-race-free-0.

Compiler optimizations for data-race-free-1 are also analogous to those for data-race-free-0 except that now reordering and register allocation can be done on data operations between consecutive acquire and release. Thus, data operations of a processor can be reordered even if there is an unpairable synchronization operation between them, and register allocation intervals can correspondingly be longer.

6.1.5. Comparison of Data-Race-Free-1 with Release Consistency (RCsc)

This section compares data-race-free-1 with release consistency on the basis of programmability, portability, and performance.

Programmability.

For ease-of-programming, release consistency (RCsc) formalizes programs, called properly labeled programs, for which it ensures sequential consistency [GLL90]. All data-race-free-1 programs are properly labeled (interpreting data operations as ordinary, pairable synchronizations as syncs, and unpairable synchronizations as nsyncs), but there are some properly labeled programs that are not data-race-free-1 (as defined by Definition 6.4) [GMG91]. The difference is minor and arises because properly labeled programs have a less explicit notion of pairing. They allow conflicting data operations to be ordered by operations (nsyncs) that correspond to the unpairable synchronization operations of data-race-free-1. Thus, data-race-free-1 is similar in terms of ease-of-programming to release consistency (RCsc) accompanied with the specification of properly labeled programs. Although a memory model that allows all systems that guarantee sequential consistency to properly labeled programs has not been formally described, such a model would be similar to data-race-free-1 because of the similarity between data-race-free-1 and properly labeled programs.

Portability.

Data-race-free-1 allows a strictly greater number of implementations than release consistency (RCsc), because as discussed below, all implementations of release consistency (RCsc) obey data-race-free-1 (interpreting data operations as ordinary, pairable synchronizations as syncs, and unpairable synchronizations as nsyncs), whereas implementations corresponding to methods 3 and 4 in Section 5.3.1 do not obey release consistency (RCsc). Figure 6.5 shows a program that would run correctly on a release consistent (RCsc) system but not on the implementations of methods 3 and 4 (assuming values returned by reads constitute the result of an execution, as discussed in Section 5.1.3). Thus, as for data-race-free-0, data-race-free-1 enhances portability when compared to release consistency (RCsc).

	P1	P2	
data	A = 1	B = 1	***data***
pairable	C = 1	D = 1	***pairable***
pairable	... = E	... = F	***pairable***
data	... = B	... = A	***data***

Figure 6.5. Portability with release consistency (RCsc) and data-race-free-1.

In a sequentially consistent execution, the reads of A and B cannot both return 0. Release consistency (RCsc) ensures this for the above program. However, the program is not data-race-free-1 and so the implementations of data-race-free-1 corresponding to methods 3 and 4 of Section 5.3.1 allow executions where the reads return 0.

Performance.

With respect to performance, data-race-free-1 allows all implementations of release consistency (RCsc) because all data-race-free-1 programs are properly labeled [GMG91] (interpreting data operations as ordinary, pairable synchronizations as syncs, and unpairable synchronizations as nsyncs), and so all implementations of release consistency (RCsc) ensure sequential consistency to data-race-free-1 programs. Further, the implementations corresponding to methods 3 and 4 in Section 5.3.1 for data-race-free-1 violate release consistency (RCsc) because they allow a processor to execute its release even while the preceding operations are incomplete and method 4 allows an acquire to succeed even while preceding operations of the paired release are incomplete. Section 4.3 qualitatively indicated the increased performance potential with such implementations for some programs when compared to weak ordering. Similar observations hold for release consistency (RCsc) since all implementations of release consistency (RCsc) must obey the specification of method 2.[18] As described in Section 4.3, our optimizations also benefit the aggressive implementations of release consistency (RCsc) based on hardware prefetch and rollback [GGH91b].

[18] It may be argued that the definition of ‘‘performs with respect to’’ on which release consistency (RCsc) is based allows a processor to execute its release as long as the release is not made visible to any other processor. Thus, it may seem that our implementation based on method 3 could potentially be extended for release consistency (RCsc) by allowing a release to be executed while preceding operations are pending, but stalling *any* external request to the released line until the pending operations complete. Such an extension, however, is not a correct implementation of release consistency (RCsc) because (1) it would lead to deadlock as illustrated by figure 5.3 in Section 5.3.1, and (2) for a program such as in figure 6.5, the implementation could return 0 for both the data reads violating release consistency (RCsc). The data-race-free-1 implementations avoid the above problems because they do not need to stall external requests to data operations and they do not need to ensure that the program in Figure 6.5 gives sequentially consistent results.

As for data-race-free-0, a potential disadvantage of data-race-free-1 is that programmers writing programs directly for release consistency (RCsc) may be able to use data races and still get correct results, but at a higher performance than possible with data-race-free-1 programs. The arguments used for data-race-free-0 in this context apply to data-race-free-1 as well as follows. Except for programs that employ asynchronous algorithms, it is not clear if exploiting data races can lead to significant performance gains. Since programming directly with release consistency (RCsc) is difficult and since for many programs the additional gains are not clear, many programmers will not want to program directly with release consistency (RCsc), and will benefit from the advantages of data-race-free-1. Programmers using asynchronous algorithms can reason with the system-centric specification of data-race-free-1 for the highest possible performance, but at the cost of the programmability and portability benefits of data-race-free-1.

Thus, compared to release consistency (RCsc), data-race-free-1 provides more flexibility to system designers by allowing more implementations than release consistency (RCsc) and has the potential for higher performance for some common programs.

6.2. The PLpc1 Memory Model

As mentioned earlier, the PLpc1 memory model is based on the PLpc memory model [GAG92]; therefore, the key concepts used to define PLpc1 below were originally presented in [GAG92].

6.2.1. Motivation of PLpc1

PLpc1 extends data-race-free-1 by exploiting additional differences between synchronization operations. Consider the program in figure 6.6. Each processor produces a distinct set of data, sets a flag to indicate the data is produced, tests the flag set by the other processor, and then consumes data produced by the other processor. Data-race-free-1 requires the reads and writes of the flags to be distinguished as synchronization. Therefore, all the implementations of data-race-free-1 considered so far require each processor to stall on its read of a flag variable until its preceding write of a flag variable completes. However, this delay is not necessary for a system to appear sequentially consistent to the program in figure 6.6. To appear sequentially consistent to this program, it is sufficient if in the phase that consumes data, each processor reads the new values of the data produced by the other processor; this is guaranteed even if the write and read of the flag variables of any processor execute in parallel or out of program order (as explained in more detail below). The PLpc1 model distinguishes between synchronization operations to allow some synchronization write and synchronization read pairs of a processor (such as the flag operations) to be executed in parallel and out of program order.

```
                 Initially all locations = 0

P1                                  P2

produce data                        produce data

Flag1 = 1;                          Flag2 = 1;

while (Flag2 != 1) {;}              while (Flag1 != 1) {;}

consume data produced by P2         consume data produced by P1
```

Figure 6.6. Motivation for PLpc1.

More specifically, paired synchronization operations often occur in loop interactions where a loop repeatedly executes synchronization reads until a read returns a specific value from a synchronization write. Further, the synchronization write executes only to terminate such loops and is necessary to terminate the loops. The flag operations of figure 6.6 represent such an interaction. We call the loop that generates the synchronization reads as a *synchronization loop*, reads from such a loop as *loop* reads, and writes that terminate such loops as *loop* writes. The next sub-section formalizes these concepts.

Loop reads and loop writes have two important properties that were first observed for the PLpc model [GAG92]. First, assuming that a synchronization loop eventually terminates, the number of times the loop executes or the values returned by its unsuccessful reads cannot be detected by the programmer and cannot comprise the result of a run of the program. Henceforth, we do not consider such reads to comprise the result of an execution. For example, in Figure 6.6, it cannot be detected and does not matter how many times a processor reads its flag unsuccessfully, or even what values the unsuccessful reads return, as long as eventually a read returns 1 and terminates the loop. Thus, the unsuccessful reads of a synchronization loop can be ignored when analyzing a sequentially consistent execution. Second, in general, two conflicting synchronization operations can occur in any order in an execution. However, the final read of a synchronization loop always must execute after the loop write that terminates the loop. For example, in Figure 6.6, P2’s final read of flag1 must execute after P1’s write of flag1.

The above observations allow reordering a synchronization write followed by a synchronization read where at least one of them is a loop operation. For example, in figure 6.6, executing the flag reads of a processor before its flag write does not violate the appearance of sequential consistency since the unsuccessful reads of the loop can be ignored and the successful read is guaranteed to execute after the correct conflicting write (i.e., the write of the flag of the other processor) and will return the correct value. As long as other constraints of data-race-free-1 implementations are maintained (e.g., the data operations of the consumer phase do not begin until preceding paired synchronization reads complete), each processor will read the data values updated by the other processor and the system appears sequentially consistent. PLpc1 gives programmers the option of distinguishing the loop synchronization operations to exploit the above optimization.

6.2.2. Definition of PLpc1

The following first formalizes synchronization loops, and then formalizes loop and non-loop operations of a sequentially consistent execution. These notions lead to the definition of PLpc1 programs and the PLpc1 model. A PLpc1 program must be a data-race-free-1 program; i.e., it must distinguish every operation as data, pairable synchronization, or unpairable synchronization, as defined for data-race-free-1 programs. Additionally, it must also distinguish all pairable operations as either loop or non-loop. Since PLpc1 seeks to optimize loop operations, any operation can be distinguished as non-loop; however, the operations distinguished as loop must obey the definition for a loop operation given below.

For simplicity, the following formalization of a synchronization loop (from [GAG92]) captures only a simple (but common) case in which the loop repeatedly executes a read or a read-modify-write to a specific location until it returns one of certain specific values. A more general definition allowing locks that employ Test&Test&Set [RuS84] and/or backoff techniques [MeS91] appears in [GAG92]. Chapter 7 provides a further generalization of this concept.

Definition 6.6: *Synchronization Loop:* A *synchronization loop* of a program is a sequence of instructions of the program that satisfies the following.

(i) The loop specifies the execution of a read or a read-modify-write to a specific location. If the value returned by the read is one of certain specified values (called *exit values*), then the loop terminates; otherwise, the loop repeats the above. The read or read-modify-write operations that terminate the loop are called *essential operations*; the other operations from the loop are called *unessential operations.*

(ii) If the loop specifies the execution of a read-modify-write, then the writes of all but the last read-modify-write store values returned by the corresponding reads.

(iii) The loop terminates in every sequentially consistent execution of the program.

Definition 6.7: *Loop and Non-loop Reads:* A read *R* in a sequentially consistent execution is a *loop read* (under PLpc1) iff it does not form a race with any write in the execution or if it satisfies all of the following conditions.

(i) *R* is the essential read of a synchronization loop.

(ii) *R* forms a race with exactly one write, *W*, in the execution.

(iii) *R* returns the value of write *W*.

(iv) The value written by the conflicting write ordered last before *W* by the execution order (or the initial value if there is no other conflicting write before *W*) is not an exit value for *R*'s loop.

(v) If *R* is part of a read-modify-write, then the corresponding write does not form a race with another write in the execution.

A read in a sequentially consistent execution is a *non-loop read* (under PLpc1) iff it is not a loop read.†

Definition 6.8: *Loop and Non-Loop Writes:* A write in a sequentially consistent execution is a *loop write* (under PLpc1) iff it does not form a race with any write or non-loop read (as defined by definition 6.7) in the execution.

A write in a sequentially consistent execution is a *non-loop write* (under PLpc1) iff it is not a loop write.

Definition 6.9: *PLpc1 Programs:* A program is *PLpc1* iff for every sequentially consistent execution of the program such that the execution does not have unessential operations, the following conditions are true:

(i) all operations are distinguished as either data, unpairable synchronization, loop pairable synchronization, or non-loop pairable synchronization,

(ii) there are no data races (as defined by data-race-free-1), and

(iii) operations distinguished as loop pairable synchronization are loop operations by definitions 6.7 and 6.8.

Definition 6.10: *The PLpc1 Memory Model:* A system obeys the PLpc1 memory model iff the result of every run of a PLpc1 program on the system is the result of a sequentially consistent execution of the program.

The definition of PLpc1 programs allows only pairable synchronization operations to be further distinguished as loop or non-loop. We do not make this distinction between data operations because the optimization this distinction allows; i.e., reordering a write followed by a read, is already allowed if either the write or read is a data operation. We chose not to make the distinction for unpairable synchronization operations because we expect that loop interactions will be used mostly by pairable operations and the added complexity to support two types of unpairable operations may not be worth the performance gain.

Further, the definition of PLpc1 programs requires considering only sequentially consistent executions without unessential operations. An alternate way of interpreting the definition is to consider all sequentially consistent executions, but to ignore any unessential operations in the execution when determining whether it obeys the necessary constraints. The previous sub-section motivated why unessential operations can be ignored.

† Definitions 6.7, 6.8, 6.12, and 6.13, and Appendix G differ slightly from the filed thesis. The definitions in the latter contained unnecessary restrictions on read-modify-writes and definition 6.13 did not have part (ii); the modifications in Appendix G reflect the changes in the definitions.

The observation for ignoring unessential operations can also be exploited for data-race-free-0 and data-race-free-1; i.e., a program can be considered data-race-free-0 or data-race-free-1 as long as every sequentially consistent execution without unessential operations does not have data races (where data race is as defined by the respective models). We could not identify any common programming scenarios for which this observation would provide a performance gain with the data-race-free models; however, when porting a data-race-free program to a PLpc1 system, it may be worthwhile to use this observation (one example involving a critical section implementation with Test&Set is given in figure 6.8(b) of Section 6.2.3).

Unfortunately, after ignoring unessential operations, the different definitions of a race given earlier (definition 4.4 and alternatives 1-4 in figure 4.2) do not result in the same set of PLpc1 programs. Specifically, alternatives 2 and 4 are not equivalent to the others because two conflicting data operations (as defined by the other definitions) are not guaranteed to have the path in the program/causal-conflict graph required by alternatives 2 and 4. The distinction between unpairable and pairable synchronization for data-race-free-1 relies on the path in the program/causal-conflict graph. To exploit this distinction and to allow ignoring unessentials at the same time, we could require programmers to use only alternatives 2 or 4 to distinguish data operations; these, however, are significantly more complex than alternative 1. A better solution results from noting that with any definition, a pair of conflicting data operations that does not have a write between them always has the required path in the program/causal-conflict graph (the proof is similar to Appendix A); furthermore, Chapter 7 shows that it is sufficient to consider only the above pairs of conflicting operations to exploit the data-race-free optimizations. For a sequentially consistent execution, call two conflicting operations that do not have another conflicting write between them (where ‘‘between’’ refers to the execution order of the execution) as *consecutive conflicting operations.* Then we can use alternative 1 (or alternative 2 or definition 4.4) to identify races if we impose the data-race-free-1 requirement only for consecutive conflicting operations (i.e., only these need be ordered by happens-before-1). This still does not produce the same set of PLpc1 programs with all definitions, but allows the use of all previous optimizations with the simplest definitions; the rest of this chapter applies to all definitions.

The next sub-section discusses programming with PLpc1, including mechanisms for distinguishing memory operations with PLpc1 and examples of PLpc1 programs.

6.2.3. Programming With PLpc1

The programmers’ interface for PLpc1 is similar to that for the data-race-free models, except that PLpc1 additionally allows programmers to distinguish pairable synchronization loop operations from pairable synchronization non-loop operations. Figure 6.7 shows how programmers can distinguish memory operations correctly for PLpc1. As for the data-race-free models, PLpc1 provides a ‘‘don’t-know’’ option allowing programmers to always distinguish a pairable synchronization as non-loop. Although the formal definitions of loop and non-loop operations are complex, the concepts are intuitively simple; in our analysis of the programs mentioned earlier, we did not have much difficulty in avoiding the ‘‘don’t-know’’ option.

For distinguishing loop and non-loop operations, a PLpc1 system can provide mechanisms similar to those for distinguishing the operations of data-race-free programs, as shown in Figure 6.8 (the examples of the figure are also used for the PLpc model in [GAG92]). Figure 6.8 (a) shows how the program of figure 6.6 can be converted to a PLpc1 program assuming the programming language provides annotations. A sufficient set of annotations for PLpc1 is: *data = ON*, *unpairable = ON*, *loop = ON*, and *non-loop = ON*.

In figure 6.8(a), conflicting flag operations form a race and so should be distinguished as synchronization operations; further, the write of a flag is pairable with the reads of the same flag. We assume that the operations to produce and consume data do not form a race and so are data operations. The while loops containing the flag operations form synchronization loops; therefore we can ignore the unsuccessful reads of flag. In any sequentially consistent execution, the final read of flag forms a race with only the write to the same flag, the read returns the value of the write, and this write is necessary for the loop of the read to terminate; therefore, the final flag read is a loop read. The writes to the flags do not form a race with another write or a non-loop read; therefore, the writes to the flags are loop writes. Thus, annotations of *data = ON* before the producer phase, *loop = ON* before the flag write, and *data = ON* before the consumer phase make the program a PLpc program.

Figure 6.8(b) shows an implementation of a critical section using the Test&Set and Unset operations previously shown in Section 4.1.3. The distinctions for the operations made in the figure are mostly analogous to the

1. Write program assuming sequential consistency

2. For every memory operation specified in the program do (ignore unessentials):

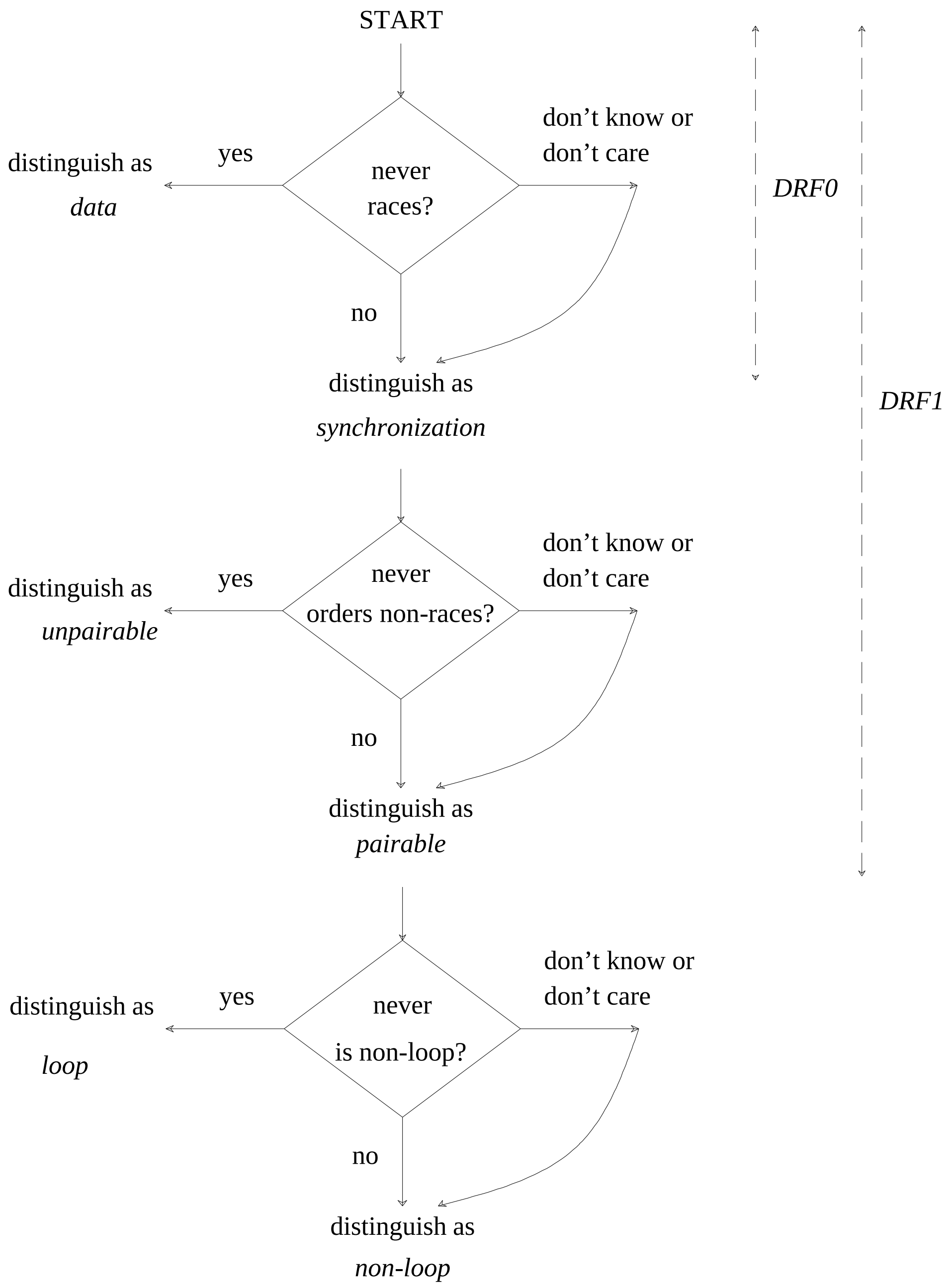


Figure 6.7. Programming with PLpc1.

```
     P1                                    P2

data = ON                              data = ON
     produce data                           produce data

loop = ON                              loop = ON

     Flag1 = 1;                             Flag2 = 1;

     while (Flag2 != 1) {;}                 while (Flag1 != 1) {;}

data = ON                              data = ON
     consume data produced by P2            consume data produced by P1
```

(a)

```
                                  Assume hardware recognizes
                                     Test = loop
       while (Test&Set,s) {;}
                                     Set = data

       data ops in critical section


       Unset,s                       Unset = loop
```

(b)

```
data ops before barrier
/* code for barrier */
 local_flag = !local_flag;                 Assume hardware recognizes

if (Fetch&Inc,count == N) {                    Fetch&Inc = non-loop
    DataWrite,count = 1;                       DataWrite = data
    SyncWrite,flag = local_flag;               SyncWrite = loop
} else
   while(SyncRead,flag != local_flag) {;}      SyncRead = loop
data ops after barrier
```

(c)

Figure 6.8. Examples of PLpc1 programs.

For (b) and (c), assume a program with $N \geq 2$ processors executing the given code.

previous example. However, this example illustrates that on a PLpc1 system, it is useful to ignore unessential operations even while making the distinctions that are also made by the data-race-free models. The while loop containing the Test&Set forms a synchronization loop; therefore, we can ignore unsuccessful Test&Sets. In any sequentially consistent execution, the Test read of an essential Test&Set forms a race only with the write from the Unset that is required for the loop to terminate. Further, a write due to the Set of an essential Test&Set never forms a race in a sequentially consistent execution that does not have unessential Test&Sets. Therefore, the Test read is a loop read and the Set write is a data write. A write from an Unset races only with the read from an essential Test; therefore, it is a loop write. If hardware recognizes Test, Set, and Unset to respectively represent loop reads, data writes, and loop writes, then the program given is PLpc1. Note that not ignoring the unessential Test&Sets would require distinguishing the write from the Set as an unpairable synchronization since it could form a race with a Test read of an unessential Test&Set and an Unset write. This precludes the full overlap of an Unset and Test&Set of a processor. Thus, for the full benefit of PLpc1, even the distinctions of data and synchronization should be made after ignoring unessential operations. On a data-race-free-1 system, however, distinguishing the Set write as data does not provide any benefit. (Also note that although the performance advantage of distinguishing the Set as unpairable in data-race-free-1 is also gained by ignoring unessential operations, the distinctions of pairable and unpairable are still useful for other cases; e.g., the Fetch&Inc in figure 6.1(b) and asynchronous data operations that may occur along with other synchronized data operations.)

Figure 6.8(c) illustrates non-loop operations. It shows the implementation of a barrier previously discussed in Section 6.1.2 for figure 6.3. The while loop containing the reads on flag forms a synchronization loop; therefore, we ignore the unsuccessful reads of flag. The write to count from DataWrite is data; the Fetch read and Inc write to count are pairable synchronization with Inc writes pairable with Fetch reads, and the write and the final read to flag are also synchronizations pairable with each other. The read and write to flag are loop operations, while the Fetch and Inc operations are non-loop operations. Thus, the program given is PLpc1 if hardware recognizes writes from SyncWrite and reads from SyncReads as synchronizations pairable with each other, writes from Inc and reads from fetch as non-loop synchronizations pairable with each other, and writes from DataWrite as data operations.

6.2.4. Implementing PLpc1

A complete system-centric specification for PLpc1 and its proof of correctness appear in Appendix G. This section focuses on the additional optimization that the extra information in PLpc1 programs allows, as compared to the information in data-race-free-1 programs. As motivated in Section 6.2.1, the key additional optimization is in the reordering of certain synchronization operations. All implementations of data-race-free-1 seen so far required synchronization operations to be executed in program order; i.e., for synchronization operations S_1 and S_2, if $S_1 \xrightarrow{\text{po}} S_2$, then $S_1(i) \xrightarrow{\text{xo}} S_2(j)$ for all i,j. With PLpc1, in the case where S_1 is a write and S_2 is a read, the above condition need not be maintained if at least one of S_1 or S_2 is a loop operation.

Figure 6.8 uses examples from [GAG92] to illustrate the performance gains in hardware with the above optimization. For the code in part (a), all data-race-free-1 implementations so far require a process to delay its read of a flag variable (and hence the consumption of data) until its write of a flag variable. PLpc1 allows the latency of the write and read to be overlapped since they are both loop operations. Further, for the simpler implementations of data-race-free-1, the flag variable is not set until the preceding data operations complete, requiring the flag read to wait for the preceding data operations to complete as well. PLpc1 does not impose such a delay. Thus, if process P_1 finishes producing its data and sets its flag before process P_2, process P_2 can read the flag and begin consuming P_1's data even while it has not set its flag. This allows the latency of the flag write (and the latency of the preceding data operations for straightforward implementations) to be overlapped with the following computation. The critical section of part (b) can lead to similar gains. Consider a case similar to part (a) where instead of producing and consuming data, each processor accesses two critical sections protected by two different locks. Thus, the writes and reads to the two flags are replaced by Unset and Test&Set of two different locks respectively. In this case, the Test&Sets to access the second critical section of a processor can be overlapped with the Unset to exit the first critical section of that processor. The barrier code of part (c) does not gain by PLpc1.

Additional compiler optimizations allowed by PLpc1 are analogous to the hardware optimizations.

6.2.5. Comparison of PLpc1 with SPARC V8 and Data-Race-Free Systems

This section compares PLpc1 to data-race-free-1 and the new hardware-centric models it unifies (total store ordering and partial store ordering) on the basis of programmability, portability, and performance.

Programmability.

Compared to data-race-free-1, PLpc1 gives the programmer the option of providing extra information to the system in the form of whether a synchronization operation is a loop or non-loop operation. Since the programmer can always conservatively specify an operation to be a non-loop operation, the PLpc1 interface is more complex than data-race-free-1 only for programmers who want higher performance than data-race-free-1 systems can provide.

Compared to total store ordering and partial store ordering, the benefits for ease-of-programming with PLpc1 are similar to those of the previous programmer-centric models. PLpc1 provides the familiar and simple interface of sequential consistency; total store ordering and partial store ordering provide more complex interfaces requiring programmers to reason about hardware features such as write buffers and out-of-order issue. Similar to weak ordering, the authors of total store ordering and partial store ordering specify certain scenarios where these models give sequentially consistent results (Section 2.2.2). PLpc1 provides a formal interface that does not restrict programmers to the above scenarios.

Portability.

Programs written for PLpc1 can run on any data-race-free-1 system with sequentially consistent results since PLpc1 programs also contain the information exploited by data-race-free-1. PLpc1 programs can also run on total store ordering and partial store ordering systems with sequentially consistent results as follows.

First consider the total store ordering model which essentially allows two optimizations: (1) reads can bypass preceding writes, and (2) a read can return the value of a write by its own processor when the write has not been seen by other processors (a write is seen by the other processors at the same time). The model additionally provides read-modify-write operations that can be used to eliminate the effect of the first optimization. During our joint work for the PLpc model, Kourosh Gharachorloo first pointed out that for a write followed by a read in program order, if either the read or the write is replaced by a read-modify-write, the effect is as if the read does not bypass the preceding write. Examples that illustrate this also appear in [SUN91, SFC91].

The hardware-centric specifications of the data-race-free models allow the first optimization above if at least one of the read or write is a data operation and allow the second optimization for all data operations. As discussed in Section 6.2.4, PLpc1 allows the first optimization for the case where at least one of the read or write is a loop operation. Thus, a PLpc1 program can be run correctly on a total store ordering system if for every write followed by a read where the write and read are distinguished as either non-loop or unpairable operations, one of the write or read is made part of a read-modify-write operation. Appendix H formally proves that the above mapping to read-modify-writes is sufficient by showing that with the above mapping, total store ordering systems obey the aggressive system-centric specification of PLpc1 in Appendix G; the proof is very similar to that for the PLpc model [GAG92]. The use of read-modify-writes is necessary in general as illustrated by figure 6.9. The figure shows a program where all operations are non-loop operations; on a total store ordering system, both reads in the program could return the non-sequentially consistent result of 0 unless the reads or writes are made parts of read-modify-write operations.

It may seem unnatural to replace a non-loop or unpairable operation by a read-modify-write. It may also seem surprising that we did not apply the idea of using read-modify-writes in data-race-free programs and claim that the data-race-free models also unify total store ordering. Consider the second point of data-race-free models first. Based on the above discussion, data-race-free programs can potentially be run on a total store ordering system if for every write synchronization operation followed by a read synchronization operation in program order, at least one of them is replaced with a read-modify-write. In general, we may expect this to result in either all write synchronizations or all read synchronizations being replaced with read-modify-writes. Converting all such operations to more expensive read-modify-writes could be detrimental to the performance of a total store ordering system. Thus, while data-race-free programs can run correctly on total store ordering with the above mapping to read-modify-writes, they may not exploit the full performance of a total store ordering system and so we do not consider the data-race-free models to unify total store ordering. With the PLpc1 model, however, read-modify-

```
        Initially A = B =0

        P1              P2

        A = 1           B = 1

        ... = B         ... = A
```

Figure 6.9. Violation of sequential consistency with total store ordering and partial store ordering.

writes are needed only with non-loop or unpairable operations. The common synchronization scenarios that we examined (involving locks, barriers, and producer-consumer type synchronization) usually consisted either of loop synchronizations or already were part of read-modify-writes. For example, in all the three programs in figure 6.8, either the synchronization operations are loop operations or they are read-modify-writes. In the example programs considered for data-race-free-1 (figures 6.2 and 6.3) also, the synchronization operations are either loop synchronizations or read-modify-writes. This suggests that many common programming constructs written for PLpc1 can work unchanged on and exploit the full performance of total store ordering systems, and so we consider PLpc1 to unify total store ordering. (The above also explains why many programs written for sequentially consistent systems work correctly on total store ordering systems without any changes.)

In case a non-loop or unpairable operation are not already part of a read-modify-write, the operation needs to be replaced by a dummy read-modify-write on a total store ordering system. Thus, such a system needs to provide general read-modify-write instructions that can replace any read or write. Replacing a write with a read-modify-write requires a read-modify-write instruction that can write any value and ignore the returned value. Replacing a read with a read-modify-write requires a read-modify-write instruction that can write back the value read. In the absence of such general read-modify-write instructions, the system may need to provide an additional mechanism (e.g., a fence instruction) to impose an order on arbitrary writes followed by arbitrary reads.

The partial store ordering model is similar to total store ordering except that it allows any two non-conflicting writes in program order that are not separated by the STBAR (store barrier) instruction to be executed in parallel or out of program order. Thus, to run a program written for PLpc1 on partial store ordering, read-modify-writes need to be used as for total store ordering, and every synchronization write must be immediately preceded by a STBAR instruction (Appendix G).

Thus, programs written for PLpc1 can be efficiently run on data-race-free-1 systems and on total store ordering and partial store ordering systems with the appearance of sequential consistency (assuming general read-modify-writes for the latter two types of systems). Thus, PLpc1 programs are portable across a wide variety of hardware-centric systems.

Performance.

PLpc1 clearly allows all the performance optimizations of data-race-free-1; the data-race-free-1 implementations however cannot exploit the extra information of loop and non-loop operations provided in PLpc1 programs. Thus, PLpc1 provides strictly greater performance potential than data-race-free-1. As discussed above, PLpc1 also allows all implementations of total store ordering and partial store ordering with the appropriate mapping of non-loop and unpairable operations to read-modify-writes. PLpc1 can provide higher performance than partial store ordering since it does not require a processor to stall on any data reads; it can provide higher performance than total store ordering since it does not require a processor to stall on any data reads and does not require any two non-conflicting writes of a processor to be executed in program order. However (as with other hardware-centric models), programs written directly for total store ordering and partial store ordering could potentially execute with higher performance than a corresponding PLpc1 program by using fewer read-modify-writes and STBARS. Nevertheless, as discussed above, for many programs, PLpc1 does not require additional read-modify-writes or STBARS (e.g., programs where the only synchronization operations are barriers and locks that implement critical sections). For other programs, we expect that the above disadvantage of PLpc1 will be offset by its advantages of programmability, portability, and higher performance potential through non-blocking reads.

6.3. The PLpc2 Memory Model

6.3.1. Motivation of PLpc2

PLpc2 extends PLpc1 by exploiting even more differences between different synchronization operations. Consider again the programs in figure 6.8. All implementations of all programmer-centric models discussed so far impose the Sync-Atomicity condition, prohibiting making a synchronization write of a processor visible to any other processor until all processors have seen the write. As discussed in Chapter 2, this makes update-based cache coherence protocols inefficient for synchronization operations. An update-based protocol would benefit the synchronization writes to the flags in the producer-consumer code, the Unset synchronization writes in the critical section code, and the synchronization write to flag in the barrier code, because such a protocol would allow the corresponding synchronization reads to succeed earlier. Note also that the updates would not violate sequential consistency for any of the programs, irrespective of how many processors were executing the critical section code and the barrier code, and even if the producer-consumer code were extended to involve more than two processors. PLpc2 distinguishes certain synchronization writes that can be executed non-atomically.

To allow non-atomic synchronization writes, PLpc2 uses the properties of synchronization loops that were also used by PLpc1. These are the properties of being able to ignore the unsuccessful operations of such a loop, and the fixed order of execution of the successful read of the loop with respect to the write that makes the read successful. Based on these properties, we can show that some types of loop writes can be executed non-atomically, as long as any pair of conflicting writes containing one such write is cache coherent. We call such writes non-atomic writes. Further, all writes can be executed non-atomically if some types of reads (called atomic reads below) are executed like writes and if all pairs of conflicting writes where at least one is synchronization are cache coherent. The next section formalizes these concepts.

6.3.2. Definition of PLpc2

The formalizations of atomic and non-atomic operations are similar to those of non-loop and loop operations except that the term race in the loop/non-loop definitions is now replaced by a slightly different concept of a *partial race* and one additional constraint is added. The following first defines the notion of a partial race, then defines atomic and non-atomic reads and writes, PLpc2 programs, and the PLpc2 memory model. For the definition of a partial race, we use the notion of a *program/semi-causal-conflict graph* of an execution (similar to definition 4.14 of a program/causal-conflict graph). This is a graph where the vertices are the (dynamic) memory operations of the execution, and the edges are due to the program order relation of the execution or of the type *Write* $\xrightarrow{co}$ *Read*. The definition of PLpc2 programs below requires programmers to distinguish operations as atomic or non-atomic; again, we make it possible for programmers to always conservatively distinguish an operation, in this case, as atomic.

> **Definition 6.11:** *Partial Race:* Two conflicting operations *X* and *Y* of a sequentially consistent execution do not form a partial race iff there is at least one path between *X* and *Y* in the program/semi-causal-conflict graph of the execution such that either
>
> (1) the path begins and ends with a program order arc, or
>
> (2) the path has at least one program order arc and consists of operations to the same location.
>
> Further, if either *X* or *Y* qualifies for a data operation (i.e., does not form a race in any sequentially consistent execution after ignoring unessential operations), then at least one of the above paths is also a $\xrightarrow{hb1}$ path. If, instead, both *X* and *Y* qualify for synchronization operations, then at least one of the above paths is also a $\xrightarrow{hb0}$ path.
>
> Two operations form a partial race in an execution iff there is no path of the type described above between them.

```
              Initially X = Y = 0                          Initially X = Y = Z = 0
P1       P2               P3                   P1        P2               P3
X = 1    while (X != 1)   while (Y != 1)       X = 1     while (Z != 1)   while (Y != 1)
              {;}              {;}                            {;}              {;}
         Y = 1            ... = X              Z = 1     Y = 1            ... = X
                   (a)                                             (b)
```

Figure 6.10. Intuition for definition of partial race.

Figure 6.10 attempts to give some of the intuition motivating the above definition. The definition was developed in a somewhat ad hoc manner with the main goal of identifying operations that can be executed non-atomically on a processor consistency and release consistency (RCpc) system; Chapter 7 provides a more formal and intuitive treatment of how to identify operations that can be executed non-atomically.

Consider the program in figure 6.10(a). Suppose P_1's write of X is not executed atomically. Consider the execution sequence where P_2 returns the value of P_1's write of X, P_2 executes its write of Y, P_3 returns the value of P_2's write of Y, and then P_3 executes its read on X. Now if P_1's write of X is not executed atomically, and if P_3 originally had X in its cache, then it is possible that P_3's cache is not yet invalidated or updated, and P_3 will return the old value for X. This is a violation of sequential consistency. Note that P_1's write of X and P_3's read of X never form a race in any sequentially consistent execution; however, the path between them that is used to communicate the value of X involves the write of X itself. Such a communication requires the write to execute atomically (or the read to bypass the cache and get the new value, which can be achieved by using a read-modify-write). Consider, in contrast, a slight modification of the program, as shown in part(b) of figure 6.10. Now P_1's write of X is communicated to P_3 through a path that begins and ends with a program order arc. If each processor executes its operations one at a time, then the write of X can be done non-atomically and will not be noticed by P_3. This case is captured by part (1) of the above definition. Note also that if all the operations in the program of part (a) were to the same location (say X), then a system that maintains cache coherence would ensure that P_1's write and P_2's write are seen by all processors in the same order and so P_3 would not return the new value. Thus, again P_1's write of X can be executed non-atomically. This case is captured by part (2) of the above definition. The last part of the definition is to maintain consistency with previous models which require data operations to be ordered by $\xrightarrow{hb1}$ paths and synchronization operations to be ordered by $\xrightarrow{hb0}$ paths.

We next use the notion of partial races to formalize atomic and non-atomic operations. In the following definition, terms such as *last*, *before*, and *between* applied to operations of a sequentially consistent execution refer to the execution order on operations of that execution. Further, for part (vi) below, to model the effect of initial values of a location, we assume that there is a hypothetical write to every location that writes the initial value of the location in the beginning of the execution order.

> **Definition 6.12:** *Atomic and Non-atomic Reads.* A read R in a sequentially consistent execution E_s is a *non-atomic read* (under PLpc2) iff it does not form a partial race with any write in the execution or if it satisfies all of the following conditions.
>
> (i) R is the essential read of a synchronization loop.
>
> (ii) R forms a partial race with exactly one write, W, in the execution.
>
> (iii) R returns the value of write W.
>
> (iv) The value written by the conflicting write ordered last before W by the execution order (or the initial value if there is no other conflicting write before W) is not an exit value for R's loop.
>
> (v) If R is part of a read-modify-write, then the corresponding write does not form a partial race with another write in the execution.

(vi) Let W_2 be the last conflicting write (if any) before W whose value is an exit value of R's loop (consider the hypothetical initial write as well). If there exists a path in the program/semi-causal-conflict graph of the execution from any conflicting write W_3 between W_2 and W to R such that the path ends with a $\xrightarrow{po}$ arc, then there should be a path from W_3 to R of the type described in the definition of a partial race and that ends in $\xrightarrow{po}$ arc.[19]

A read in a sequentially consistent execution is an *atomic read* (under PLpc2) iff it is not a non-atomic read. (It follows that all non-loop reads under PLpc1 are atomic reads.)

Definition 6.13: *Atomic and Non-atomic writes.* A write W in a sequentially consistent execution E_s is a *non-atomic write* (under PLpc2) iff

(i) it does not form a partial race with any write or atomic read (as defined by definition 6.12) in E_s, and

(ii) if there exists a path in the program/semi-causal-conflict graph of E_s from W to a read R such that the path ends with a $\xrightarrow{po}$ arc and R is an atomic read in E_s, then there should be a path from W to R in E_s of the type described in the definition of a partial race and such that the path ends in $\xrightarrow{po}$ arc.

A write in a sequentially consistent execution is an *atomic write* (under PLpc2) iff it is not a non-atomic write. (It follows that all non-loop writes under PLpc1 are atomic writes.)

Definition 6.14: *PLpc2 Programs:* A program is *PLpc2* iff the program is PLpc1 and if for every sequentially consistent execution of the program such that the execution does not have unessential operations, all operations distinguished as loop or data are further distinguished as atomic or non-atomic, and the operations distinguished as non-atomic are non-atomic operations as defined by definitions 6.12 and 6.13.[20]

Definition 6.15: *The PLpc2 Memory Model:* A system obeys the PLpc2 memory model iff the result of a run of a PLpc2 program on the system is the result of a sequentially consistent execution of the program.

6.3.3. Programming with PLpc2

Figure 6.11 captures the programmers' model for PLpc2. It is similar to PLpc1 except PLpc2 allows programmers to additionally distinguish non-atomic data and loop operations from atomic data and loop operations. Again, every operation can be distinguished conservatively as non-atomic. Mechanisms for making the new distinctions are similar to those for the distinctions of the previous programmer-centric models. For examples of PLpc2 programs, consider again the PLpc1 programs of figure 6.8. All of the partial races in any sequentially consistent execution (without unessentials) of any of the programs are also races; therefore, all the operations distinguished as loop and data are also non-atomic and can be distinguished as non-atomic. To make the various distinctions, the word ''non-atomic'' can be added to each of the annotations of figure 6.8(a). For the examples in parts (b) and (c) of the figure, no change is required if the operations that hardware previously recognized as loop and data are also recognized as non-atomic.

6.3.4. Implementing PLpc2

The key additional optimization allowed by PLpc2 is that it allows all synchronization writes distinguished as *non-atomic* to be executed non-atomically, as long as cache coherence is imposed on any pair of conflicting writes where at least one is non-atomic. Thus, the Sync-Atomicity condition (Condition 5.11(b) in Section 5.3.1)

[19] Again, Chapter 7 provides the more formal treatment that motivates the definition of non-atomic and atomic reads.

[20] Implementations of PLpc2 described in this chapter do not exploit the distinction between atomic and non-atomic data writes; we retain the distinction in the definition of PLpc2 to maintain uniformity with the original PLpc model.

1. Same as PLpc1

2. For every operation distinguished as data or loop do:

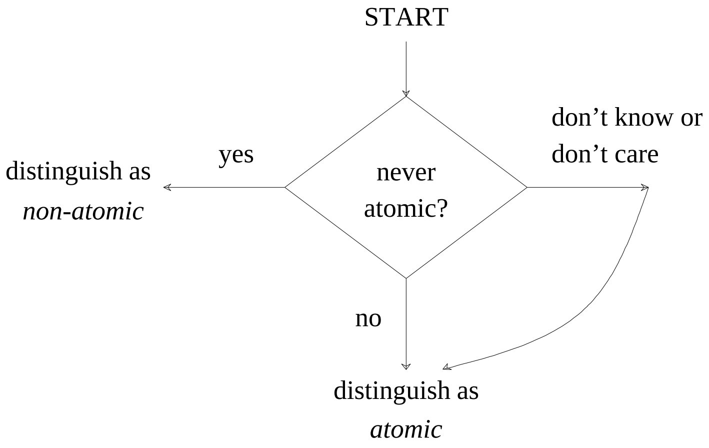


Figure 6.11. Programming with PLpc2.

is necessary only for the synchronization writes that are distinguished as unpairable, non-loop, or atomic. Unpairable synchronization writes could also be distinguished as atomic or non-atomic; however, we did not choose to make this distinction in favor of simplicity. (Non-loop operations cannot be non-atomic.)

Distinguishing data read operations as atomic and non-atomic allows an alternative of executing all synchronization writes non-atomically, as long as (1) reads distinguished as unpairable, non-loop, atomic loop, and atomic data are made part of a read-modify-write where the corresponding write for a synchronization (respectively data) read is distinguished as synchronization (respectively data),[21] and (2) conflicting pairs of writes, where at least one write is synchronization, are cache coherent. The above alternative is particularly useful in a system that does not provide any atomic writes, but does provide cache coherence and read-modify-writes. A system-centric specification of PLpc2 along with its proof of correctness appears in Appendix G.

To see the hardware performance gains possible with PLpc2, consider figure 6.8 again. As discussed before, all the data and loop operations in the figure are also non-atomic. Thus, all of those operations (including the flag writes for the producer-consumer, the Unset and Set writes for the critical section, the SyncWrites on flag for the barrier can be executed non-atomically. Thus, as mentioned before, an update based coherence protocol can be used for each of them (assuming a system with hardware-coherent caching of shared data); the corresponding pairable synchronization reads will find the location in their respective caches, and will return the successful value faster. Note also that the only operations distinguished as either non-atomic, unpairable, or non-loop are the Fetch&Inc operations in the barrier code. These are already part of a read-modify-write and so the conditions the alternative described above are already satisfied; i.e., all the programs in the figure can be run on a system that does not provide any atomic writes.

For compilers, we do not expect any additional gains from PLpc2 since the usual compiler optimizations do not affect the atomic (or non-atomic) execution of writes.

[21] The conversion to read-modify-writes can be replaced by any mechanism that makes a read ‘‘behave’’ like a write.

6.3.5. Comparison of PLpc2 with Processor Consistency, RCpc, and PLpc1 Systems

This section compares PLpc2 with PLpc1 (the most aggressive programmer-centric model so far) and with the additional hardware-centric models it unifies (release consistency (RCpc) and processor consistency).

The relation between PLpc2 and the earlier programmer-centric and hardware-centric models based on programmability is similar to the corresponding relation for PLpc1.

With regard to portability, clearly, all programs written for PLpc2 can be run on PLpc1 systems. PLpc2 further unifies the hardware-centric models of processor consistency and release consistency (RCpc) with the PLpc1 systems as follows. The optimizations allowed by processor consistency are (1) reads can bypass preceding writes, and (2) all writes are non-atomic. Release consistency (RCpc) combines the optimizations of release consistency (RCsc) and processor consistency. Both models enforce cache coherence. PLpc2 programs can be run on processor consistent systems by converting all reads that are distinguished as unpairable, non-loop, and atomic (loop or data) into read-modify-writes. Based on the discussion of PLpc1, this ensures that the effect of the first optimization is not seen for a write followed by a read that are either non-loop or unpairable (for all others, the optimization is safe). Based on the discussion of PLpc2 implementations, the mapping to read-modify-writes ensures that the effect of non-atomic writes is not seen. Appendix H gives the formal proof (based on the proof in [GAG92]) that these mappings are correct. Figure 6.12 illustrates that this conversion to read-modify-writes is, in general, necessary as follows. Processor P3's read of *X* forms a partial race with P1's write of *X* and is an atomic read. If P3's read of *X* is not converted into a read-modify-write, a processor consistent system can return the value 0 for P3's read of *X*, a non-sequentially consistent result.

```
                 Initially X = Y = 0

      P1        P2                     P3

      X = 1     while (X != 1) {;}     while (Y != 1) {;}

                Y = 1                  ... = X
```

Figure 6.12. Violation of sequential consistency with non-atomic writes.

PLpc2 programs can be run on release consistency (RCpc) systems by combining the mappings of release consistency (RCsc) and processor consistency; i.e., all operations distinguished as data for PLpc2 should be converted to ordinary, all operations distinguished as unpairable for PLpc2 should be converted to nsyncs, all operations distinguished as pairable for PLpc2 should be converted to syncs, and all reads distinguished as unpairable, non-loop, or atomic for PLpc2 should be converted to read-modify-writes. For the conversion to read-modify-write, the write of the read-modify-write for the synchronization reads should be either sync or nsync (the read should be as discussed above). (Again, figure 6.12 shows that the conversion to read-modify-writes is, in general, necessary.)

Thus, programs written for PLpc2 can be run on all PLpc1 systems and on processor consistency and release consistency (RCpc) systems. As for PLpc1 and its corresponding hardware-centric models, for many common programs, the reads that need to be converted to read-modify-writes are already present as read-modify-writes, as is the case for all the programs in figure 6.8.

With regard to performance, programs written for PLpc2 can exploit the full performance of PLpc1 systems. Further, as discussed above, many programs written for PLpc2 can exploit the full performance of processor consistent and release consistent (RCpc) systems. (Again, as for other programmer-centric models and their corresponding hardware-centric models, there are programs that may perform better when written directly for processor consistency or release consistency (RCpc) systems; but we believe the advantages of PLpc2 offset those gains.) Note that release consistency (RCpc) is the most aggressive hardware-centric system proposed so far. PLpc2, however, allows even more implementations (e.g., the aggressive implementations of data-race-free-0 in Chapter 5 and possibly even more aggressive implementations based on the system-centric specification of Appendix G).

Thus, the PLpc2 memory model unifies many commercial and academic models, resulting in advantages of programmability, portability, and performance for many of the programmers of such systems.

6.4. The PLpc Memory Model

For completeness, this section first briefly describes the PLpc model [GAG92] (the original model on which PLpc1 and PLpc2 are based) and then briefly describes why we choose to use PLpc1 and PLpc2 instead.

PLpc is based entirely on the notion of a partial race as opposed to a race.[22] PLpc calls an operation that may be involved in a partial race as competing and the other operations as non-competing. It calls all competing operations that are also atomic (by definition 6.12 and 6.13) as loop operations, and competing operations that are also non-atomic (by definitions 6.12 and 6.13) as non-loop operations. It requires that all operations be distinguished as non-competing, loop, or non-loop; further, operations distinguished as non-competing must obey the definition of non-competing operations, and operations distinguished as loop must obey the definition of either non-competing or loop operations. Figure 6.13 captures the programmers' interface with PLpc.

The information present in a PLpc program is almost similar to that present in a PLpc2 program. An exception is that PLpc does not distinguish between pairable and unpairable synchronizations; however, that distinction can be easily added and is not the key issue. Thus, the knowledge required of the programmer to provide (non-conservative) information for PLpc and PLpc2 is almost the same. The advantage of PLpc when compared to PLpc2 is that it seems much simpler (although, again, the knowledge required of the programmer for maximum performance gain is the same for both models) However, PLpc1 and PLpc2 may be more desirable than PLpc in terms of programmability and portability for the following two reasons.

First, programs written for the data-race-free models do not contain any information that can be used by PLpc since PLpc is based entirely on information regarding partial races whereas the data-race-free programs contain information on races. Thus, if a program written for one of the data-race-free models must be moved to a PLpc system, it must be re-analyzed for any performance optimizations. PLpc1 and PLpc2 eliminate this problem by expressing most of the required information in terms of races rather than partial races. Thus, data-race-free programs can run unchanged on PLpc1 and PLpc2 systems, and these systems can exploit all the information that can be exploited by a data-race-free system for such a program.

Second, the information exploited by PLpc is motivated by two different optimizations: reordering some synchronization writes followed by some synchronization reads, and executing some synchronization writes non-atomically. PLpc gives the simpler specification that allows *both* these optimizations *and* the optimization of reordering data operations and executing data writes non-atomically that are allowed by the previous programmer-centric models. This specification requires programmers to reason about partial races for *any* optimization. We chose to separate the information required for all the optimizations so that only familiarity with the notion of a race would be sufficient for as many optimizations as possible. This is desirable since reasoning about partial races is more difficult than reasoning about races; determining whether two conflicting operations will race just requires determining if the operations can execute ‘‘one after another or consecutively’’ whereas determining whether two conflicting operations form a partial race requires reasoning about whether certain paths exist between the operations. Similarly, the definition of loop operations under PLpc is also more complex than that under PLpc1.

Thus, the first advantage of defining PLpc in terms of PLpc1 and PLpc2 is that it allows programmers to analyze their programs more incrementally, allowing simpler information to be used for some optimizations and requiring more difficult information for only aggressive optimizations. The second advantage is that (assuming data-race-free and PLpc systems will both exist in the future) programs written for data-race-free systems can be run unchanged on PLpc systems with potentially the same performance as a data-race-free system; for exploiting the higher performance of the PLpc systems, programmers can further analyze their programs incrementally. Note that an alternative approach for the second advantage would have been to define the data-race-free models also in terms of partial races. However, as discussed above, reasoning about partial races may be more difficult than rea-

[22] PLpc, however, does not require the last part regarding $\xrightarrow{hb1}$ and $\xrightarrow{hb0}$ paths in the definition of a partial race.

1. Write program assuming sequential consistency

2. For every memory operation specified in the program do:

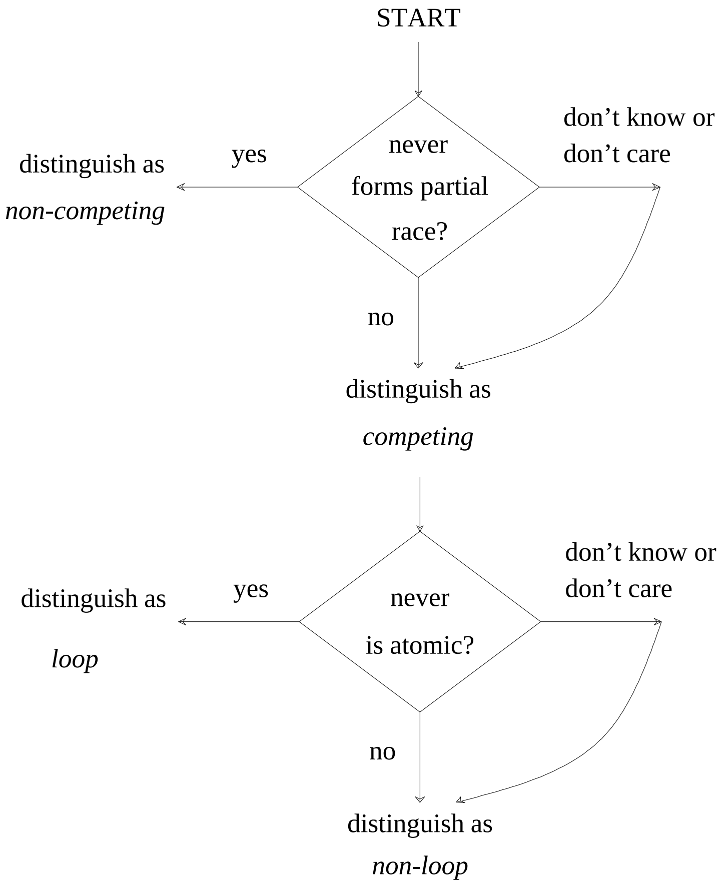


Figure 6.13. Programming with PLpc.

soning about races and so it does not seem worthwhile to change the simpler data-race-free models.

6.5. Comparison of PLpc Models with IBM 370 and Alpha

From Chapter 2, the IBM 370 model can be viewed as either a combination of processor consistency and weak ordering, or a combination of total store ordering and weak ordering. In either case, PLpc2 can be viewed as unifying this model with the other hardware-centric models as well.

The Alpha model is difficult to compare with the SCNF models because it does not obey any form of a control requirement needed by our system-centric specifications. We believe, however, that some form of a control requirement is essential for Alpha for the following reason. The Alpha reference manual informally states protocols that ensure reliable communication. For example, it states that data sharing is reliable if a processor updates a data location, executes an MB, writes a flag, and then another processor reads the new value of flag, executes an MB, and then reads the updated data location. In the absence of some form of a control requirement, however, the informal protocols (including the above) are not sufficient to guarantee correct communication in all cases [Hor92]. The specified protocols are similar to those for PLpc1 and PLpc2 programs, if we interpret certain uses of MB instructions as certain operation types. Specifically, if we place an MB before every write operation distinguished as synchronization, an MB after every read distinguished as synchronization, and an MB between every write synchronization followed by a read synchronization where either is a non-loop or unpaired operation, and if we assume the control requirement, then Alpha systems obey PLpc1 and PLpc2. Thus, with the above assumptions, the PLpc models can be viewed as unifying the Alpha model as well. (If we require an MB between every write synchronization followed by a read synchronization, then the Alpha systems also obey data-race-free-1.)

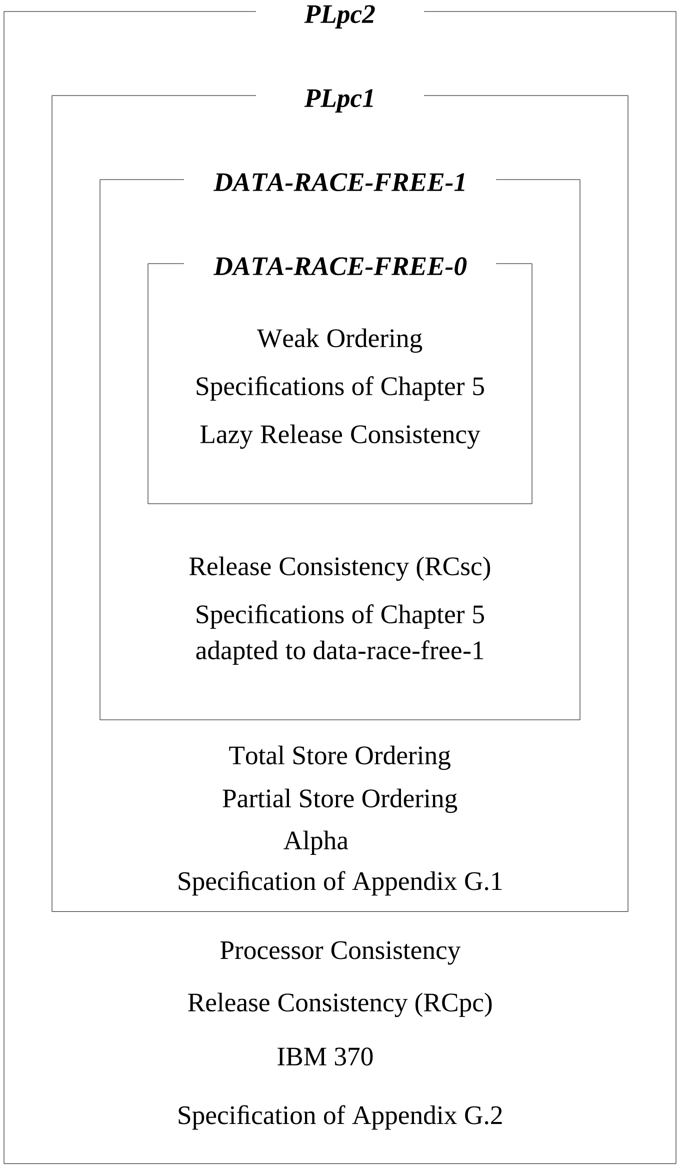

PLpc2
PLpc1
DATA-RACE-FREE-1
DATA-RACE-FREE-0
Weak Ordering
Specifications of Chapter 5
Lazy Release Consistency
Release Consistency (RCsc)
Specifications of Chapter 5
adapted to data-race-free-1
Total Store Ordering
Partial Store Ordering
Alpha
Specification of Appendix G.1
Processor Consistency
Release Consistency (RCpc)
IBM 370
Specification of Appendix G.2

Figure 6.14. Summary of SCNF Models.

For lazy release consistency, assume that all synchronization reads are distinguished as acquires and all synchronization writes are distinguished as releases. For IBM 370 and Alpha, assume interpretations of Section 6.5.

6.6. Discussion

We have discussed four SCNF models so far that exploit increasing amounts of information and unify increasingly many hardware-centric systems. Figure 6.14 summarizes this contribution by showing the programmer-centric models and their relation with various hardware-centric systems and with each other. The data-race-free-0 model unifies implementations of weak ordering, lazy release consistency (assuming all synchronization reads are acquires and all synchronization writes are releases), and other implementations discussed in Chapter 5. The data-race-free-1 model unifies data-race-free-0 systems with release consistency (RCsc) and other implementations based on the methods of Chapter 5. The PLpc1 model unifies data-race-free-1 systems with total store ordering and partial store ordering systems, and with Alpha systems (assuming the interpretation of Section 6.5). The PLpc2 model unifies PLpc1 systems with processor consistency and release consistency (RCpc) systems, and with IBM 370 systems (assuming the interpretation of Section 6.5).

The SCNF models address the 3P criteria as follows.

Programmability: Programmers can reason with sequential consistency. They need to provide increasing amounts of information about the program, but this can be done incrementally and conservatively.

Portability: A program written for any of the four SCNF models can be run correctly on any of the systems of figure 6.14. Further, it is possible to design future SCNF systems so that the performance for a program is proportional to the information in the program, no matter which SCNF system it runs on. For this, SCNF systems should not penalize conservative distinctions (e.g., it should be possible to delay a non-loop read for a preceding non-loop write without requiring expensive read-modify-writes).

Performance: Each of the four SCNF models allows more implementations than the old hardware-centric models it unifies. For many programs, this leads to higher performance potential than previous hardware-centric models. Each SCNF model has higher performance potential than the previous less aggressive SCNF model for all programs.

Chapter 7

The Design Space of SCNF Memory Models

The four SCNF models discussed so far unify several hardware-centric models and present a wide spectrum of performance potential and programmability. Other researchers have proposed further relaxations of these models; e.g., the entry consistency model requires programmers to associate locks with the data they protect [BeZ91, BZS92], Gibbons and Merritt allow programmers to associate a release with a specific subset of operations that it releases [GiM92], Carlton suggests associating locks with the data they protect and allocating them in the same memory module [Car91]. A natural question that arises next is: are there other models that can potentially give higher performance and/or better programmability than the current models? This chapter answers the above question. Rather than simply propose yet another better memory model, however, this chapter aims to explore and characterize the design space for memory models.

With the SCNF view, a memory model is simply a way to obtain help from the programmer to identify when certain optimizations can be performed without violating sequential consistency. Therefore, the question of whether we can design other memory models with higher performance reduces to addressing the following issues.

- Are there other potential optimizations that have not been considered by, or fully exploited by, current models?
- Given an optimization, can we determine the cases where it can be performed safely (i.e., without violating sequential consistency)?
- How can the programmer communicate the safe cases for the optimizations to the system?

Conversely, often certain aspects of program behavior are already known because programmers generally write well-structured programs. Then the question of designing an appropriate memory model reduces to determining what optimizations are guaranteed to be safe with the given information, and how this information can be made explicit.

This chapter shows that there indeed are optimizations and well-defined programming constructs that have not been fully exploited by previous models, and shows how the programmer can help the system to exploit these optimizations and constructs. To exploit any general optimization or program information, we develop a mapping between optimizations and program information. This mapping determines the program information that will allow the safe use of a given optimization, and conversely determines optimizations that can be performed safely with given program information. This is an important contribution because in the past, determining when a certain optimization was safe to use involved lengthy and complex proofs that required reasoning about executions on the optimized system and proving that such an execution would appear sequentially consistent [AdH90b, AdH93, GLL90, GMG91, GiM92]; our mapping largely eliminates that complexity. The mapping between program information and optimizations leads to a characterization of SCNF memory models that determines their performance and programmability. We define a generic memory model and a system-centric specification for the generic model in terms of this characteristic. Our common characterization for memory models is similar to the MOESI [SwS86] and $\mathrm{Dir}_i[B|NB]$ [ASH88] characterizations of cache coherence protocols that unified several seemingly disparate protocols and exposed the design space for more protocols.

Our overall approach is to first identify a system-centric specification for sequential consistency, and then use it to characterize when an optimization will not violate sequential consistency. Section 7.1 gives such a specification based on several previous works [AdH92a, Col84-92, GAG92, LHH91, ShS88]. Section 7.2 uses the above specification to deduce the mapping between optimizations and information. Section 7.3 uses the mapping of Section 7.2 to examine several optimizations and common program constructs. For each optimization, it shows how the optimization can be applied safely to more cases than allowed by previous models and how the programmer can communicate these cases to the system, thereby resulting in new memory models. For the programming constructs, it shows how more optimizations than previously considered can be applied to the constructs and how programmers can make such constructs explicit, again leading to new models. Section 7.4 uses the results of Section 7.2 to deduce a key characterization of SCNF memory models that determines their performance and programmability, and characterizes the design space in terms of a generic model. Section 7.5 discusses implementations of the generic memory model. Our work extends and is related to many previous studies. Section 7.6 discusses the related work and its relationship to our work. Section 7.7 concludes the chapter and discusses the limitations of this work.

As in previous chapters, we continue to use the terms *preceding* and *following* to indicate program order.

7.1. A Condition for Sequential Consistency

Section 7.1.1 first gives a simple system-centric specification for sequential consistency. Section 7.1.2 makes four observations that modify the specification to reflect certain optimizations. Section 7.1.1 and Observation 1 of Section 7.1.2 follow directly from previous work by Shasha and Snir [ShS88], Collier [Col84-92], and Landin et al. [LHH91]. Observations 2 and 3 of Section 7.1.2 are extensions of similar concepts developed for the PLpc model [GAG92].

7.1.1. A Simple Condition

We use the notions of conflict order ($\xrightarrow{\text{co}}$) and program/conflict graph defined earlier for a sequentially consistent execution and reproduced below for a general execution.

> **Definition 7.1:** *Conflict Order ($\xrightarrow{co}$):* Let X and Y be two memory operations in an execution. $X \xrightarrow{\text{co}} Y$ iff X and Y conflict and $X(i)$ is ordered before $Y(i)$ by the execution order for some i.

> **Definition 7.2:** The *program/conflict graph* for an execution E is a directed graph where the vertices are the (dynamic) operations of the execution and the edges represent the program order and conflict order relations on the operations.

For the rest of this section, assume for simplicity that the number of instruction instances ordered before any instruction instance by program order is finite for any execution considered below. Call this the *finite speculation assumption.* Practically, the assumption means that we do not allow processors to speculatively execute a memory operation until it is known that all preceding loops will terminate. The assumption does not prohibit programs that might execute infinite instructions (e.g., operating systems); it only restricts speculative execution beyond potentially unbounded loops. Also assume that if a write operation is in an execution, then all its sub-operations are in the execution. Call this the *write termination assumption*. Practically, this assumption is automatically obeyed in systems that have atomic memory or that use a hardware cache coherence protocol to perform a write in the memory copy of every processor. Section 7.5 will alleviate these restrictions.

A simple system-centric specification for sequential consistency (with the finite speculation and write termination assumptions) is that the program/conflict graph of the execution should be acyclic (as also observed by others in various forms [Col84-92, LHH91, ShS88]). The following first illustrates the above condition with an example, and then gives a formal proof of correctness. Figure 7.1(a) shows code in which processor P0 writes location x and then location y. Processor P1 reads y and then x into its registers r1 and r2 respectively. Consider a system where P0 could execute its writes out of program order. Then it is possible for P1 to read y after P0 modifies it, but to read x before P0 modifies it. Figure 7.1(b) gives the corresponding execution order. We assume that memory operations are executed atomically and give the execution order on operations rather than sub-operations. Figure 7.1(c) shows the program/conflict graph. There is a cycle in this graph and so the execution does not obey the above system-centric specification. Indeed, the execution does not have the same result as a sequentially consistent execution (assuming the value returned by reads comprises the result), and the system does not appear sequentially consistent. Note that if the read of x returned the new value written by P0, then the cycle would be broken, the execution would obey the above specification, and the system would appear sequentially consistent. The formal specification and proof follow.

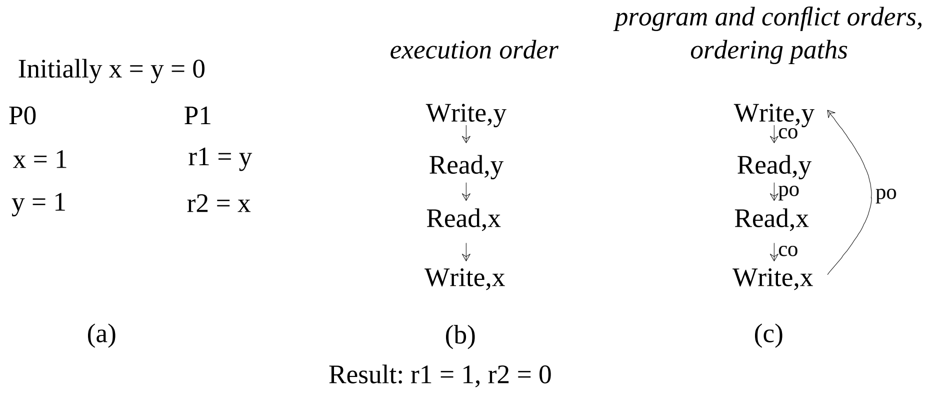


Figure 7.1. An example where sequential consistency is violated.

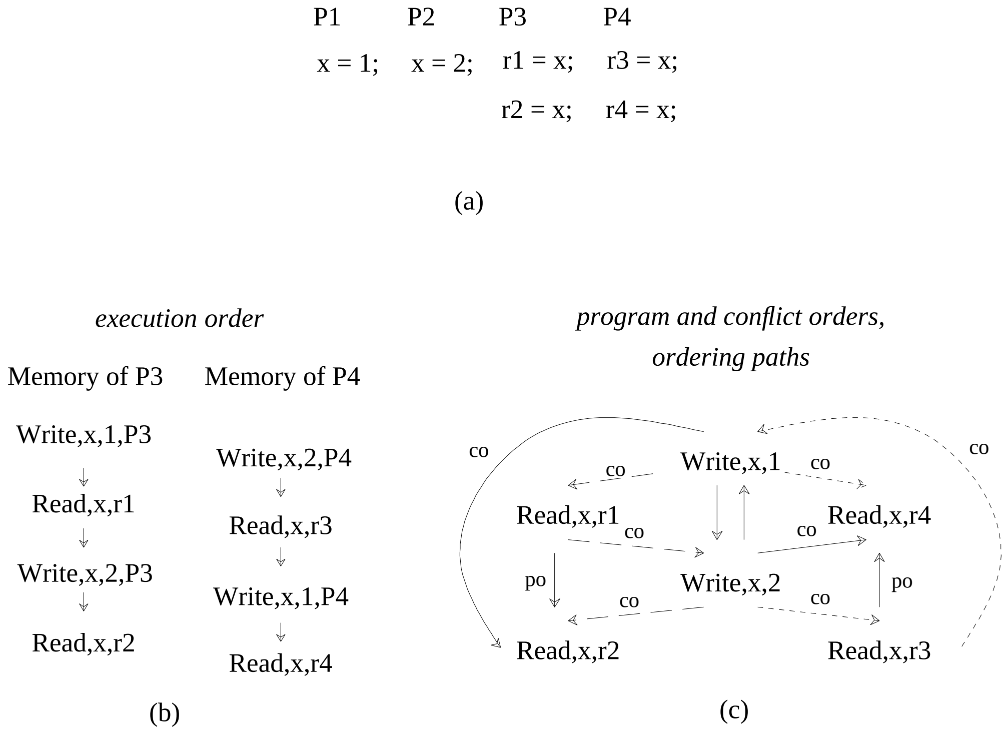
Condition 7.3: *System-centric specification for sequential consistency:* The program/conflict graph of the execution must be acyclic (assuming finite speculation and write termination assumptions).

Proof: An acyclic program/conflict graph of an execution implies a total order on the memory operations of the execution such that the total order is consistent with program order, a read in the execution returns the value of the last conflicting write before it by this order, and there is no conflicting write between a read and write of a read-modify-write. Thus, this total order qualifies for an execution order of a sequentially consistent execution (Condition 3.12 in figure 3.4). Therefore, the instruction set, operation set, and value set of *E* along with the above total order as the execution order represent a sequentially consistent execution that has the same result as *E*. Thus, an acyclic program/conflict graph is a correct system-centric specification for sequential consistency. □

We are interested in a slightly different form of the above specification that is more amenable to further optimizations. This form considers two types of paths in the program/conflict graph and requires that conflicting sub-operations of operations ordered by such a path be executed in the same order as the path. We discuss the two types of paths and then formalize the specification below. Both types of paths are between conflicting operations.

The first type of path is a path that has at least one program order edge. For example, in figure 7.1 (c), there is a path of the above type from the write of *x* to the read of *x* through the write and read of *y*. The specification requires the sub-operation of the write of *x* in the memory copy of P1 to appear before the sub-operation of the read of *x* in the execution order. This is not true for the execution order of figure 7.1 and indeed the execution does not appear sequentially consistent. Imposing the condition would require that the read of *x* return the value of the write of *x* by P1.

Considering only the above types of paths is sufficient if writes are executed atomically. In the presence of non-atomic writes, we need to consider another type of path as follows. Consider two conflicting operations such that there is no path between them with at least one program order edge in the program/conflict graph. Such operations were called *race* operations in definition 4.12 in Section 4.1.2; we call the pure conflict order path between them as a *race path*. With non-atomic writes, it is possible that the conflicting sub-operations of two race writes in different memory copies appear in a different order in the execution order. This can be detected by the programmer, exposing the underlying non-atomicity and non-sequentially consistent behavior. To prohibit this, the specification below also requires that conflicting sub-operations of writes ordered by a race path execute in the same order in all memory copies. As an example, consider figure 7.2. It shows two processors writing the same location, *x*, and two other processors reading the location twice. Assume the reading processors see the writes of *x* in different orders. Figure 7.2(b) shows parts of the corresponding execution order (on sub-operations) relevant to our discussion. The left and right sides show sub-operations in the memory copies of processors P3 and P4 respectively; Write,x,1,Pi denotes a write of *x* with value 1 in the memory copy of processor Pi and Read,*x*,rj denotes a read of *x* into register rj. Note that the write sub-operations appear in different orders in the two



Result: r1 = 1, r2 = 2, r3 = 2, r4 = 1

Figure 7.2. Motivation for considering race paths.

memory copies. Part (c) shows the corresponding program/conflict graph; the dashed and dotted lines trace the execution order of conflicting sub-operations in the memory copies of P3 and P4 respectively. There is no path in the program/conflict graph that contains at least one program order edge and for which the condition mentioned above is violated. However, there are still cycles in the program/conflict graph, and the execution does not give sequentially consistent results. Requiring that conflicting sub-operations of two writes ordered by a race path be executed in the same order eliminates the above cycle and precludes the given execution.

The following formalizes the above concepts and the system-centric specification.

Definition 7.4: A path in the program/conflict graph from operation X to operation Y is called a *race path* iff $X \xrightarrow{co} Y$ and there is no path from X to Y in the program/conflict graph that contains at least one program order edge.

Definition 7.5: An *ordering path* is a path in the program/conflict graph that is between two conflicting operations, and either it has at least one program order edge or it is a race path between two writes.

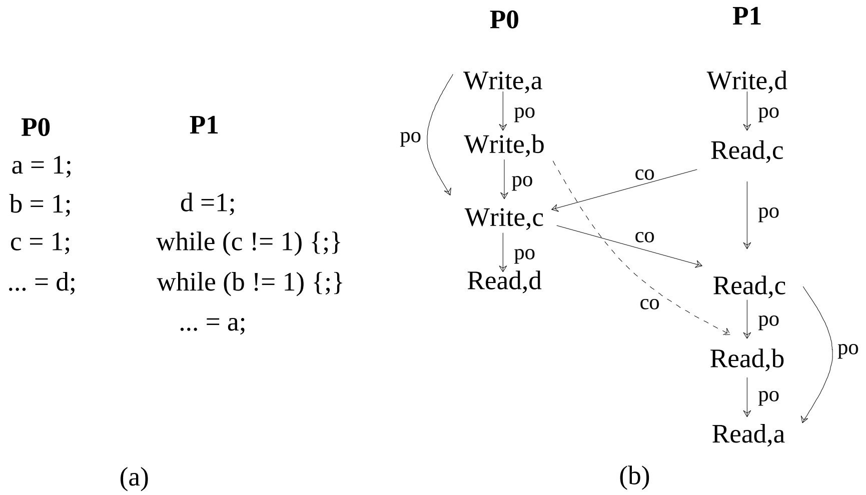
Condition 7.6: *System-centric specification for sequential consistency:* For every pair of conflicting memory operations X and Y in an execution, if there is an ordering path from X to Y, then $X(i)$ is before $Y(i)$ for all i in the execution order (assuming finite speculation and write termination assumptions).

Proof: The proof follows directly from observing that the specification prevents any cycles in the program/conflict graph, thereby obeying Condition 7.3. □

7.1.2. Modifications to the Condition for Sequential Consistency

This section first makes four observations that lead to a less restrictive form of the system-centric specifications of the previous sub-section, and then gives the modified system-centric specification based on the four observations.

The following motivates observations 1, 2, and 3. Condition 7.6 requires that if there is an ordering path from operation *X* to operation *Y*, then the sub-operations of *X* must appear before the corresponding conflicting sub-operations of *Y* in the execution order. The simplest way to achieve this is to ensure that all operations on an ordering path are executed one at a time in the order specified by the conflict and program order edges of the path. On a system with atomic memory, operations on a conflict order edge are, by definition, already executed atomically and in the order of the edge. On a system with non-atomic memory, the same effect can be achieved by executing writes on conflict order edges of ordering paths atomically. The additional condition to satisfy Condition 7.6 is to ensure that operations on a program order edge of an ordering path be executed in program order. For example, consider the program and execution represented in Figure 7.3, where all program order edges are on ordering paths except the one between the reads of *c* by processor P1. Requiring operations on program order edges of ordering paths to be executed one at a time in program order implies that all operations of P0 and all operations of P1 except its reads of *c* must execute one at a time in program order.



```
    P0            P1
  a = 1;
  b = 1;        d =1;
  c = 1;      while (c != 1) {;}
 ... = d;     while (b != 1) {;}
                 ... = a;
```

(po = program order, co = conflict order)

Figure 7.3. Example for observations 1, 2, and 3.

The four observations below show that to satisfy Condition 7.3, certain ordering paths (also present in Figure 7.3) need not be considered by the system. Thus, the system can exploit more parallelism and execute more writes non-atomically, because it need not constrain operations on the ordering paths identified by the following observations.

Observation 1.

The first observation (also made in [ShS88] in a different form) is that if there are multiple ordering paths from *X* to *Y*, then the system needs to ''be careful'' when executing the operations of only one of those ordering paths. For example, in Figure 7.3(b), there are three ordering paths from the write of *a* to the read of *a*:

$\text{Write},a \xrightarrow{po} \text{Write},b \xrightarrow{po} \text{Write},c \xrightarrow{co} \text{Read},c \xrightarrow{po} \text{Read},b \xrightarrow{po} \text{Read},a;$

$\text{Write},a \xrightarrow{po} \text{Write},b \xrightarrow{co} \text{Read},b \xrightarrow{po} \text{Read},a;$ and $\text{Write},a \xrightarrow{po} \text{Write},c \xrightarrow{co} \text{Read},c \xrightarrow{po} \text{Read},a.$

Serializing operations on program order edges of all the above paths would require P0 to execute its three writes one at a time. However, by the above observation, it is sufficient to serialize operations on program order edges of only the third path, allowing P0's writes to *a* and *b* to be executed in parallel. (The writes to *b* and *c* need to be done one at a time for the ordering path between operations to *b*.) Similarly, P0's write to *b* can be executed non-atomically (although this example does not illustrate the performance gains that may be possible with such an optimization).

Observation 2.

The second observation, based on work for PLpc [GAG92], is that certain instructions and operations, called *unessential* operations, can be removed from the execution, leaving behind instructions and operations that also form an execution and with the same result as the original execution. Consequently, the system need not consider ordering paths resulting from unessential operations. For example, consider the while loops in Figure 7.3(a) which execute a read until the read returns some value. The loops are only used for control; as long as the loop finally terminates on the correct value, the number of times the loop executed unsuccessfully does not matter and the actual values returned by the unsuccessful reads are not useful. These unsuccessful reads can be removed from the execution and what remains is still an execution with the same result as the original execution; hence, these reads are unessential. The ordering path from the write of *d* to the read of *d* results from the unessential read on *c*; therefore, this path can be ignored and the system need not restrict the operations on this path.

A case of unessential operations was identified for the PLpc model [GAG92] (and a conservative definition was used for PLpc1 and PLpc2 in Chapter 6). This chapter allows operations from more general types of loops where the termination of the loop may require several reads to different locations to return certain values. The definition, given below, is a straightforward extension of definition 6.6 used for the PLpc models. A synchronization loop is a sequence of instructions that are repeatedly executed until an exit predicate is satisfied. The exit predicate of the loop specifies reads of one or more memory locations (called exit reads) and requires each read to return one of several values (called exit values of the read). Like PLpc, an exit read may be part of a read-modify-write, but only if it is the only read specified by the predicate. The write of the read-modify-write is called an exit write. The other constraints below are the same as for the aggressive definition for PLpc given in [GAG92]; they ensure that all but the last iteration of the loop can be ignored in every execution and the only shared-memory operations in the last iteration are the exit reads and writes. Like PLpc, the only writes from a synchronization loop that are allowed to change the value of any memory location are the exit writes from the last iteration of the loop. With non-atomic writes, an additional system assumption is required to ensure that the other writes of the loop do not change the state of any memory copy: if $W_1 \xrightarrow{\text{co}} W_2$ and either W_1 or W_2 is from a synchronization loop, then $W_1(i) \xrightarrow{\text{xo}} W_2(i)$ for all *i*. We call this the *loop coherence assumption.*

> **Definition 7.7:** A *synchronization loop* is a sequence of instructions in a program that would be repeatedly executed until a predicate (called the exit predicate) based on the values returned by certain shared-memory reads (called the exit reads) in the sequence is satisfied. The exit predicate is satisfied iff each exit read returns one of certain specified values called the exit values. An exit read may be part of a read-modify-write iff it is the only exit read on which the exit predicate depends. In that case, the write corresponding to the exit read-modify-write is called an exit write. Below, a loop refers to a synchronization loop.
>
> (1) For an instantiation of a loop[23] in any execution, the exit predicate, the locations to be accessed by the exit operations, and the corresponding exit values cannot be changed by any instruction instance of the instantiation; the loop instantiation terminates iff all the exit reads in an iteration return their exit values in which case this is the last iteration; the exit write (if any) writes a value which is a fixed function of the value returned by the corresponding exit read and this function cannot be changed by any instruction instance of the instantiation; every iteration of the loop instantiation terminates and contains all the exit reads for the exit predicate of the instantiation.
>
> (2) For an instantiation of a loop in any execution, the values of registers or private memory changed by the instruction instances of the instantiation cannot be accessed by any instruction not in this instantiation, and the only way instruction instances in the instantiation can affect an output interface is

[23] By instantiation of a loop, we mean instruction instances from every iteration of the loop beginning (in program order) from the first iteration since the predicate of the loop was last satisfied and until the next iteration where the predicate of the loop is satisfied.

by writing the values returned by the exit reads of the final iteration to the output interface.

(3) For any instantiation of a loop in any execution, the first instruction instance program ordered after the instruction instances of the loop is from the same instruction.

(4) For an instantiation of a loop in any execution, if conditions for the exit predicate to succeed persist in memory, then the loop eventually exits.

(5) For an instantiation of a loop in any execution, the only writes in the loop that change the value of a shared-memory location are the exit writes of the iteration in which the exit predicate is satisfied.

(6) In any execution, the only shared-memory operations in the iteration of a loop where the exit predicate is satisfied are exit reads and exit writes.

(7) Every instantiation of a loop terminates in every sequentially consistent execution.

Definition 7.8: *Unessential and Essential Operations:* A set of operations in an execution are *unessential* if they are from an iteration of a synchronization loop where the exit predicate is not satisfied. All operations that are not unessential are *essential*.

Parts (1) - (6) of the definition of a synchronization loop require examining *any* execution, as opposed to just sequentially consistent executions. However, each part requires examining just the control flow graph of a loop of a single processor's code in isolation; this does not require examining non-sequentially consistent interactions between two processors. The purpose of the various conditions is to ensure that irrespective of the values returned by the shared-memory reads of the loop, the unsuccessful iterations of the loop can be ignored. The conditions are trivially satisfied if the only code in the loop is the exit reads and writes, the evaluation of the exit predicate (without any writes), and a conditional jump to a fixed instruction out of the loop or back to the beginning of the loop depending on whether the predicate is satisfied or not.

Observation 3.

The third observation is that there are certain reads that will always execute after certain writes irrespective of how the operations on the ordering path between them are executed by the system (assuming ordering paths from previous conflicting writes are executed correctly). Therefore, ordering paths between such a write and read need not be considered by the system. For example, consider the final read of *b* in the while loop of Figure 7.3. In Figure 7.3(b), there is an ordering path from the write to the read of *b*. However, the loop was waiting for the value written by the write; therefore, no matter how the system behaves with respect to the ordering path between the write and read, the final read will always execute after the write. Therefore, no explicit system restrictions are required to impose this particular order. Such a read is called self-ordered.

Definition 7.9: The essential exit reads from synchronization loops are *self-ordered* reads. Synchronization loops are also called *self-ordered loops*.[24]

Observation 4.

The final observation uses the notion of consecutive conflicting operations.

Definition 7.10: Two conflicting operations in an execution are called *consecutive* conflicting operations iff there is an ordering path or race path from one of these operations (say *X*) to the other operation (say *Y*), and no such path from *X* to *Y* has another write on it that conflicts with *X* and *Y*.

[24] Although so far essential reads from synchronization loops and self-ordered reads are defined in the same way, we will make a distinction between the two concepts for two reasons. First, for some models, we may want to exploit observation 2 but not observation 3 for some types of loops; therefore, we would like to identify system constraints for exploiting observation 3 separately. Second, in the future we may be able to extend the definitions of synchronization loops and self-ordered loops such that they are no longer identical. We, however, make the reasonable assumption that a loop that is exploited as a self-ordered loop is also exploited as a synchronization loop; therefore, all self-ordered loops are synchronization loops and all constraints on synchronization loops are also implicitly applicable to self-ordered loops.

It follows that as long as ordering paths between consecutive conflicting operations are maintained, all ordering paths will be maintained. This does not lead to optimizations, but allows for a simpler analysis since we do not need to analyze ordering paths between conflicting operations that are not consecutive.

Modified Condition.

Based on the above four observations, we next define a critical set[25] and modify condition 7.6. An explanation of the definition and condition follows next. For simplicity, we give the definition assuming writes are atomic; i.e., the execution order can be defined on operations rather than sub-operations. Below, terms such as last and after refer to the ordering by the execution order. Further, for parts (2) and (3) of the definition below, we need to consider initial values of a location. To model the effect of initial values, assume that there is a hypothetical write to each memory location that writes the initial value of the location at the beginning of the execution order. (Hypothetical writes are not considered unless explicitly mentioned.)

> **Definition 7.11:** A *critical set* for an execution is a set of ordering paths that obey the following properties. Let *X* and *Y* be any two consecutive conflicting operations such that there is an ordering path or race path from *X* to *Y*. Ignore all unessential operations.
>
> (1) If *Y* is not a self-ordered read and if there is an ordering path from *X* to *Y*, then one such path is in the critical set.
>
> (2) If *Y* is an exit read from a synchronization loop that is not self-ordered and if there is no ordering path from *X* to *Y*, then let *W* be the last write (including the hypothetical initial write) before *X* that conflicts with *Y* and writes an exit value of *Y*. If *W* exists, then let *W′* be a write after *W* that has an ordering path to *Y* that ends in a program order arc. If *W′* exists, then one ordering path from the last such *W′* to *Y* that ends in a program order arc must be in the critical set.
>
> (3) If *Y* is a self-ordered read, then let *W* and *W′* be as defined in (2) above. If *W′* exits, then one ordering path from *any W′* to *Y* that ends in a program order arc must be in the critical set.
>
> For every execution, we consider one specific critical set, and call the paths in that set as *critical paths*.

For non-atomic writes, parts (2) and (3) of the above definition must consider write *sub-operations* that conflict with *Y*'s *sub-operation*. The modified definition is given in Appendix C.

> **Condition 7.12:** *System-Centric Specification of Sequential Consistency:* The execution should have a critical set such that if an ordering path from *X* to *Y* is in the critical set, then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all *i* (assuming finite speculation, write termination, and loop coherence).

The formal proof of the above specification appears in Appendix C. The following informally explains the specification. A critical set is the set of ordering paths that are not excluded by the four observations of this section as follows. The definition exploits observation 2 by ignoring all unessential operations and exploits observation 4 by considering only consecutive conflicting operations. Part (1) of the definition exploits observation 1 by choosing only one ordering path between a pair of conflicting operations to be critical. Part (2) of the definition is an artifact of exploiting observation 2 and ignoring unessential operations, and is necessary only because later we will want to restrict ourselves to analyzing only sequentially consistent executions (see Section 7.2).[26] Part (3) of the definition exploits observation 3. Observation 3 is that self-ordered reads will always execute after the correct write as long as they execute after the necessary previous writes that do not write an exit value. Thus, even if there

[25] The term ‘‘critical’’ is inspired from the work by Shasha and Snir [ShS88].

[26] We will later use information from only sequentially consistent executions to identify the critical paths of any execution (see Section 7.2). Part (2) of definition 7.11 then ensures that a synchronization loop does not terminate too early as follows. Consider the operations *W′* and *Y* in part (2). For an acyclic program/conflict graph, *Y* must execute after *W′*. However, if returning the value of *W′* would make *Y* unessential, then *Y* will execute after the next write that makes it essential (in this case *X*). Thus, in any sequentially consistent execution *W′* and *Y* are never consecutive conflicting operations. If there is no ordering path from *X* to *Y*, then to ensure that *Y* executes after *W′*, we need to explicitly consider the path from *W′* to *Y*.

is an ordering path from *X* to *Y*, a critical path for this case is necessary only if there is also an ordering path from a previous write that writes a non-exit value. In that case, *any* ordering path from the non-exit value write or *X* (as opposed to just from *X*) could be critical. (The reason for requiring a critical path even when there is no ordering path from *X* to *Y* is the same as for part (2); however, again, this path can be chosen from more writes than allowed by part (2).)

Our definition of a critical set contains the minimal ordering paths we have ascertained so far. It may be possible to reduce this set in the future. (One example appears in Appendix C.) We expect the rest of this work to be applicable to such reductions. We next use the above condition to determine how to design SCNF memory models.

7.2. Designing An SCNF Memory Model

An SCNF memory model solicits help from the programmer to facilitate system optimizations without violating sequential consistency. Call an optimization *safe* iff it does not violate sequential consistency. Many optimizations are safe to use for certain parts of the program, but not for others. For example, the optimization of reordering operations of a processor is safe when applied to two data operations of a data-race-free-0 program, but may not be safe when applied to two synchronization operations. SCNF models allow optimizations to be used selectively based on the information provided by the programmer. Thus, an important aspect of designing an appropriate SCNF model for a system is determining when an optimization is safe to use and how the programmer can help the system detect these safe cases. Conversely, often some information may already be known about the program either because of the limited synchronization libraries and parallel programming constructs provided by the system or because programmers tend to write well-structured programs. We would like to be able to determine optimizations that would not violate sequential consistency with this known information. This section uses Condition 7.12 to establish a mapping between optimizations and information that allows the safe use of optimizations.

From Condition 7.12, an optimization is safe as long as all conflicting sub-operations ordered by a critical path are executed in the order of the critical path. We say that the system executes an ordering path *safely* if it executes two conflicting sub-operations ordered by the path in the same order as the path; otherwise, we say the system *violates* the ordering path. Therefore to use an optimization safely, the system needs to selectively apply the optimization to *only the parts that will not violate the critical paths of the execution.* The information that the programmer can provide to allow an optimization is to *distinguish parts of the program where the optimization will not violate any critical paths.* Conversely, if some information about the critical paths of the program is already known, then the system can apply any optimizations that will not violate those critical paths.

The above is a correct mapping between optimizations and information; however, it is not sufficient for our purpose. Specifically, we use the term programmer to refer to either the compiler or the human programmer. To give the above information, the compiler requires a global data dependence analysis which can be quite pessimistic and/or inefficient. (This approach has been suggested by Shasha and Snir as discussed in Section 7.6.1.) Therefore, we expect that the human programmer will have to give the necessary information on the critical paths. In that case, we can only expect information pertaining to the critical paths of sequentially consistent executions since making the programmer reason about other executions defeats the goal of SCNF models. Thus, it is important to determine when an optimization is safe to use based on information from *only sequentially consistent executions*. A large part of the complexity in designing an SCNF memory model arises because of this restriction. This work largely eliminates the complexity by providing a general relationship between optimizations and when they are safe to use based on information about the critical paths of sequentially consistent executions of a program. Specifically, we develop a pre-condition for the system and show that if the system obeys this pre-condition, then the programmer need only provide information pertaining to critical paths of sequentially consistent executions. We call this condition the *control requirement*. The control requirement is dependent on the optimizations the system will employ and is formalized in Sections 7.4 and 7.5. Since it is satisfied by most currently practical systems, for now, we will assume that the system satisfies it.

Thus, *assuming a system obeys the control requirement, if the programmer distinguishes the parts of the program where an optimization will not violate a critical path of a sequentially consistent execution, the system can apply the optimization safely to the distinguished parts.* Consequently, if some information on the critical paths of sequentially consistent executions is already available due to other reasons discussed above, then the system can apply any optimizations that will not violate such critical paths.

The work in designing a memory model is now restricted to characterizing when an optimization will not violate the critical paths of a sequentially consistent execution, and determining how the programmer can communicate this information. Conversely, for known information about the program, the work in designing a memory model is now restricted to converting this information into information on critical paths and determining when an optimization will not violate these critical paths. As we will see in the next section, this work is far less complex than proving from scratch when an optimization will not violate sequential consistency. Specifically, the complex proofs for previous memory models [AdH90,AdH92,GLL90,GMG91,GiM92] can now be done in a few steps as illustrated by Appendices F and G. Also note that expressing the required information in terms of the behavior of sequentially consistent executions means that not only programmers but also future designers of memory models need worry about the effect of optimizations on only sequentially consistent interactions of the program. Thus, when reasoning about programs, writes can henceforth be assumed to be atomic and the definition of conflict order used in the previous section can be replaced by the previous definition (Definition 4.10 of Section 4.1.2) assuming atomic writes; consequently, the notions of program/conflict graph, unessential operations, self-ordered operations, consecutive conflicting operations, critical paths, and execution order no longer assume non-atomic operations. Particularly, note that consecutive conflicting operations are now simply conflicting operations not separated by another conflicting write in the execution order. Determining whether an optimization might violate a critical path of a sequentially consistent execution and implementing the resulting model, however, still requires reasoning with non-atomicity; to aid this process, we provide generic system-centric specifications for generic memory models in Section 7.5.

We illustrate the use of our mapping between information and optimizations in the next section by considering several optimizations and common programming constructs to explore a large part of the design space of SCNF models.

7.3. New SCNF Memory Models

This section considers optimizations and common programming constructs that are not fully exploited by previous work. For the optimizations, the approach is to begin with a general, sequentially consistent base system and then examine optimizations that would relax some of the constraints imposed due to sequential consistency. We reason about cases where the optimizations can be used safely and how the programmer can indicate to the system when these optimizations can be used safely. The requirements for the programmer to indicate when the optimizations can be used safely is the memory model for the system. Every optimization we consider leads to one or more new memory models. For the programming constructs, we consider common uses of the constructs, reason about the critical paths that result from such uses, and deduce optimizations that do not violate those critical paths. This leads to new and simple to use models. Such models prescribe how the construct should be used and the only constraint on the programmer is to use the construct as prescribed.

The advantage of beginning with optimizations directly is that it is easier to envisage potential optimizations starting from a constrained system. However, the disadvantage is that the information required of the programmers deduced using this approach may not be directly related to high-level sharing patterns and synchronization behavior. The advantage of beginning with common program constructs is that the constraints on programmers are easier to describe and satisfy; the disadvantage is that this approach is restricted to the sharing patterns that can be identified in current programs.

Sections 7.3.1 and 7.3.2 respectively discuss models based on the above two approaches.

7.3.1. Models Motivated by Optimizations on Base System

Our approach is to begin with a general, sequentially consistent base system and then examine optimizations that would relax some of the constraints imposed due to sequential consistency. Specifically, we consider a system with a general interconnection network which may or may not have caches. In the presence of caches, we assume some form of a hardware cache coherence protocol [ASH88].

In practice, such a system needs to adhere to the following to appear sequentially consistent [ScD87]. A processor does not issue a memory operation until the preceding operations complete. Thus, a processor must wait until a preceding read returns a value and a preceding write reaches memory. For systems with caches, a processor must also wait until all the caches that have the line accessed by a preceding write are invalidated or updated (using a cache coherence protocol). To enable the processor to detect when its write has reached memory or when the relevant caches have received invalidations or updates, the memory and/or the caches usually need to send acknowledgements to the processor. Further, systems with caches also need to ensure that writes appear to execute atomically. For this, first, a cache coherence protocol is used to serialize writes of all processors to the same line. Second, a processor is not allowed to read the value of a write until all other processors have seen that write. For an update-based cache coherence protocol, this requires two phases. In the first phase, memory sends updates to caches and receives acknowledgements. In the second phase, memory sends messages to the caches indicating they can now allow their processors to read the value.

It follows that, in general, sequentially consistent systems do not use the uniprocessor optimizations of write buffers, out-of-order issue of memory operations (as with superscalar processors), and pipelined (or overlapped in the inter-connection network) memory operations. In addition, they incur unnecessary network traffic and latency by requiring invalidations/updates, acknowledgements, and serialized conflicting writes. If an update based cache coherence protocol is used, a two phase protocol is required which decreases the potential performance gain of an update protocol.

Below, we consider the optimizations of out-of-order issue, pipelining (with in-order issue), eliminating the second phase of an update protocol, and eliminating acknowledgements. The first three optimizations are allowed by previous SCNF models for some cases; we show we can apply them to more cases with more help from the programmer leading to new models. The last optimization has not been considered by previous models in the presence of general interconnection networks.

7.3.1.1. Out-Of-Order Issue

This section investigates when it is safe to execute two operations out of program order. Consider the example in Figure 7.4. Ignore the program and conflict order edges for now. The program represents processor P1 producing separate data for processors P2 and P3 to consume. Variables Flag1 and Flag2 are used to indicate when the data is ready. For processor P1, previous SCNF models allow some of its operations to be executed out-of-order. However, all the implementations we considered imposed some unnecessary ordering constraints. For example, all of them require P1's write of Flag2 to wait for the completion of the write of Flag1. Intuition suggests that this delay is unnecessary because both operations seem to be involved in ''independent'' tasks of producing independent data for different processors. This intuition is correct and we can use our results to formalize a memory model that will exploit it as follows.

From the previous section, two operations can be executed out of program order as long as this does not violate a critical path. Out-of-order execution of operations that are not on a program order edge of any critical path cannot violate any critical path. Thus, two operations that are not on a program order edge of a critical path can be executed out of program order.

In Figure 7.4, we overload the instructions of the program to also represent memory operations of an execution of the program. Certain program and conflict order edges of executions are shown. The paths between conflicting operations (with at least one program order edge) constitute a critical set of ordering paths. Now it is clear why P1's writes of Flag1 and Flag2 can be reordered: it is because these operations are not on a program order edge of any critical path.

Thus, to reorder operations, we need the programmer to identify program order edges that are not on critical paths. This requires a mechanism in the programming language and hardware that will allow the programmer to distinguish program order edges that are not on critical paths from those that are on critical paths (or for which this information is not known to the programmer). We use the following terminology adapted from message-passing nomenclature.

An operation that lies on a conflict order edge of a critical path is called a *communicator*; others are called *non−communicators*. For example, in Figure 7.4, only operations on Flag1 and Flag2 are communicators. We call the first operation on a conflict order edge a *sender* and the second operation a *receiver*. The write on Flag1 is a sender and the read of Flag1 is a receiver. If a sender is also on a program order edge, we say the sender *sends* the first operation on that edge. Analogously, if a receiver is also on a program order edge, then we say the receiver *receives* for the next operation on that edge. The write of Flag1 sends the writes of A and B. The read of

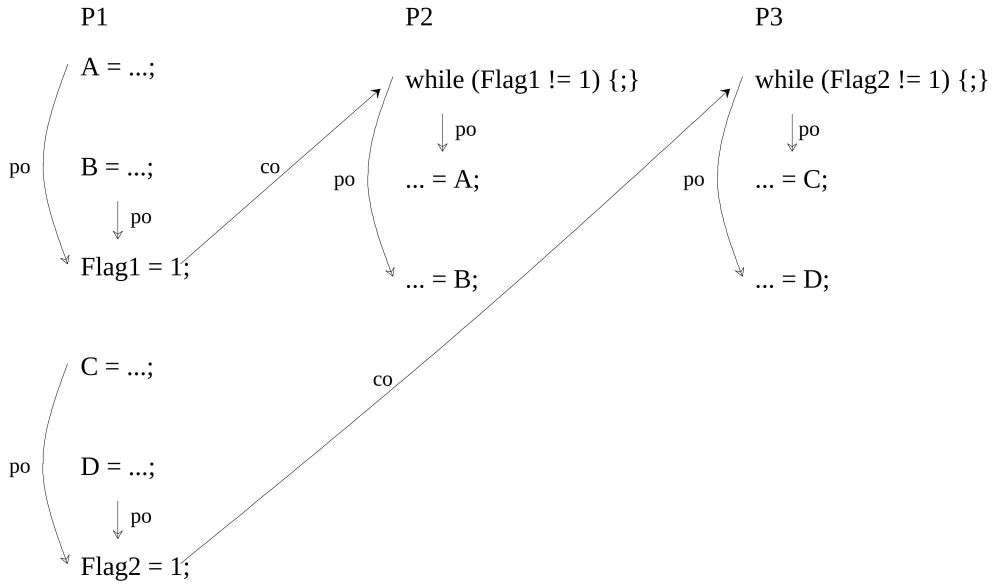


Figure 7.4. Reordering memory operations.

Flag1 receives for the reads of A and B.

The use of message-passing terminology is intentional and suggests an intuitive interpretation of operations on a critical path. In message passing systems, processors communicate by using explicit sends and receives. In shared-memory systems, processors communicate through conflicting accesses to a common memory location. In the absence of additional information, a shared-memory system must assume that any pair of conflicting accesses are implicitly involved in a communication. The conflict order edges of critical paths indicate the true (as opposed to potential) communication in a shared-memory execution; we therefore call the first operation on such an edge as a sender and the second operation as a receiver. Further, we interpret edges of the type $op_1 \xrightarrow{po} S \xrightarrow{co} R \xrightarrow{po} op_2$ on a critical path as the sender *S* sending ''information'' about the operation op_1 to the receiver *R*, and the receiver *R* passing this information on to operation op_2. Thus, a critical path can be considered as a sequence of sends and receives that pass a message about the first operation on the path to the last operation on the path. Making critical paths explicit makes this communication explicit to the system. Note that senders and receivers on the critical path can be either reads or writes (as long as at least one operation per conflict order edge is a write), because any pair of conflicting operations can communicate information in a shared-memory system.

A mechanism to distinguish program order edges not on critical paths from those that are on critical paths is to associate every operation with the operations it sends and the operations that receive for it. Then the associated operations are on critical edges. For a model that uses such a mechanism, the programmer must associate with each memory operation all other operations sent by the operation and all receivers for the operation. Then the system can execute the operation out of order if all its associated operations are already executed. This model exposes more operations that can be reordered than the SCNF models we have proposed so far. For example, the write to Flag2 can be executed before the writes to A and B as well as before the write to Flag1. Optimizations similar to those of the above model are suggested by Bershad et al. [BeZ91, BZS92] and Gibbons and Merritt [GiM92]; we discuss how they compare to our models at the end of this sub-section.

The above mechanism may be difficult for programmers to use and difficult for system designers to implement. We can define a simpler, intermediate model that is not as aggressive as the above but still allows more reordering than the previous models. Specifically, it allows Flag2 to be executed before Flag1. This model is based on the notion of distinguishing between different operation types also used by the earlier SCNF models. The following describes the operation types that the model distinguishes; the actual mechanism for distinguishing operations can be any of those described for the previous models in Chapters 4 and 6.

op_2	**op_1**	
	Non-Communicator	*Communicator*
	Condition when op_2 can bypass op_1	
Non-Communicator	Always safe	op_1 does not receive for op_2 *OR* op_1 does not receive for non-communicators
Communicator	op_2 does not send op_1 *OR* op_2 does not send non-communicators	op_2 does not send op_1 and op_1 does not receive for op_2 *OR* op_2 does not send communicators and op_1 does not receive for communicators

(a) Let op_1 be before op_2 by program order. The table states when op_2 can be executed before op_1.

Mechanism	**Supported by**		
	DRF0	*DRF1*	*PLpc1, PLpc2*
Distinguish:			
non-communicators	data	data	data
communicators that do not receive for non-communicators	-	release, unpaired sync	release, unpaired sync
communicators that do not send non-communicators	-	acquire, unpaired sync	acquire, unpaired sync
communicators that do not send communicators	-	-	loop read
communicators that do not receive for communicators	-	-	loop write

(b) Useful mechanisms based on distinguishing operations

Table 7.1. Mechanisms for reordering memory operations.

Table 7.1(a) specifies several characterizations for when an operation op_2 can bypass a previous operation, op_1, of its processor. The table gives four cases depending on whether op_1 and op_2 are communicators or non-communicators. For example, the first entry states that if both op_1 and op_2 are non-communicators, it is always safe for op_2 to bypass op_1. This motivates a mechanism to distinguish non-communicators from communicators. The remaining entries first specify an aggressive characterization requiring mechanisms to associate operations as discussed above, and then the corresponding more conservative characterization requiring mechanisms to distinguish operations.

Table 7.1(b) summarizes the conservative mechanisms motivated by Table 7.1(a). For a model that uses these mechanisms, the programmer must ensure that an operation distinguished as in Table 7.1(b) does indeed obey the corresponding specification. For example, an operation distinguished by the programmer as a non-communicator should indeed be a non-communicator. A system obeying such a model can execute out-of-order all pairs of operations characterized by the conservative parts of Table 7.1(a) (using the identification by the programmer).

Table 7.1(b) also shows the mechanism that each of the SCNF models discussed in earlier chapters supports and the corresponding terminology the model uses for the particular type of operations. (The proofs for the models in the appendices clarify why the correspondence is true.) PLpc1 and PLpc2 are the most aggressive since they provide more mechanisms than the other models. However, they support the last two mechanisms in a limited way, and so are less aggressive than the new model which supports the last two mechanisms in the most general way. Specifically, PLpc1 and PLpc2 distinguish communicators which do not send other communicators from those that do; however, they do so only for read communicators. In Figure 7.4, the write to Flag1 is a write communicator that does not send other communicators. The PLpc models do not provide a mechanism for the programmer to indicate that this write does not send communicators. Therefore, the PLpc models cannot use any information from the programmer to determine that the write of Flag2 need not be executed before the write of Flag1. The above more general model allows the programmer to provide this information and allows the writes of Flag1 and Flag2 to be reordered.

The models resulting from the mechanisms given by Table 7.1(b) also require fairly detailed reasoning from programmers about the behavior of the programs. Are there other easier to identify characterizations of program order edges that are not on critical paths? The earlier SCNF models use such characterizations as shown in Table 7.1(b): the data and loop characterizations are simpler to identify than the corresponding more general distinctions in the leftmost column of the table. Section 7.3.2 examines models based on common programming constructs that make some of the more aggressive information also easier to obtain.

Finally, we compare the optimizations of the above models to the aggressive models of entry consistency proposed by Bershad et al. and an aggressive version of release consistency proposed by Gibbons and Merritt. The optimizations of allowing the write of Flag2 in Figure 7.4 to be executed before the write of Flag1 and before the writes of A and B are allowed by the definition of the entry consistency model [BeZ91] as well. The write of Flag2 is allowed before the write of Flag1 because entry consistency does not impose any ordering constraints on synchronization operations of the same process (the Flag operations qualify as synchronization). However, reordering any arbitrary pair of synchronization operations can violate sequential consistency; e.g., consider a program where all operations are distinguished as synchronization. Thus, by allowing this optimization, entry consistency has to forgo sequential consistency (even for the programs it supposedly allows). Entry consistency allows the optimization of the write of Flag2 to execute before the writes of A and B. The cost, however, is that it requires every access to a shared variable to be protected by a lock and to be executed within an acquire and release of this lock, thereby restricting the use of common synchronization techniques such as the use of task queues (further illustrated in Section 7.3.2.3).

Gibbons and Merritt have proposed an aggressive version of release consistency where the programmer can associate a release (or sender) operation with data operations that it must release [GiM92]. However, the system they propose does not allow overlapping two releases (or senders). Thus, just by associating releases with appropriate operations, the system they propose would not execute the writes of Flag1 and Flag2 in parallel. Their framework, however, allows program order to be a partial order per process. With that, it would be possible for the programmer to indicate directly to the system that the last three operations of P1 are not program ordered after the first three operations of P1, and the system can execute these operations in parallel. However, this mechanism may not be general enough for some cases. For example, there may be dependences between any of the above operations that may be inconvenient to remove in the high-level program, thereby prohibiting the relaxation of program order. Further, at a very fine-grain, it may be difficult for programmers to determine when to relax program order and to express this relaxation; our work gives programmers a way to reason about whether a set of operations might be safely executed in parallel and also suggests conservative mechanisms that make it easier to express this parallelism.

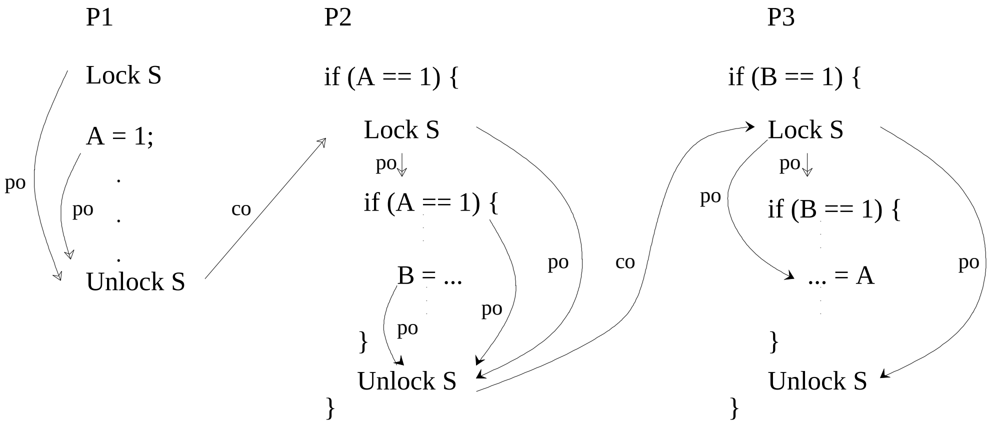
7.3.1.2. Pipelining Operations (With In-Order Issue)

We next consider systems where processors issue memory operations in program order, but we would like to pipeline (in the inter-connection network) as many operations as possible. The observation of the previous section holds here as well; i.e., only operations that form a program order edge of a critical path cannot be pipelined. However, for a processor that does not issue operations out-of-order, this requires that in Figure 7.4, processor P1 should stall on its write of Flag1 until it receives the acknowledgements for its writes of A and B. Our condition for sequential consistency requires only that critical paths be executed safely. In this case, it requires only that P1's writes of A and B be executed only before P2 reads those locations. Previous work has shown how to pipeline P1's writes of A, B, and Flag1 while still ensuring the above [AdH90, AdH92b, BeZ91, Car91, Col84-92, KCZ92, LHH91]. However, except for the work by Carlton [Car91], all the previous methods either require considerable hardware or software complexity or are applicable only to highly restricted (acyclic or ring) networks. Carlton proposed (without proof) that if all data is protected by locks, a lock and the data it protects are put in the same memory module, and the network does not reorder accesses along the same path, then all operations can be pipelined [Car91]. We use Carlton's approach, but argue formally to show how it can be applied to only a few accesses, eliminating the need for only lock-based synchronization and the need to allocate a lock variable and all data associated with it in the same memory module.

Consider hardware that inherently preserves the order of some (but not all) operations. For example, consider a system that does not have caches (or where certain locations can be marked uncacheable) and where there is no reordering of operations from a given processor to a given memory module (i.e., network guarantees pairwise order). Then if A, B, and Flag1 were in the same memory module, they could be pipelined and would still be executed in memory in program order (since P1 issues them in program order). P2 then would always return the updated values for A and B. This pipelining can be done by the hardware if it knows that all the operations that Flag1 sends are for its own memory module. Thus, a useful model for this scenario would be one that provides a mechanism in the programming language for programmers to associate with senders the operations they send. The compiler could then allocate the locations of the associated operations in the same memory module (if possible) and communicate that to the hardware. This communication could be done, for example, by distinguishing the above senders simply as non-communicators. Since hardware does not wait on a non-communicator for previous non-communicators, hardware will not wait on Flag1 for the operations on A and B. A similar scheme for cache based systems can exploit the fact that most cache coherence protocols ensure that operations to the same cache line are seen by all processors in the same order. Note that this model needs to associate senders with operations they send only at the high level programming language level, not at the hardware level. So, at the hardware level, this model is different from previous models and that described in the previous section.

7.3.1.3. Single Phase Update Protocols

We next consider eliminating the second phase of update protocols; i.e., we would like a read to return the value of a write even while other processors could return the value of a previous write. Previous SCNF models identify two cases of writes where this is possible: data writes for all models and the loop non-atomic writes for the PLpc models. (The PLpc models also allow all writes to be non-atomic if some operations are made read-modify-writes.) We use our methodology to characterize another case where non-atomic writes are safe without the use of read-modify-writes, resulting in a new memory model.



```
P1                  P2                          P3

Lock S              if (A == 1) {               if (B == 1) {
A = 1;                 Lock S                      Lock S
   .                   if (A == 1) {               if (B == 1) {
   .                     B = ...                     ... = A
   .                   }                           }
Unlock S               Unlock S                    Unlock S
                    }                           }
```

Figure 7.5. Non-atomic writes.

Consider Figure 7.5, which shows three processes sharing data through critical sections. However, to avoid unnecessary lock accesses, P2 and P3 first check (outside the critical section) if some data they want to access has the desired value. These values are updated by P1 and P2 respectively. If the values are not as desired, they do not need to do the lock access. Otherwise, they access the lock, but read that data again to ensure they get the latest value. The figure also shows the critical paths.

P1's write on A races with P2's first read on A which is not in a loop. Therefore, that write does not qualify for a single phase update protocol with the other models. Therefore, the implementations of those models discussed so far do not allow P2's first read on A to return the new value until the update for A reaches P3 (assume P3 has A cached from a previous phase of the program). The purpose of this delay is to ensure that P3 does not later read the old value of A. However, the critical path from P1's write of A to P3's read of A ensures that P3 will return the new value of A even if P2's first read completes before the write. Thus, intuitively, a two phase update protocol is not needed for this write. The following shows how to formally translate this intuition to characterize such a case so the programmer can indicate it to the system.

In the following, we say that a write, W_a, executes atomically if no read R returns the value of W_a or the value of a conflicting write ordered after W_a in the execution order, until W_a completes; i.e., if $W_a \xrightarrow{co} R$, then $W_a(i) \xrightarrow{xo} R(j)$ for all i, j. Otherwise, we say the write executes non-atomically.

Proposition 7.13: Executing a write W with a single-phase update protocol (or non-atomically) does not violate a critical path if (a) there is no critical path of the type $W \xrightarrow{co} R \cdots R'$ where R and R' are different reads, or (b) there is no critical path with W as a receiver (assuming operations on a program order edge of a critical path are executed serially in program order and conflicting writes are coherent).

The following analysis shows that the above proposition is true.

Analysis: We first examine critical paths that either begin with a program order arc, or begin with a *Write* $\xrightarrow{co}$ *Read* arc and end with a read. Let $A_1, A_2, \cdots, A_n$ represent the operations on the path. The following induction shows that given the system assumptions in the above proposition, for all A_k, either (1) $A_k = A_1$, or (2) $A_1(i) \xrightarrow{xo} A_k(j)$ for all i,j, or (3) $A_1(i) \xrightarrow{xo} A_k(j)$ for all i and some j. In addition, for (3), A_k must be a write and there exists a read R on the critical path such that the part of the path from R to A_k consists only of writes (except R) and $\xrightarrow{co}$ arcs.

Base case: Holds trivially for $n = 1$.

Induction for $n = k+1$.

Case 1: $A_k \xrightarrow{po} A_{k+1}$ (i.e., $A_k(i) \xrightarrow{xo} A_{k+1}(j)$ for all i, j): It follows that A_{k+1} obeys (1) or (2).

Case 2: $A_k \xrightarrow{co} A_{k+1}$ (i.e., $A_k(i) \xrightarrow{xo} A_{k+1}(i)$ for all i) and A_k obeys (2): It follows that A_{k+1} obeys (3) when A_k is a read and A_{k+1} is a write, and obeys (2) otherwise.

Case 3: $A_k \xrightarrow{co} A_{k+1}$, A_k obeys (3) and A_{k+1} is a write: It follows A_{k+1} obeys (3).

Case 4: $A_k \xrightarrow{co} A_{k+1}$, A_k obeys (3), and A_{k+1} is a read: The conditions necessary for A_k to obey (3) imply that A_k is a write and is a receiver on the path. Therefore, by proposition 7.13, A_k must be executed atomically and so $A_k(i) \xrightarrow{xo} A_{k+1}(j)$ for all i,j. It follows that A_{k+1} obeys (2), completing the induction.

We need to show that $A_1(i) \xrightarrow{xo} A_n(i)$ for all i. This clearly follows if A_n obeys (2). If A_n obeys (3), it must be a write. If A_1 is a write, the proposition follows. If A_1 is a read, then suppose $A_n(i) \xrightarrow{xo} A_1(i)$ for some i. Since A_n obeys (3), it is executed atomically, and so $A_n(i) \xrightarrow{xo} A_1(j)$ for all i,j. But since A_n obeys (3), there is a read R on the path such that $A_1(i) \xrightarrow{xo} R(j)$ for all i, j and $R(j) \xrightarrow{xo} A_n(j)$ for some j, a contradiction to the result above that $A_n(p) \xrightarrow{xo} A_1(q)$ for all p, q.

We next consider the critical paths not considered above; these must consist of a chain of $\xrightarrow{co}$ arcs from (say) X to Y followed by a path from Y to (say) Z of the type considered above, and such that X and Y conflict. Applying the above analysis to the path from Y to Z, it follows that $Y(i) \xrightarrow{xo} Z(i)$ for all i. It must also be that $X(i) \xrightarrow{xo} Y(i)$ for all i and so it follows that $X(i) \xrightarrow{xo} Z(i)$ for all i. □

A write satisfies cases (a) and (b) of the proposition if it is never a communicator on a critical path. In Figure 7.5, the write of A by processor P1 is not on a communicator on any critical path; therefore, it does not need a two phase update protocol. Thus, a model that allows more single phase updates than previous models is one that provides a mechanism to distinguish writes that are either non-communicators (or more generally that obey (a) and (b) above) or writes that are non-atomic loop writes (as defined by PLpc2). (The data writes identified by previous models for single phase updates are a subset of non-communicator writes.) Call the above writes as *non-atomic+* writes. Then the programmer's requirement for such a model is that writes distinguished as non-atomic+ should obey the above conditions. The system can execute all writes distinguished as non-atomic+ with single phase updates.

As for the optimization involving reordering, Section 7.3.2 examines common programming constructs that contain writes that obey the above characteristics but are easier to identify.

7.3.1.4. Eliminating Acknowledgements

Finally, we consider eliminating acknowledgements for some writes with general interconnection networks.[27] Previous work has considered this optimization [Car91,Col84-92,LHH91], but either assumes restricted networks [Col84-92, LHH91], or does not ensure sequential consistency (as in some schemes in [Col84-92, LHH91]), or requires restricted, lock-based synchronization [Car91]. Only the work by Carlton [Car91] indicates (without proof) how this optimization might be used for an SCNF model, but requires restricted, lock-based synchronization. Acknowledgements are needed only to indicate the completion of a write operation. The completion of a write operation needs to be indicated only if some later operation will wait for this write. Thus, to determine when acknowledgements are not required, we need to characterize writes for which no other operation waits. The analysis in the previous sub-sections makes this characterization obvious. These are writes that are not sent by a sender, that do not receive for any operation, and that are non-atomic+. Call such writes *non-ack* writes. An example of such a write is the write of Flag1 in Figure 7.4. Thus, a memory model that will allow some acknowledgements to be eliminated is one that provides a mechanism to distinguish non-ack writes from others. The requirement of the programmer is simply to distinguish a write as non-ack only if it satisfies the above mentioned properties. Again, we will see easier-to-identify characterizations for such writes in Section 7.3.2.

7.3.2. Models Motivated by Common Programming Constructs

This section proposes new memory models based on common programming constructs. The goal of this section is not to determine new optimizations or new cases where the optimizations of the previous section are safe. Instead, the goal of this section is to determine safe cases that are easy to identify for the programmer. For this, the section examines easy-to-identify high-level programming constructs and determines the optimizations they allow. For constructs that allow useful optimizations, the memory model requires only that the high-level programmer make this easy-to-identify construct explicit. We show how the high-level programmer can make the construct explicit to the compiler. We assume, however, that there are underlying mechanisms for the compiler to convey this high-level information to the hardware, and for the hardware to perform the optimizations possible with this information. The previous section discussed such mechanisms for the various optimizations. Note that the compiler can directly exploit all the optimizations cited in this section (that are relevant to the compiler) without any additional mechanisms.

The following sub-sections examine constructs involving producer-consumer synchronization, barriers, locks, and constructs used to decrease lock contention. For some of the new models, the analysis may seem lengthy; however, all the analysis in this section is far shorter and simpler than any of the original proofs for the earlier SCNF models [AdH90,AdH92,GLL90,GMG91,GiM92].

Below, we use terms such as *executes after*, *executes before*, *before*, *after*, *between*, and *last* to indicate the ordering by execution order. Since we need to analyze only sequentially consistent executions, we assume the ex-

[27] Ross Johnson brought the possibility of such an optimization to our notice, as an optimization examined by the Scalable Coherence Interface (SCI) extensions group.

ecution order on operations. The terms *following* and *preceding* still indicate program order.

7.3.2.1. Producer-Consumer Synchronization

In a producer-consumer interaction, a producer process typically generates some data and then, using a write to a shared-memory location, signals to the consumer that the data is ready. The consumer process meanwhile reads the above shared-memory location in a loop, waiting for the read to return the value to be written by the signal. Call the loop an await loop, its reads as await reads, and the values the loop waits for as exit values. With the PLpc models, the usual use of signal writes and await reads would constitute loop operations, allowing certain optimizations. This section examines a slightly more general case of an await loop, and other possible optimizations on the signal and await operations. More generally, an await loop may wait for more than one shared-memory location to reach an exit value. Figure 7.6 illustrates an example where the loop waits for all of the locations to reach certain values. We investigate such loops below.

```
P1              P2              P3

                                /* spin until both flags are 1 */

                                while ( (Flag1 != 1) ||
                                        (Flag2 != 1) ) {;}

A = 52;         C = 721;        ... = B;

B = 46;         D = 543;        ... = D;

Flag1 = 1;      Flag2 = 1;      ... = C;

                                ... = A;
```

Figure 7.6. Producer-consumer interaction.

For the following analysis, consider a program for which every sequentially consistent execution is divided into phases, where each phase ends with a barrier and locations accessed by barrier operations are not accessed by any other operations. Thus, for two conflicting operations in different phases, there exists an ordering path consisting of only barrier operations (other than the operations being ordered). Assume such a path to be chosen as the critical path. Also assume that for two conflicting barrier operations, the critical path chosen (if any) consists only of barrier operations.

For the await loop described above, a useful (and intuitively safe) optimization is parallel execution of the reads of an iteration of the await loop. The data-race-free and PLpc models require distinguishing all await reads as synchronization and none of their described implementations would allow the above optimization. The following uses our framework to determine the properties of an await loop that would make the above optimization safe, and then gives the corresponding model that allows the above properties to be made explicit. The section concludes with other optimizations possible with the new model.

Optimization of parallel execution of reads in an await loop: Analysis.

Two reads in program order can be executed in parallel if they do not form a program order edge of any critical path. By the assumption discussed above, two reads from an await loop cannot be on a critical path ordering operations from different phases. Thus, we need only be concerned with critical paths ordering operations from the same phase. Denote the two reads from an await loop as R_1 and R_2 where $R_1 \xrightarrow{po} R_2$. For this program order arc to be on a critical path, R_2 should either be a sender, or R_1 should receive for R_2. The following two paragraphs show that with a few reasonable assumptions, neither of the above conditions is true.

For R_2 to be a sender, in some sequentially consistent execution, there must be a write in the same phase as R_2, such that the write conflicts with R_2 and executes after R_2. It is reasonable to expect that there will be only one write to the location of R_2 in the entire phase and this write writes the exit value for R_2 (as in figure 7.6).

Since R_2 must be essential, it follows that R_2 must return the value of the above write, and so no conflicting write can execute after R_2 in the same phase. Thus, R_2 cannot be a sender.

We next investigate if R_1 can receive for R_2, given the assumptions already made to ensure that R_2 is not a sender. If R_2 is not a sender, then a critical path with $R_1 \xrightarrow{po} R_2$ must end on R_2, and so it must begin with a write W (in the same phase) that conflicts with R_2. The described await loop is self-ordered. Therefore, the path from W to R_2 need not be chosen as critical. However, we must ensure (by definition of a critical set) that if there is a conflicting write W' before W that writes an exit value of the loop, then some ordering path (to R_2) from a write between W' and W is critical. The current phase cannot have any other writes that conflict with W (by the assumptions of the previous paragraph); therefore, any W' must be in a previous phase and any writes between W' and W already have a critical path to R_2. Thus, we only need to ensure that if a W' exists, then another write between W' and W that writes a non-exit value exists. We can ensure this by requiring that when a phase begins, no location accessed by an await loop in the phase has an exit value. Thus, the path from W to R_2 is not critical and R_1 cannot be a receiver for R_2.

Optimization of parallel execution of reads in await loop: The model.

The above analysis motivates a model that recognizes special signal-await constructs shown on the left side of figure 7.7 with semantics shown on the right side of the figure. The signal construct specifies a location and a value. Semantically, it represents a write to the given location updating it to the given value. The await construct specifies one or more locations (called the await locations) and a predicate for each await location that is a function of the value of the location. Semantically, an await repeatedly reads all the specified locations until each read returns a value that satisfies the corresponding predicate. The model allows programmers to use the constructs to represent the given semantics only if they obey the following constraints for every sequentially consistent execution.

(1) A phase with an await construct should begin with the await locations in a state that would not satisfy the corresponding await predicates, and an await loop should eventually terminate.

(2) If a phase has an await, then exactly one signal per location specified by the await must be in the phase, and no other write to the same location is in the phase.

signal (LOC, VALUE) :	LOC = 1
await (LOC1, LOC2, ..., LOCn;	
PRED1, PRED2, ..., PREDn) :	while (!PRED1 || !PRED2 || ... || !PREDn) {;}
/ PREDi depends only on the value of LOCi */*	

Figure 7.7. Signal-await constructs.

The above constraints are reasonable to expect from some producer-consumer scenarios and should be simple to verify; the model simply requires that whenever these constraints are met, the programmer explicitly indicate it to the system by using the special constructs. Simply using the constructs provides enough information to the system that the reads of the awaits can be executed in parallel without violating sequential consistency.

The rest of this section investigates other possible optimizations with the above constructs. For optimizations involving signal writes, we add two other reasonable constraints to the three constraints imposed above:

(4) In any phase, a location specified by a signal or await can be accessed only by other signals or awaits.

(5) If a phase has a signal, then the phase must also have an await specifying the signal location.

Other optimizations.

The following first derives relevant critical path information for the signal and await constructs, and uses it to show that two non-conflicting, program ordered operations from these constructs can be executed out-of-order (and in parallel) in all cases except for an await read followed by a signal write. We also show that all signal writes can be executed non-atomically.

We can derive the following information about the signal and await constructs of the new model.

(i) An await read is not a sender. (Follows directly from the earlier analysis.)

(ii) An await read does not receive for another await read.

> The earlier analysis proved the above for await reads from the same loop; it applies to await reads from different loops as well.

(iii) A signal write cannot be a receiver.

> The constraints above imply that a signal write can conflict only with an await read in a phase. Since an await read cannot be a sender, a signal write cannot be a receiver.

(iv) No ordering path beginning with a signal write is critical.

> An ordering path in a given phase that begins with a signal write must end in an await read (because a signal write cannot conflict with another write in the same phase). The earlier analysis showed that since the await read is from a self-ordered loop, and since the phase begins with the await locations with non-exit values, the path from a signal write to an await read is not critical.

(v) No operation sends a signal write.

> For an operation to send a signal write, the signal write must either begin the critical path or must be a receiver on the critical path. From (iii) and (iv) above, neither is possible.

We next investigate the optimizations the above information allows. First consider the optimization of out-of-order (or parallel) execution of two non-conflicting, program ordered operations, *X* and *Y*, from signal and await constructs. Assuming $X \xrightarrow{\text{po}} Y$, the optimization is safe if *Y* is not a sender for *X* and if *X* is not a receiver for *Y*. From (i) and (ii), it follows that two non-conflicting await reads (from any await loops) can be reordered. From (i) and (iii), it follows that a signal write followed by a non-conflicting await read can be reordered. From (iii) and (v), it follows that two non-conflicting signal writes can be reordered. Thus, all combinations of non-conflicting operations from signal and await constructs, except an await read followed by a signal write, can be reordered.

Next consider the optimization of non-atomic signal writes (as described for proposition 7.13). From proposition 7.13, a write needs to be executed atomically only if it is either a receiver or if it begins a critical path that ends in a read. From (iii) and (iv), neither is possible, and so a signal write can be executed non-atomically.

7.3.2.2. Barriers

This section examines overlapping the latency of barrier operations with operations that precede and follow the barrier. The resulting model is similar to the notion of fuzzy barriers proposed by Gupta [Gup89]; we discuss the relationship with Gupta's work at the end of the section.

As before, assume programs that are divided into phases where adjacent phases are separated by a barrier and locations accessed by barrier operations are not accessed by non-barrier operations. Also assume that for two conflicting operations from different phases and from different processors, the critical path chosen consists of only barrier operations (other than the operations being ordered). For conflicting operations from the same processor, a single program order arc between the operations is the chosen critical path.

If in some phase, an operation *O* accesses a location that is not accessed in the immediately next phase, then intuitively, the operations from the barrier immediately following that phase can be overlapped with the operation *O*. The following explains how this notion (and an analogous notion for barriers preceding an operation) can be formalized and gives the corresponding SCNF memory models.

A non-barrier operation *O* can be overlapped with a following barrier operation *B* if $O \xrightarrow{\text{po}} B$ is not on a critical path. With the above assumptions, this arc can be on a critical path only if the path begins with *O* and ends with an operation O' that conflicts with O and is from a different processor than O. If O and O' are not in adjacent phases of the program, then there are more than one ordering paths between O and O' that contain only barrier operations; only one of these paths needs to be considered critical and executed safely. Call the barrier selected for the critical path from O to O' as a sender barrier for O and a receiver barrier for O'. Then if $O \xrightarrow{\text{po}} B$ (where B is a barrier operation), O and B can be executed in parallel as long as B is not from (or follows) a sender barrier for O. Similarly if $B \xrightarrow{\text{po}} O$, then O and B can be executed in parallel as long as B is not from (or precedes) a receiver barrier for O.

To exploit the above observation, a memory model can provide a mechanism to associate with every operation the sender and receiver barriers of that operation. A conservative approach would be to provide a mechanism to only distinguish between the case when the sender barrier is the immediately following barrier from the case when it is not, and to distinguish between the case when the receiver barrier is the immediately preceding barrier from the case when it not. Assume the default to be the case when the sender/receiver barrier is the immediately following/receiving barrier. A sufficient requirement of the programmer for such a model is that if an operation O is distinguished as having the non-default sender barrier, then the following phase in any sequentially consistent execution should not contain an operation that conflicts with O and is by a processor different from that of O. Similarly, if an operation O is distinguished as having the non-default receiver barrier, then the preceding phase in any sequentially consistent execution should not contain an operation that conflicts with O and is by a processor different from that of O. Communicating and exploiting the information for the above model is much simpler than associating general senders and receivers for general programs as discussed in Section 7.3.1.1. For example, operations can be distinguished using annotations as discussed for earlier models, and the information for the sender barriers can be exploited in hardware by using two counters to indicate the outstanding operations each of the next two barriers needs to wait for.

The model described above is related to the notion of fuzzy barriers proposed by Gupta [Gup89]. The work assumes a barrier implementation using a hardware broadcast mechanism without involving memory. A processor can execute a barrier in parallel with its other memory operations. For each barrier in a process, a region of contiguous instructions before and after the barrier is identified that could potentially be overlapped with the barrier. This region is called the barrier region; the compiler can reorder operations to increase the size of this region. The hardware execution of the barrier is overlapped with the barrier region (operations after the barrier region are guaranteed to begin only after operations before the barrier region complete). Thus, the key intuition behind the fuzzy barrier and our model above is similar. However, Gupta assumes that the compiler will determine the barrier and non-barrier regions of a program. Since compilers can be conservative, we expect to get information from the programmer to determine which operations can be overlapped with a barrier. Since we expect programmers to provide information only about sequentially consistent executions, we need a formal proof that this information is sufficient to guarantee sequential consistency in the presence of the above optimizations. Specifically, unless the system obeyed the control requirement (discussed in detail in Section 7.5), the optimizations discussed above with information from only sequentially consistent executions would not be safe and anomalies similar to figure 5.2 in Section 5.2 could occur. Thus, our work shows that the key intuition used by Gupta is applicable to non-sequentially consistent systems even without aggressive compiler technology, and shows how the necessary information to exploit this intuition can be obtained from the programmer and exploited without hardware support for a barrier.

The work by Gibbons and Merritt [GiM92] discussed in Section 7.3.1.1 is related to the model of this section as well since it allows associating a sender barrier with all operations it sends. Our work additionally allows associating the receiver barrier with an operation, and explicitly identifies barriers as constructs that are relatively easy to reason with for associating senders and receivers.

7.3.2.3. Locks and Unlocks

We investigate a typical use of lock locations for implementing critical sections. Typically (in any sequentially consistent execution), such lock locations are accessed only through special lock and unlock constructs where the lock construct is a self-ordered loop whose only shared-memory operation in the final iteration is a read-modify-write that ‘‘acquires’’ the lock, and the unlock construct is a write that ‘‘releases’’ the lock. Further, programmers ensure that all processes that access the lock location obey the following protocol: a process executes an unlock for a location only if it was the last process to successfully lock the location and it has not already executed another unlock to the same location since that lock. Finally, every execution of a lock construct eventually acquires the lock. We call lock and unlock operations that obey the above protocol as critical section locks and unlocks, or CS_locks and CS_unlocks for short.

The following considers optimizations for CS_locks and CS_unlocks; we first consider optimizations involving only CS_lock/CS_unlock operations and then consider optimizations involving interactions between CS_lock/CS_unlock operations and other operations. Uses of locks and unlocks that do not obey the CS_lock and CS_unlock restrictions cannot, in general, safely use the following optimizations.

Optimizations involving only CS_lock/CS_unlock operations: Analysis.

Essential CS_lock reads and CS_unlock writes obey the conditions for a loop operation of PLpc and an essential CS_lock write obeys the conditions for a data operation (as discussed in Chapter 6). Thus, a CS_unlock write followed by a CS_lock read can be executed in parallel, and CS_lock and CS_unlock writes can be executed non-atomically. The following investigates if other optimizations are possible by deriving critical path information for CS_locks and CS_unlocks. The information derived and the analysis used is similar to that for the signal and await constructs. It shows that along with optimizations of PLpc, two non-conflicting program ordered CS_unlock writes can be reordered or overlapped, and a CS_unlock write does not require acknowledgements. (The analysis below also provides a simple proof for why the optimizations of the PLpc models are safe for CS_locks and CS_unlocks.)

The protocol for a CS_lock and CS_unlock mentioned above leads to the following useful observation. Let *X* and *Y* be conflicting operations from CS_lock or CS_unlock constructs where *X* executes before *Y* (i.e., $X \xrightarrow{co} Y$). Then either *X* is a CS_unlock write and *Y* is a CS_lock read, or *X* and *Y* are separated by alternating CS_unlock writes and CS_lock reads. Both cases imply that there is a path from *X* to *Y* in the program/conflict graph where the only $\xrightarrow{co}$ arcs are from a CS_unlock write to a CS_lock read. This implies that if a $\xrightarrow{co}$ arc consisting only of CS_lock and CS_unlock operations is on an ordering path, then the arc can always be replaced with another path where only CS_unlock writes are senders and CS_lock reads are receivers. Thus, if when choosing a critical path, we always choose an ordering path where such a replacement has been made, it follows that the only sender operations from CS_lock/CS_unlock constructs are CS_unlock writes and the only receiver operations from such constructs are CS_lock reads. Assuming that critical paths are chosen based on the above criterion, we can derive the following information.

(1) CS_lock read cannot be a sender. (Follows from assumption in previous paragraph.)

(2) CS_unlock write cannot be a receiver. (Follows from assumption in previous paragraph.)

(3) CS_lock write cannot be a sender or receiver. (Follows from assumption in previous paragraph.)

(4) No operation need send a CS_unlock.

> Consider a CS_unlock *U*. *U* cannot be a receiver. Therefore, if *U* is sent by some operation *O*, then the corresponding critical path must begin with *U*. The path can end at either a CS_lock or CS_unlock operation (say) *UL*. *UL* and *U* must be consecutive conflicting operations for the following reason.
>
> Suppose for a contradiction that *UL* and *U* are not consecutive conflicting operations. The only critical paths between operations that are not consecutive conflicting operations are paths to self-ordered reads. Therefore, assume that *UL* is a read. Therefore, *UL* must be a CS_lock read. The only path to a self-ordered read that needs to be critical and is not from a consecutive conflicting operation is a path from a write that does not write an exit value for the read. However, *U* writes an exit value for *UL*, a contradiction.
>
> Therefore, *U* and *UL* must be consecutive conflicting operations. But then *UL* cannot be a CS_unlock because a CS_lock write must come between two CS_unlocks to the same location. Therefore, *UL* is a CS_lock write or CS_lock read. If *UL* is a CS_lock write, then the path $U \xrightarrow{co} R \xrightarrow{po} UL$ is critical where *R* is the read of *UL*'s lock. *U* is not sent by any operation on this path. So suppose *UL* is a CS_lock read. Then since *UL* is from a self-ordered loop, a path to *UL* needs to be critical only if there is either a path from *U* or from the last previously executed conflicting CS_lock write to *UL* that ends in a program order arc. Any such path can be chosen as critical. The last conflicting CS_lock write *W* before *UL* must be by *U*'s processor. Therefore, if a critical path is required, then there exists an ordering path from *W* to *UL*

that ends in a program order arc. We can require this path to be critical; therefore, no operation need send *U*.

Thus, if an ordering path from an essential CS_lock write to the next conflicting essential CS_lock read and that ends in a program order arc is considered critical (when such a path exists), and a path of the type CS_unlock $\xrightarrow{co}$ CS_lock-read $\xrightarrow{po}$ CS_lock-write is considered critical, then no operation need send a CS_unlock.

(5) No critical path begins with a CS_unlock write operation on a conflict order arc and ends on a read.

Suppose there is a path of the kind stated above beginning with a CS_unlock *U* and an arc of the type *U* $\xrightarrow{co}$ *O*. Let the path end on a read *L* (*L* must be a CS_lock read). For reasons discussed in the previous analysis, *U* and *L* must be consecutive conflicting operations. Therefore, *O* cannot be a write. Therefore, *O* must be a CS_lock read. However, then *O* is a read-modify-write and this write must be between *O* and *L*. Thus, again *U* and *L* are not consecutive conflicting operations and so the path is not critical.

The model motivated by the above analysis is discussed below.

Optimizations involving only CS_lock/CS_unlock operations: The Model.

The above analysis motivates a model that recognizes special lock/unlock constructs called CS_lock and CS_unlock respectively. Programmers are allowed to use these constructs for locking and unlocking only if they obey the constraints mentioned in the beginning of the section for CS_locks and CS_unlocks; i.e., in any sequentially consistent execution of the program, a process executes a CS_unlock only if it was the last process to successfully acquire the lock location and it has not already issued a CS_unlock to the same location since that lock, locations accessed by a CS_lock or CS_unlock are accessed only by other CS_lock or CS_unlock operations, and a CS_lock operation always eventually succeeds in acquiring the lock location.

The above constraints describe the conventional protocol for implementing critical sections and are usually obeyed; the model simply requires the programmer to make this explicit by using a special construct. Providing a special construct allows programmers who might want to use other protocols to continue to do so by using the usual locking constructs provided by the system. It also allows for portability of dusty deck programs for which it may be difficult to verify whether the usual locks and unlocks obey the above protocol.

Using the above constructs (as specified) provides enough information to the system to execute CS_unlock and CS_lock writes non-atomically (from (2), (3), and (5) and as described for proposition 7.13), to reorder or overlap two program ordered non-conflicting CS_unlock writes (from (2), (4)), to reorder or overlap a CS_unlock followed by a non-conflicting CS_lock (from (1), (2)), and to not require acknowledgements on a CS_unlock write (from (4) and since CS_unlock writes can be non-atomic).

We next investigate optimizing interactions between CS_lock/CS_unlock operations and other operations.

Other optimizations.

The data-race-free models showed that data (or non-communicator or non-race) operations following an unlock (or preceding a lock) can execute in parallel with the unlock (or lock). As discussed earlier, for increased parallelism, the entry consistency model requires that a data location be accessed only between an acquire and release of a lock that has been declared as protecting that data [BeZ91, BZS92]. Using our terminology, an unlock in this model only sends the operations following the preceding lock to the same location, and a lock only receives for operations preceding the following unlock to the same location. Providing only the above types of locks and unlocks is restrictive for programmers since not all data can be accessed in the above manner, as illustrated by figure 7.8. The figure shows processors repeatedly getting a task from a task queue, processing the task, and possibly enqueuing new tasks for others to process. Assume that in any sequentially consistent execution, procedures that process or generate tasks do not execute any operation that may conflict with any other processor's concurrent accesses. Thus, the only necessary synchronization is to access the queues. Since a processor may add and remove tasks to and from different queues, a data operation may never be between a lock and unlock to the same location and the entry consistency mechanism is insufficient.

To alleviate the above problem, the Midway system allows different data locations to be accessed using different consistency protocols [BZS92]. Thus, the locations accessed by the task processing and task generating procedures could be declared as release consistent; consequently, all following unlocks would send operations to

```
Master_Process_Code

while (More_Tasks) {
    generate initial task to be done
    Enqueue_Task(Task,Queue_i)
}
```

```
Slave_Process_Code

while (!All_Tasks_Done) {
    Task = Deque_Task(Queue_j)
    process the task
    if (New_Tasks_Generated)
        Enqueue_Task(New_Task,Queue_k)
}
```

```
procedure Deque_Task(Queue_i)
{
  lock(Lock_for_Queue_i)
  remove task from queue_i and set pointers
  unlock(Lock_for_Queue_i)
  return(Task)
}
```

```
procedure Enqueue_Task(Task,Queue_i)
{
  lock(Lock_for_Queue_i)
  put task in queue_i and set pointers
  unlock(Lock_for_Queue_i)
}
```

Figure 7.8. Task Queues.

such locations. This implies that the unlock in a dequeue procedure would be a sender for all the operations in the preceding task processing procedures, and would typically have to wait for those operations to complete. Intuitively, this serialization is not necessary. How can we extend the key idea of entry consistency without the above limitation? Gibbons and Merritt proposed a general scheme to associate releases with operations they release [GiM92]; i.e., an unlock could be associated with the operations it sends. However, such a general mechanism is difficult to implement and use, and not necessary for figure 7.8. Can we identify easier-to-use mechanisms?

Figure 7.8 uses two types of easily distinguishable unlocks: unlocks that send all preceding operations and unlocks that send only the operations that follow the last preceding lock. Thus, the unlock of the enqueue procedure potentially sends all preceding operations, but the unlock of the dequeue only needs to send the operations following the preceding lock (to ensure that the queue is accessed consistently). Analogous observations hold for the locks in figure 7.8. Further, note that all the lock and unlock accesses in the figure are CS_locks and CS_unlocks.

The above motivates a model that distinguishes between two types of CS_lock and CS_unlock constructs. The first is represented as a regular CS_lock or CS_unlock construct that is not optimized any more than in the previous section. The second version, represented by partial_CS_lock and partial_CS_unlock, is optimized similar to entry consistency. Informally, users of the partial constructs must use those constructs only to order data accessed in the manner described for entry consistency; all other data operations must be ordered by communicators other than the partial constructs. More formally, the above model requires a location accessed by an operation distinguished as a non-communicator (or non-race or data) to obey one of two conditions in any sequentially consistent execution. The first condition is that the location is associated with some lock (say *L*), an operation to the location is always preceded by any CS_lock construct (partial or otherwise) accessing the lock *L* and is always followed by any CS_unlock construct accessing the lock *L*. The alternative condition is that any two conflicting operations that access the location must be ordered by an ordering path consisting of communicators other than the partial_CS_lock and partial_CS_unlock constructs.

The above model ensures that the only non-communicators that a partial_CS_unlock sends are the non-communicators following the last preceding CS_lock to the same location. Any other preceding non-communicators can be executed in parallel with the unlock. For example, in figure 7.8, if we assume that all operations within a lock and unlock in the enqueue and dequeue procedures are always accessed within the same lock and unlock variable, then the unlock in the dequeue procedure can be made partial. This allows the non-communicator operations in an enqueue or task processing procedure to be overlapped with the unlock of the following dequeue procedure. We already know that two CS_unlocks can be overlapped. Thus, it follows that once the preceding enqueue procedure (from the previous iteration) gets its lock, the unlock of a dequeue procedure can be executed even if the enqueue operations or the task processing operations of the previous task are not yet done. While this does not directly speed up the dequeuing process any more than the previous models, it allows the queue locks to be released faster and can potentially aid overall performance by decreasing contention on the queues.

Analogous optimizations are possible with a partial_CS_lock in general, but cannot be exploited in the example in figure 7.8 because a dequeue lock cannot in general be executed in parallel with a preceding enqueue lock (in the absence of further constraints).

The above model combines the notions of entry consistency which essentially allows only partial lock/unlock constructs, and the data-race-free models which essentially allow only the ‘‘regular’’ lock/unlock constructs. However, this combination is different from the model provided by Midway that combines entry and release consistency [BZS92] since Midway requires the unlock of the dequeue to send data operations from the preceding task processing procedure. Further, since we have shown that CS_unlocks can be executed in parallel, it follows that the dequeue unlock need not wait for preceding data operations from the enqueue procedure either; Midway does not formally guarantee sequential consistency with such an optimization. Also note that although from the programmer’s perspective, the above model can be viewed as a special case of the relaxed release consistency model in [GiM92], the above optimization is not allowed by the systems considered in [GiM92] since they do not reorder unlocks; further, the special case we cite is easier to use than the general mechanism of [GiM92].

7.3.2.4. Constructs to Decrease Lock Contention

Consider figure 7.9(a). It shows a processor that accesses some shared variables protected by a lock. Before the processor can make the accesses, however, it must ensure that some predicate based on the values of certain shared-memory locations is true (e.g., it may need to ensure that a queue to be accessed is not empty). If the predicate is not true, the processor must try again. If the predicate is true the processor performs some work which may involve updating some of the locations read to test the predicate. Assume that any other accesses in the rest of the program to the locations accessed within the critical section are also protected by the same lock; i.e., the accesses are non-communicators.

To decrease lock contention, it may be worthwhile to wait for the predicate to become true outside of the critical section, as shown in figure 7.9(b). Call such a loop that tests the predicate as a predicate loop. A predicate loop implies that the write accesses within the critical section body that conflict with the reads of the predicate loop race with those reads. Without any further analysis, these writes must be declared as communicators that might send or receive for other communicators. This implies that these writes now have to wait for preceding such writes in the critical section and must be done non-atomically. Intuitively, just the introduction of a performance-enhancing loop should not impose such a restriction. Below, we show how we can reason about the loop and define an SCNF model that eliminates this restriction.

Analysis.

We assume the locks and unlocks in figure 7.9 are CS_locks and CS_unlocks (as defined in the previous section). Also similar to the previous section, we assume that when choosing an ordering path to be critical, we choose a path with minimum number of $\xrightarrow{co}$ arcs that are not from a CS_unlock write to a CS_lock read. For simplicity, the analysis below first assumes the predicate in figure 7.9 involves only one shared-memory read.

Consider a read *R* from the predicate loop and a write *W* from the critical section to the same location as *R*. To eliminate the restrictions described above, we need to analyze when and if *W* can send to or receive from *R*.

```
while (!done) {
    Lock(L)
    if (A == ... && B = ...) {
        ...
        B = ...
        A = ...
        ...
        done = true
    }
    Unlock(L)
}
```

(a)

```
while (!done) {
    while (A != ... || B != ...) {;}
    Lock(L)
    if (A == ... && B = ...) {
        ...
        B = ...
        A = ...
        ...
        done = true
    }
    Unlock(L)
}
```

(b)

Figure 7.9. Loop for decreasing lock contention.

First consider when *W* could send to *R*; i.e., $W \xrightarrow{co} R$ is on a critical path from (say) operation O_1 to operation O_2 in some sequentially consistent execution. Assume sequentially consistent executions without unessential operations. *W* is followed by an unlock write *U*, and *R* is immediately followed by a successful lock read *L* that executes after *U*. If *R* receives from *W* for an operation that follows *L*, then $W \xrightarrow{co} R$ on the critical path can be replaced with $U \xrightarrow{co} L$, a contradiction to our choice of critical paths. Thus, $W \xrightarrow{co} R$ cannot be on a critical path where *R* receives for an operation that follows *L*. So for *W* to send to *R*, *R* must receive for *L*. From the previous section, we know that *L* cannot be a sender. Again, from the previous section, a critical path that ends at a CS_lock read must be from the write of the last CS_lock. Thus, if *W* sends to *R*, then the corresponding critical path must be CS_lock *write* $\xrightarrow{po} W \xrightarrow{co} R \xrightarrow{po}$ CS_lock *read*, where the CS_lock write is from the last lock of that location preceding *W* and CS_lock read is the first lock read following *R*.

Next consider when *W* could receive from *R*. Again, let *L* be the lock access that successfully acquires the lock and immediately follows *R*. If *W* executes after *L*, then there is an ordering path from *R* to *W* consisting of alternating CS_unlock writes and CS_lock reads, contradicting our choice of critical paths. Therefore, *W* must execute before *L*. Suppose *R* is not from the first iteration of the outer-most loop in figure 7.9(b), then the unlock from the previous iteration of the outer-most loop and the lock preceding *W* can replace $R \xrightarrow{co} W$ on the critical path, again a contradiction. So assume that *R* is from the first iteration of the outer-most loop. The following simple extension to our framework shows that a path with *R* as a sender need not be in the critical set.

Consider a program *Prog* that uses the construct shown in figure 7.9(b). Consider a program *Prog′* that is the same as *Prog* except that the read *R* from the predicate loop in the first iteration of the outer-most loop is not program ordered after anything (recall that program order is partial per process). Then *R* cannot be a sender on a critical path of an execution of *Prog′* because *R* has nothing to send. Consider a system that safely executes all critical paths of the program *Prog* except those that have *R* as a sender (the system also obeys the corresponding control condition). Then such a system safely executes all the critical paths of program *Prog′* and so the result of an execution of program *Prog* on such a system will be the result of a sequentially consistent execution of the program *Prog′*. We show below that the result of a sequentially consistent execution of *Prog′* is the same as that of a sequentially consistent execution of *Prog*; therefore, the above mentioned system appears sequentially consistent to *Prog* and so ordering paths where *R* is a sender are not critical.

Consider a sequentially consistent execution E of program *Prog′* without any unessential operations. We divide our reasoning into two cases as follows.

Case (1): The first iteration of all outermost loops of the constructs of the type in figure 7.9(b) are not the last iterations of the loops in E.

If the first iteration is not the last, then no operation from outside the iteration can see the effect of the iteration. Thus, the first iteration of all the loops can be deleted from the execution leaving behind another sequentially consistent execution of *Prog′* with the same result as E. This execution is also a sequentially consistent execution of *Prog*, proving the required result.

Case (2): The first iteration of some outermost loop of the constructs of the type in figure 7.9 is the last iteration of the loop in E.

In this case, again delete the first iteration of all the loops where the iteration is not the last one. Let this execution be E'. Now consider a loop where the first iteration is the last. Let R be the read from the predicate loop and let L be the successful lock access immediately following R. Then the last write W in E that is before L and to the same location as R (if any) must make R terminate the predicate loop. Consider an execution order that is the same as that of E' except that R is placed immediately before L and returns the value of W (or the initial value if W does not exist). This execution order represents an execution with the same result as that of E and is also an execution order of a sequentially consistent execution of program *Prog*, proving our proposition.

Thus it follows that a write W in the critical section of figure 7.9 (b) is never a receiver, and is a sender only for a path of the type CS_lock-*write* $\xrightarrow{\text{po}}$ W $\xrightarrow{\text{co}}$ R $\xrightarrow{\text{po}}$ CS_lock-*read*, where the lock write is from the last lock of its location preceding W. It follows that W can be executed non-atomically. Further, since it only sends the lock write, it need not wait for any operations other than the lock write in the critical section. Therefore, the restrictions described earlier are not required.

The above analysis has assumed a predicate whose value depends on a single read. The analysis easily extends to predicates with multiple reads by reasoning about a program where the reads in the predicate are not program ordered with respect to each other, and the reads in the predicate in the first iteration of the outer-most loop are not program ordered after any other operations.

The Model.

The above observations motivate a model that recognizes a special synchronization construct called Lock_With_Predicate as shown in figure 7.10(a). The construct consists of a lock variable, a predicate, and a body. The predicate is a function of the values of zero or more shared-memory locations. The semantics of the construct are as shown in figure 7.10(b). The model requires users of the construct to obey the following constraints for every sequentially consistent execution: (1) locations accessed by the construct are accessed only by other Lock_With_Predicate constructs that specify the same lock variable (the predicates and bodies may be different), (2) a process that executes the construct eventually finds the predicate to be true, and (3) the predicate code does not write any shared-memory location and the values of any registers or private memory written by an iteration of the predicate code can only be read by instruction instances of that iteration.

With the above model, a Lock_With_Predicate construct provides enough information to the system such that the accesses (other than the lock and unlock) within the construct can be treated like non-communicators except that an ordering path such as CS_lock-*write* $\xrightarrow{\text{po}}$ *write* $\xrightarrow{\text{co}}$ *read* $\xrightarrow{\text{po}}$ CS_lock-*read* made of accesses from such constructs must be safely executed. (It is sufficient if the above path is safely executed only when the first read on the above path is from a predicate loop. Thus, such a path would be automatically safe with hardware that does not speculatively execute a write or read-modify-write following an unbounded loop until it is known that the loop will terminate.) Specifically, the hardware can execute a write from the above construct non-atomically and can execute all operations other than the lock and unlock operations from the above construct in parallel.

```
Lock_With_Predicate {
        { Lock_Variable }
        { Predicate }
        { Body }
}
```

(a)

```
while (!Predicate) {;}
Lock(Lock_Variable)
if (Predicate) {

     Body

}
Unlock(Lock_Variable)
```

(b)

Figure 7.10. Predicate-loop construct.

7.4. Characterization of the Design Space of SCNF Memory Models

From the results of the previous sections, we deduce that the key characteristic common to all SCNF memory models is that they can identify certain ordering paths as special and promise to execute those safely. If the programmer ensures that these special paths include a critical set of paths for every sequentially consistent execution of the program, then the model promises sequential consistency to the programmer. We call the special ordering paths that the model executes safely as the *valid paths* of the model, and define a generic model in terms of valid paths as follows.

> **Definition 7.14:** *A Generic SCNF Memory Model:* An SCNF memory model specifies a characteristic set of ordering paths called the *valid paths* of the model. A program is a *valid program* for an SCNF memory model iff for all sequentially consistent executions of the program, a critical set of ordering paths for the execution are valid paths of the model. A system obeys an SCNF memory model iff it appears sequentially consistent to all programs that are valid programs for the model.

The constraints on the programmer of a generic SCNF model are apparent from the above definition:

> **Definition 7.15:** *Constraints on the Programmer of a Generic SCNF Memory Model:* Programmers of an SCNF model must write programs that are valid programs for the model.

System constraints are less obvious from the above definition, but follow from the earlier discussion in Sections 7.1 and 7.2. The system-centric specifications of this and the next section are generalizations of similar specifications developed for the data-race-free-1 [AdH92] and PLpc memory models [AGG93]. The generalization makes explicit the relation between the model, the programmer constraints, and the system constraints. The methodology used to describe the specifications is based on the work in [AdH92] and [GAG93]. Section 7.6 further explains the relationship between the specifications of this chapter and those in [AdH92, AGG93, GAG93].

In the specification below (and in the rest of the chapter), we use the term ‘‘synchronization loop’’ only for those loops that are exploited as synchronization loop by the model being considered; i.e., some programs valid for the model are no longer valid if operations from the unsuccessful iterations of the loop are not ignored. Similarly, we use the term ‘‘self-ordered read’’ only for a read R that is exploited as self-ordered by the relevant model; i.e., in some sequentially consistent execution of a valid program for the model, part (3) of the definition of critical paths is used to select a critical path to R to be the valid path to R. Similarly, we use the term ‘‘self-ordered loop’’ only for those loops that have a self-ordered read in the above sense. Recall that we assume that all loops exploited as self-ordered loops are also exploited as synchronization loops; therefore, any system constraints applicable to the latter are also applicable to the former.

Condition 7.16: *High-Level System-Centric Specification of Generic SCNF Model:*

Valid Path: If there is a valid path of the model from X to Y and if either X and Y are from the same processor, or if X is a write, or if X is a read that does not return the value of its own processor's write, then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i.

Control: For any valid program for the model, the following should be true.

(a) *Critical set:* The valid paths of the execution form a critical set of the execution.

(b) *Finite speculation:* The number of instruction instances program ordered before any instruction instance is finite.

(c) *Write termination:* For any write in the execution, all sub-operations of the write are in the execution.

(d) *Loop Coherence:* If $W_1 \xrightarrow{\text{co}} W_2$ and one of W_1 or W_2 is from a synchronization loop, then $W_1(i) \xrightarrow{\text{xo}} W_2(i)$ for all i.

Proof: The proof follows directly from Condition 7.12. The valid path requirement and the critical set part of the control requirement ensure that the critical paths of the execution are executed safely. It is safe to ignore valid paths from a read R to a write W where R and W are from different processors and R returns the value of its own processor's write W', because the control requirement ensures a critical path corresponding to W' and W will be valid and so $W'(i) \xrightarrow{\text{xo}} W(i)$ for all i. This implies that $R(i) \xrightarrow{\text{xo}} W(i)$ for all i and so the path from R to W is automatically safe.[28] The remaining parts of the control requirement make explicit the finite speculation, write termination, and loop coherence assumptions of Condition 7.12. □

Although current memory models are not specified directly in terms of valid paths, they can be converted into the form of Definition 7.14. For example, the valid paths for the data-race-free-0 model are the happens-before-0 paths. For a model that only distinguishes between communicators and non-communicators, the valid paths are all ordering paths where the conflict order edges are between operations distinguished as communicators.

Similarly, for many current and future models, the above form may not be the best specification for programmers or system designers to use directly. For example, the easiest-to-use characterization of the model based on signal-await constructs discussed earlier is rather different from the valid path characterization above. Similarly, the data-race-free-0 model is easier to program with when stated as requiring the identification of all race operations as synchronization. Nevertheless, the above characterization in terms of valid paths is valuable for the following three reasons.

First, the above definition of a generic model characterizes and exposes the design space of memory models. Theoretically, any set of ordering paths can be chosen as valid paths and considered as a new memory model. Similarly, any two specifications that imply the same set of valid paths represent the same memory model.

Second, the above definitions directly provide a qualitative assessment of each point in the design space since the valid paths of a model determine its programmability and performance potential as follows.

The programmer must ensure that for every sequentially consistent execution of the program, the critical paths of the execution are valid paths. Therefore, the ease of programming with a model depends on the ease with which programmers can convert critical paths into valid paths.

The generic system-centric specification (Condition 7.16) described above indicates the performance potential of the model. The next section will show that the control requirement is mostly a function of uniprocessor control dependences and the valid paths of the model, and is often already obeyed by systems. Therefore, the key

[28] The relaxation for reads that return the value of their own processor's write is specifically used in Theorem H.1 in Appendix H.

determinant of performance is the valid path requirement, which requires the valid paths of an execution to be executed safely. The minimal requirement for sequential consistency that we have been able to ascertain in Section 7.2 is that the system must execute critical paths safely. Thus, a measure of the performance potential of a memory model is the precision with which the valid paths of an execution can capture the critical paths of an execution. Specifically, providing a mechanism to allow programmers to convert exactly the critical paths to valid paths is usually complex and expensive. Therefore, a model will usually provide simpler mechanisms by which a programmer can convert a superset of the critical paths into valid paths; i.e., converting a critical path into a valid path may implicitly convert other ordering paths into valid paths which will also be executed conservatively. Thus, the performance potential of the model depends on how much of the critical path information the model allows the programmer to convey to the system. For example, a model that only distinguishes between communicators and non-communicators assumes that a write distinguished as a communicator is a sender for all preceding operations and so uses less precise information (and has lower performance potential) than a model that can associate a sender with operations that it sends. The valid paths of a model make apparent how much critical path information the model allows the programmer to convey, and so directly indicate the performance potential of the model.

Finally, the third advantage of the characterization, as we have already seen in the previous section, is that it facilitates converting well-known information about programs into useful memory models that can exploit that information. Conversely, it allows us to determine for important optimizations or system implementations, the corresponding information from the programmer that would ensure sequential consistency. Program information can be converted into optimizations by converting it into the paths that it ensures will not be critical; the system need not consider those paths as valid and so can optimize those paths. For a set of system constraints or optimizations, the necessary information can be determined by determining the ordering paths that the system executes safely and using those paths for which the control requirement is also obeyed as the valid paths.

7.5. Implementing An SCNF Memory Model

The previous section gave a generic high-level system-centric specification for the generic memory model. This section describes lower-level specifications and implementations. Since the control requirement is not very intuitive, Section 7.5.1 first gives the motivation for the various aspects of the control requirement. Section 7.5.2 then gives low-level system-centric specifications and corresponding hardware implementations. Section 7.5.3 discusses corresponding compiler implementations.

7.5.1. Motivation for the Control Requirement

The following intuitively motivates the various parts of the control requirement of Condition 7.16.

Motivation for the critical set part of the control requirement.

The critical set part of the control requirement states that for an execution of a valid program, the valid paths of the execution should form a critical set of the execution. This part is needed to avoid situations where the programmer ensures that a critical set of every sequentially consistent execution of the program consists of valid paths, the system executes all valid paths safely, and yet the resulting execution is not sequentially consistent because some paths from every critical set of the execution are not valid. This is possible with a valid program since a valid program guarantees a critical set will constitute valid paths only for sequentially consistent executions. Figure 7.11 illustrates the need for the requirement.

The example in figure 7.11(a) was used to motivate the control requirement for data-race-free-0 in Chapter 4; the same reasons are applicable to all other SCNF models. Specifically, in any sequentially consistent execution, the reads of *X* and *Y* should return the value 0, and the writes should not be executed. Thus, there are no ordering paths in any sequentially consistent execution and the program is valid for any model. Without the control requirement, it is possible to have an execution where both processors return the value 1 for their reads and execute their writes (the corresponding set of instructions and operations obey the uniprocessor correctness condition and allow an execution order with the required properties). Such an execution is not sequentially consistent and it has ordering paths which are not executed safely. For example, the path Read,X $\xrightarrow{po}$ Write,Y $\xrightarrow{co}$ Read,Y $\xrightarrow{po}$ Write,X is not executed safely since the read of X returns the value written by the write of X. This path does not have to be a valid path for the program to be valid and so violating this path does not violate the valid

```
Initially X = Y = 0

P1              P2
if (X == 1)     if (Y == 1)
   Y = 1           X = 1
```

(a)

```
Initially X = Y = flag = 0

P1                      P2
if (X == 0) {Y = 1}     while (flag != 1) {;}
flag = 1                if (Y == 0) {X = 1;}
```

(b)

Figure 7.11. Motivation for critical set part of control requirement.

path requirement. The critical set part of the control requirement prohibits the above path from occurring in the execution by requiring that the critical paths of the execution be the valid paths of the model. Note that since most processors do not execute writes speculatively, processors P_1 or P_2 will usually not execute their write until the read returns a value and it is known that the write needs to be executed. This precludes the execution described above. Thus, simply observing uniprocessor control dependences prohibits an anomaly of the above type.

Figure 7.11(b) illustrates an example (from [AdH92]) where observing simple uniprocessor control dependences may not suffice. In any sequentially consistent execution of the program, P_2's read on *Y* would always return the value 1, and therefore P_2 would never issue the write on *X*. Thus, the only ordering path in any sequentially consistent execution is Write,*Y* $\xrightarrow{po}$ Write,*flag* $\xrightarrow{co}$ Read,*flag* $\xrightarrow{po}$ Read,*Y*. Assume a memory model where only the above path is valid. In the absence of the control requirement, an aggressive implementation [FrS92] could allow P_1 to write *flag* before its read of *X* returned a value. This could result in the following sequence of events which makes P_2's read on *Y* return 0, and violates sequential consistency without violating the valid path requirement: (a) P_1 writes *flag*, (b) P_2's read on *flag* returns 1, (c) P_2's read on *Y* returns 0, (d) P_2 executes its write on *X*, (e) P_1's read on *X* returns 1, (f) P_1 does not issue its write on *flag*. This execution is not sequentially consistent and it has an ordering path which is not executed safely: Read,*X* $\xrightarrow{po}$ Write,*flag* $\xrightarrow{co}$ Read,*flag* $\xrightarrow{po}$ Write,*X*. This path does not have to be a valid path for the program to be valid and so need not be executed safely to obey the valid path requirement. For example, in data-race-free-1, assuming operations on *Y* and *flag* are unpairable synchronization operations and operations on *X* are data operations, the given program is valid (i.e., data-race-free-1) but the path above is not a valid path. The critical set part of the control requirement prohibits the above path from occurring. Again, note that the anomaly described above occurs because processor P_1 executed its write on *flag* too early; however, note that the write of *flag* is not (uniprocessor) control dependent on any operation of its processor, and so this case is different from that in figure 7.11(a).

In general, in the absence of the critical set part of the control requirement, anomalies of the type in figure 7.11(b) may occur because the system is free to execute a logically future operation that results in some non-sequentially consistent behavior which destroys a future valid path whose safe execution was to have prevented this very behavior. We will see that the low-level specification of the control requirement will break the above cycle in causality by ensuring that no logically future operation can affect valid paths from its logical past. Anomalies of the type in figure 7.11(a) may occur because a logically future operation that should not be executed occurs and affects the rest of the execution in a way that makes it possible for this operation to be executed. The low-level control requirement will break this cycle in causality by ensuring that no speculative operation whose execution is not yet certain can affect the course of the execution.

Motivation for the finite speculation part of the control requirement.

The finite speculation part of the control requirement states that in an execution of a valid program, the number of instruction instances program-ordered before any instruction instance should be finite. This requirement is met if processors do not execute instructions speculatively since without speculative execution, each instruction instance is preceded by a finite number of instruction instances even in an infinite execution. For processors that do speculative execution, Figure 7.12 illustrates the need for this condition. (Similar examples appear in [AGG93].) In part (a) of the figure, for every sequentially consistent execution, P_1's while loop will be in an infinite loop and so its write of X will not be executed. Therefore, P_2 will never write an error message on the output. Thus, there are no conflicting operations in any sequentially consistent execution and the program is valid for any model. If, however, the finite speculation assumption of the control requirement is not ensured, then it is possible that P_1 executes its write of X. This does not violate the valid path requirement or the critical set part of the control requirement since there are no valid or critical paths in the execution. The finite speculation part of the control requirement prohibits the above execution.

```
                 Initially X = 0                                  Initially X = Y = 0

    P1                 P2                              P1                    P2

    while (true) {;}   if (X == 1)                     while (X != 1) {;}    if (Y == 0)

    X = 1                  write error on output       Y = 1                     X = 1

                    (a)                                                 (b)
```

Figure 7.12. Motivation for finite speculation part of control requirement.

In figure 7.12(b), P_1's while loop terminates in every sequentially consistent execution. However, without the finite speculation requirement, it is possible for processor P_1 to execute the write of Y and for processor P_2 to read the updated value of Y and never execute the write of X, making P_1's loop never terminate. There are no ordering paths in the execution and so only the finite speculation assumption prohibits this anomaly.

Note that the examples discussed above differ in that the first involves a loop that never terminates in a sequentially consistent execution while the second involves a loop that always terminates in a sequentially consistent execution.

Motivation for the write termination part of the control requirement.

```
            Initially X = 0                  Initially X = Y = 0

    P1          P2                           P1          P2

    X = 1;      while (X != 1) {;}           X = 1       Y = 1

                                             Z = 1       Z = 2

                                             ... = Y     ... = X

              (a)                                  (b)
```

Figure 7.13. Motivation for write termination part of the control requirement.

The write termination part of the control requirement states that if an execution of a valid program contains a write, then it should contain all possible sub-operations (in all memory copies) of the write. Figure 7.13 illustrates the need for this condition. For the program in figure 7.13(a), in every sequentially consistent execution, some read of X by processor P_2 will return the value of the write by processor P_1, terminating P_2's while loop. However, if the write termination part of the control requirement is not ensured, then this may not happen. For the program of figure 7.13(b), in any sequentially consistent execution, either P_1's write of Z is ordered before P_2's write of Z by the conflict order, or vice versa. Therefore, in any sequentially consistent execution, either there is an ordering path of the type Write,$X \xrightarrow{po}$ Write,$Z \xrightarrow{co}$ Write,$Z \xrightarrow{po}$ Read, X, or of the type Write,$Y \xrightarrow{po}$ Write,$Z \xrightarrow{co}$ Write,$Z \xrightarrow{po}$ Read, Y. Thus, for the program to be valid, both of the above paths must be recognized as valid, and in any sequentially consistent execution either the read of *X* or the read of *Y* returns the updated value. Consider, however, an execution where P_1's write of *Z* does not happen in P_2's memory copy and P_2's write of *Z* does not happen in P_1's memory copy. Then in this execution, the two writes to *Z* are not related by conflict order and there are no valid paths in the execution. Thus, the reads of *X* and *Y* can both return the initial values, giving a non-sequentially consistent result. The write termination requirement prohibits such an execution.

Motivation for the loop coherence part of the control requirement.

The loop coherence part of the control requirement states that in an execution of a valid program, all sub-operations of a write from a synchronization loop should either appear before or after all the sub-operations of any other write in the execution order. Figure 7.14 motivates this condition through a hypothetical variant of a compare and swap instruction denoted by CAS. The primitive has three arguments and has the following semantics. It reads the value in the location denoted by the first argument. If this value is the same as the second argument, then it updates the location to the value indicated by the third argument and returns the value true. If the value returned is not the same as the second argument, then it writes back the value read, and returns the value false. The loop containing the primitive in the figure qualifies as a synchronization loop. If the loop coherence part of the control requirement is not obeyed, it is possible that P_1's write of *X* to the value 1 happens in P_2's memory copy, P_2's read from the primitive returns the value 1, P_2's write from the primitive writes back the value 1, P_2's write on *X* happens in P_3's memory copy before P_1's write on *X*, P_3 returns the value written by P_2's write, P_3's loop terminates, P_3 returns the value 0 for its read of *Y*. The above execution does not appear sequentially consistent, and is prohibited by the loop coherence requirement.

Initially X = Y = 0

P_1	P_2	P_3
Y = 1		
X = 1	while (CAS(X,5,0)) {;}	while (*X* != 1) {;}
X = 5		... = Y

Figure 7.14. Motivation for loop coherence part of the control requirement.

7.5.2. Low-Level System-Centric Specifications and Hardware Implementations

The following gives low-level system-centric specifications of the valid path and control requirements and describes corresponding hardware implementations. As mentioned earlier, the specifications of this section and the various concepts they use are related to those in [AdH92, AGG93, GAG93]; Section 7.6 discusses this relationship.

Valid path requirement.

The high-level valid path requirement in Condition 7.16 is analogous to the data requirement of the data-race-free-0 model discussed in Chapter 5. Consequently, analogous low-level specifications are possible. The following is a conservative low-level specification that follows from the discussion in Section 7.3.

Condition 7.17: *Low-level system-centric specification of the valid path requirement:* Below, we use $*W$ to indicate a path with one or more conflict order edges between conflicting writes and ending with the write *W*. *R* and R' represent reads, *W*, W_1, and W_2 represent writes.

(1) If $X \xrightarrow{po} Y$ is on a valid path, then $X(i) \xrightarrow{xo} Y(j)$ for all i,j.

(2) If $R \xrightarrow{co} *W \xrightarrow{co} R'$ is on a valid path, or if $W \xrightarrow{co} R$ begins a valid path that ends on a read, or if $R \xrightarrow{co} *W$ ends a valid path that begins with a read, then *W* should be atomic; i.e., for any read R', if $W \xrightarrow{co} R'$, then $W(i) \xrightarrow{xo} R'(j)$ for all i,j.

(3) If a valid path between two writes, W_1 and W_2, could begin and/or end in a $\xrightarrow{co}$ arc, or if $W_1 \xrightarrow{co} W_2$ is on a valid path, or $W_1 \xrightarrow{co} R \xrightarrow{co} W_2$ is on a valid path, then W_1 and W_2 should be coherent; i.e., if $W_1(i) \xrightarrow{xo} W_2(i)$ for any i, then $W_1(j) \xrightarrow{xo} W_2(j)$ for all j.

The above specification is similar (but slightly more aggressive) than the system implied by Proposition 7.13 in Section 7.3.1.3 and has a very similar proof of correctness; it is straightforward to go through the steps of the proof of Proposition 7.13 and verify that the above specification also allows the conclusions made in that proof.

A hardware implementation of the above specification requires providing mechanisms that allow hardware to recognize when two operations can be on a program order arc of a valid path, and to recognize writes mentioned in (2) and (3) above. Hardware can recognize writes mentioned in (2) if special instructions are used for atomic writes; to obey (2), hardware can execute writes from such instructions atomically. Hardware can recognize writes mentioned in (3) if special instructions are used for coherent writes; to obey (3), hardware must ensure that each such write is executed in a coherent manner with respect to all other writes (i.e., for each coherent write, either all its sub-operations are before all sub-operations of any other write, or all its sub-operations are after all sub-operations of any other write). Thus, for (2) and (3), the complexity of the hardware implementation is independent of the specific model; only the compiler needs to be aware of how the high-level programmer identifies atomic or coherent writes for the given model and to convert these writes into the appropriate instruction for the hardware.

In contrast to (2) and (3), the complexity of implementing (1) directly in hardware varies depending on the model; specifically, it depends on how many different types of program order arcs could appear on valid paths. Applying the sender-receiver terminology to valid paths, it follows that the most general mechanism required in hardware is a mechanism to recognize senders, receivers, the operations that a sender sends, the operations that a receiver receives for. Further, a mechanism is required to ensure that a sender is not issued until all the operations it sends are complete, and an operation is not issued until the operations that receive for it complete. For a model like data-race-free-0 where the only senders are synchronization writes that send all operations preceding them, and only receivers are synchronization reads that receive for all operations that follow them, the requirement is straightforward to implement with the aid of a single counter and a mechanism to stall on a receiver. For more complex models, a simple extension of the data-race-free-0 scheme requires a counter per sender category and a bit per receiver category, where the counter indicates the outstanding operations the corresponding sender needs to send and the receiver bit indicates that the corresponding receiver type is outstanding. This scheme can be used only for a limited number of sender and receiver categories, and thus puts a practical bound on the amount of information that hardware exploit with a generic model. Alternative schemes are possible; e.g., Tera [ACC90] uses a field with every instruction to indicate the number of following instructions that can be overlapped with this instruction.

As for data-race-free-0, more aggressive hardware implementations of the high-level valid path requirement are possible. For example, a sender need not wait for all operations it sends to complete; it is sufficient if these operations complete before the next receiver that receives from the sender completes. The completion of the above operations can be delayed further until a processor that receives from the above sender actually accesses the locations of the above operations or executes its following sender. As with lazy release consistency, software-based shared virtual memory systems may also implement the high-level valid path requirement directly [KCZ92].

Control requirement.

Our work so far does not make any assumptions on how the programmer can make the valid paths explicit to the system. However, to define a low-level specification of the control requirement and prove systems correct, we need to make some assumptions about how a system may recognize the valid paths. We also need to make a few assumptions about the structure of the valid paths. We make the following, fairly general, assumptions. (The models and system implementations discussed so far easily obey the assumptions below.)

Condition 7.18: *Assumptions for Valid Paths Made by Low-Level Control Requirement:*

(1) To distinguish an ordering path in an execution as a valid path, the system can take only the following aspects into consideration. (Below, W and R represent a write and a read respectively.)

(a) The instructions that generated the operations on the ordering path.

(b) The addresses accessed by the operations on the ordering path.

(c) The presence of some specific instruction instance (e.g., the MB instruction in Alpha) preceding the instruction instance of an operation on the ordering path.

(d) The presence of some specific instruction instance i (e.g., the MB instruction in Alpha) following the instruction instance of an operation O on the ordering path and such that the instructions of i and O are in the same basic block.

(e) For a $W \xrightarrow{co} R$ arc on the ordering path, W may need to be the last essential conflicting sub-operation before R's sub-operation in the execution order.

(f) For a $W \xrightarrow{co} R$ arc on the ordering path, R may need to return the value of another processor's write in the execution.

(2) $W_1 \xrightarrow{co} R$ and $R \xrightarrow{co} W_2$ (where W_1 and W_2 are writes and R is a read) are not consecutive arcs on any valid path.

(3) $X \xrightarrow{po} Y$ where X and Y conflict is a valid path.

(4) If $R \xrightarrow{co} W$ (where R is a read and W is a write) ends a valid path (say *VP sub 1*) and there is a valid path (say *VP sub 2*) from R to W, then the path obtained from VP_1 by replacing $R \xrightarrow{co} W$ by VP_2 is also a valid path.

The following specification of the control requirement assumes that the system obeys the high-level valid path requirement (in Condition 7.16). Given that, the critical set part of the control requirement is necessary only in the presence of reads that control which code will execute next (through deciding the direction of a branch), or control the address accessed by a memory operation (e.g., through array indexing), or determine the value to be written by a write, or establish valid paths based on the values they return. We call such reads control reads and need to formalize a relation that will relate such reads to the necessary operations they control. The control relation expresses conventional uniprocessor data and control dependence and also certain multiprocessor dependences to preserve valid paths. Intuitively, the uniprocessor dependence part requires that a read control an operation if it either determines whether the instruction instance of the operation will execute, or the address that the operation will access, or the value that the operation will write (if the operation is a write). Intuitively, the multiprocessor dependence part requires that if a read controls an operation X such that $X \xrightarrow{po} Y$ could be an arc on some valid path of the considered model, then the read should control the operation Y. Thus, the control relation depends on the model being considered. Additionally, for the finite speculation part, the control relation also includes dependences due to some types of potentially unbounded loops.

We define the control relation at a high-level that expresses the above properties below; Appendix D gives a low-level and more constructive definition that obeys the following high-level properties. Some of the underlying concepts for the control relation were first developed for the data-race-free-1 model [AdH92] and then formalized for the PLpc model [AGG93]. The material in this section and Appendix D is a generalization of those concepts and formalizations for all models within our framework.

We first explain some terminology used by the following definitions. For two memory operations X and Y, we say $X \xrightarrow{vpo} Y$ in an execution (where $\xrightarrow{vpo}$ stands for valid program order) if $X \xrightarrow{po} Y$ could be a program order arc on some valid path in some execution, given the instruction of X and Y, the address of X and Y, and the other instructions executed by the processor of X and Y. The control relation also requires determining if a memory operation in one execution executes in some sequentially consistent execution and vice versa as follows. (Most of the following is taken from [AGG93].) We say an operation O in execution E_1 executes in some execution E_2 if its instruction instance executes in E_2, and the corresponding operation of the instruction instance in E_2 accesses the same address and writes the same value (if O is a write) as O in E_1. Further, we also require that if some preceding instruction instance can be used to determine whether the operation is on a valid path (as specified in part 1(c) of condition 7.18), then that instruction instance should be present either in both E_1 and E_2, or in neither of E_1 or E_2.[29] For an instruction instance from one execution to execute 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. For instructions that are not parts of loops and in the absence of recursion, it is straightforward to determine if an instance that appears in one execution also appears in another. For instructions that are parts of loops, care has to be taken to match consistent pairs of instruction instances. Instruction instances between two executions are matched consistently if the set 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.)

The definitions below may seem complex. Much of the complexity, however, arises because the definitions allow fairly aggressive implementations. For example, parts (d), (e), and (g) in definition 7.19 and part (2) of definition 7.21 are not necessary if a write following a loop is not executed until it is known whether the loop will terminate in the execution. Similarly, part (3) of definition 7.21 is not required if the system always executes all sub-operations of any write in an execution.

> **Definition 7.19:** *Properties of the control relation (* $\xrightarrow{ctl}$*):* Consider a program *Prog*. Below, E_s represents any sequentially consistent execution of *Prog*, E represents any execution of *Prog*, and a read is said to control an operation ordered after it by the control relation. The control relation orders a read operation before an operation O if the read precedes O by program order, and such that the following properties hold.
>
> (a) If for every read R that controls an operation O in E, R is in E_s and returns the same value in E and E_s, then O is in E_s. Further, if O is an exit read of a synchronization loop, then the exit value of O and the value written by the exit write corresponding to O (if any) are the same in E and E_s.
>
> (b) If for every read R that controls an operation O in E, R is in E_s and returns the same value in E and E_s, and if $O' \xrightarrow{vpo} O$ in E_s, then $O' \xrightarrow{po} O$ in E.
>
> (c) If R controls an operation O' in E and if $O' \xrightarrow{vpo} O$ in E, then R controls O in E.
>
> (d) Consider an instance of a loop such that it does not terminate[30] in E, it terminates in every E_s, and its termination in E_s depends[31] on the value returned by one or more of its shared-memory reads R that could be involved in a race in some sequentially consistent execution without unessential operations. Then a read of the type R above that is not the last such read from its loop instance by program order in E must control any operation O in E such that O is not in R's loop instance and either O is an exit read of a self-ordered loop or O forms a race in some sequentially consistent execution without unessential operations.
>
> (e) R controls O in E if (i) R is either an exit read from a self-ordered loop or R is an exit read from a synchronization loop that forms a race in some sequentially consistent execution without unessential operations, and (ii) either $R \xrightarrow{vpo} O$ in some sequentially consistent execution, or O could form a race in some sequentially consistent execution without unessential operations, or O is an exit read of a

[29] Note that when an operation O is distinguished by an instruction instance *following* the operation, part 1(d) of condition 7.18 requires that the distinguishing instruction be in the same basic block as the instruction of O. Thus, whenever O is in an execution, the distinguishing instruction is in the execution as well.

[30] See appendix B for the definition of an instance of a loop and when an instance of a loop terminates.

[31] For simplicity, the somewhat ambiguous term of ‘‘depends’’ is used here. Appendix D gives a more constructive specification. A simpler, more conservative specification would be to consider all reads in a loop that may be involved in a race in some sequentially consistent execution without unessential operations.

self-ordered loop.

(f) If W, R, and O are in E and E_s, $W \xrightarrow{co} R \xrightarrow{po} O$ is on a valid path in E_s, and the above $\xrightarrow{co}$ arc is on the valid path only if W is the last conflicting essential write before R (by execution order), then $R \xrightarrow{ctl} O$ in E.

(g) If R is an exit read of a synchronization loop instance that does not terminate in E, R controls W in E, and W is not from R's loop instance, then all exit reads in R's loop instance that are from the same static operation as R control W in E.

Definition 7.20: *Control Path:*

A *control/semi-causal-conflict graph* of an execution is a graph where the vertices are the (dynamic) memory operations of the execution, and the edges are due to the transitive closure of the control relation of the execution, or of the type $Write \xrightarrow{co} Read$.

A *control path* for an execution is a path between two conflicting operations in the control/semi-causal-conflict graph of the execution such that no read on the path returns the value of its own processor's write in the execution.

Condition 7.21: *Low-Level System-Centric Specification of the Control Requirement:*

An execution E of program *Prog* obeys the control requirement if it obeys the high-level valid path requirement (Condition 7.16) and the following. (Below E_s represents a sequentially consistent execution of program *Prog*.)

(1) *Critical set:* If there is a control path from R to W, then $R(i) \xrightarrow{xo} W(i)$ for all i.

(2) *Finite speculation.*

(a) The number of memory operations that are ordered before any write operation by $\{\xrightarrow{ctl}\}+$ in E is finite.

(b) Let j be an instance of any instruction j' in E that writes shared-memory or writes to an output interface in E. If j' follows (in E) an instance L of a loop that does not terminate in some E_s, then the number of instances of instructions that are from the loop instance L and that are ordered by program order before j in E is finite.

(3) *Write termination:*

(a) Let operation X and write W be in E and in some E_s. Let X and W be essential in E_s. If there is a race path between X and W in E_s, then W's sub-operation in the memory copy of X's processor must be in E.

(b) Let R and W be in E and in some E_s. Let R and W be essential in E_s. If R is an exit read in E from a self-ordered loop that does not terminate in E, W writes the exit value read by R in E_s, and no ordering path from W to R is valid in E_s, then W's sub-operation in the memory copy of R's processor must be in E.

(4) *Loop Coherence:* If $W_1 \xrightarrow{co} W_2$ and one of W_1 or W_2 is from a synchronization loop, then $W_1(i) \xrightarrow{xo} W_2(i)$ for all i.

The proof of the above specification is the most complex part of this work and appears in Appendix E.

The above requirement may seem complex; however, as discussed below it is straightforward to obey in hardware. For an intuitive understanding of the requirement, observe that the critical set part prohibits the anomalies illustrated by figure 7.11. Parts (a) and (b) of the finite speculation requirement respectively prohibit the anomalies illustrated by figure 7.12(b) and 7.12(a). Part (a) of the write termination requirement prohibits the anomalies of figures 7.13(a) and 7.13(b). Part (b) of the requirement is needed for cases similar to that illustrated by figure 7.13(a) when the loop is self-ordered and there is an ordering path from the write to the loop read but not a valid path. The loop coherence part is the same as the high-level requirement and prohibits the anomaly of figure 7.14.

Note that the control relation and control requirement can exploit information about whether an operation is from a synchronization or self-ordered loop. This information may or may not be directly available from the specification of valid paths.[32] If it is not directly available, then the control relation and requirement have to make conservative assumptions. Note, however, that this information is mainly exploited if either the system allows a write following a loop to be executed before it is known whether the loop will terminate, or the system does not execute all sub-operations for a given write. An exception is the latter part of property (a) which requires that reads that control the exit values of a synchronization loop (and the value of exit writes) should also control the exit reads of the loop. This property, however, is automatically satisfied if the exit value of a synchronization loop instance (and the value of exit writes) is always the same in every execution. This is indeed the case for all the examples of synchronization loops that we have seen so far and we henceforth assume this restriction. (Note that this does not require the exit values to be the same for *every* instance of a synchronization loop. For example, in the barrier code of figure 6.8, the exit values of different instances of the synchronization loop are different; however, the value for each instance is the same in every execution.)

We next consider satisfying the control requirement in hardware. Again, the implementations discussed below are based on those in [AdH92, AGG93, GAG93]. The write termination and loop coherence parts are automatically satisfied in systems that have only a single copy of any memory location, or in systems that employ a cache coherence protocol to ensure that all sub-operations of a write are executed in every memory copy and all processors see conflicting writes in the same order.

For the critical set part of the requirement, the most conservative implementation is to make a processor stall on a read until the read returns its value. A more aggressive implementation can allow a processor to proceed with pending reads as long as it does not execute a write sub-operation W(j) until (a) it is resolved that the instruction instance for W will indeed be executed (i.e., will not have to be rolled back) with the given address and value, and (b) it is known which memory operations preceding W by program order will be executed, which addresses such operations will access, and which values such writes will write. Thus, reads can always be executed speculatively, but writes need to wait until the control flow for the execution and the addresses and values to be accessed by different operations before the write are resolved. Further, corresponding to part (f) of definition 7.19 for the control relation, an operation O should be stalled for a preceding read R if $W \xrightarrow{co} R \xrightarrow{po} O$ could be on a valid path depending on whether W is the last essential conflicting write before R. Note that this delay is already imposed by all the implementation proposals for the valid path requirement so far.

For the finite speculation part of the requirement, part (a) is obeyed if the critical set part is obeyed in either of the two ways described above. For part (b), an additional requirement is necessary that prohibits the speculative execution of an instruction that follows a loop that is not guaranteed to terminate in every sequentially consistent execution. Only shared-memory instructions that generate a shared-memory write or affect the output interface need be prohibited.

Note that the control requirement allows a processor to execute a write even if it is not known whether previous loops (by program order) will terminate in the execution, as long as the loop is known to terminate in every sequentially consistent execution and as long as no memory operation from the loop will be ordered before the write by $\xrightarrow{ctl}$. Programmers usually know whether a loop will always terminate in a sequentially consistent execution, and can provide this information to the system. Since programs are often written so that either they terminate in all sequentially consistent executions, or there are no shared-memory operations that follow a potentially non-terminating loop, aggressive systems can potentially take advantage of the above optimization for many programs.

The above methods are conservative; a processor can always obey the requirement by exactly following the system-centric specification, which is more aggressive.

[32] For synchronization loops that can be exploited by a model, at least one of the exit reads must be involved in a race in some sequentially consistent execution or must be exploited as self-ordered. These attributes are often recognizable from the valid path information.

7.5.3. Compiler Implementations

Compiler constraints due to the low-level system-centric specifications are analogous to those for hardware. We consider the compiler optimizations of reordering operations of a process and allocating shared-memory locations in registers. We reason about register allocation in the same way as for data-race-free-0; part of the reasoning was done jointly with Kourosh Gharachorloo for the work in [AGG93, GAG93]. Thus, we allow two types of intervals over which a memory location can be allocated to a register. The first type allows only read operations to be substituted by register operations and is preceded by a start memory read. The second type begins with a write that is replaced by a register operation and ends with an end memory write (called the flush write). Then register reads in the first type interval can be modeled as memory operations that execute just after the start read of the interval, while register operations in the second type of interval can be modeled as memory operations that execute just before the end write of the interval. The following discusses the constraints on the compiler imposed by the valid path and control requirements. Much of the discussion parallels that for data-race-free-0 in Section 5.4, and evolved from joint work in [AGG93, GAG93]. Recall that the compiler must ensure that the relevant constraints are obeyed for all executions (that might be possible on the system the output program will run on). Below, all references to an ordering refer to program order, unless stated otherwise.

First consider the low-level specification of the valid path requirement. The first part states that operations that could form a program order edge of a valid path must be executed in program order. To obey this part, the compiler must not reorder instruction instances whose operations could form a $\xrightarrow{vpo}$ arc in any execution. Further, for a register allocation interval of the first type, between the start read of the interval and the end of the interval, there should not be any receiver that could have received for a read that was substituted by a register operation in the interval, and no read substituted by a register operation should be a sender for any operation in the interval. Similarly, for a register allocation interval of the second type, between the beginning of the interval and the end write of the interval, there should not be any sender that could have sent an operation that was substituted by a register operation in the interval, and no operation substituted by a register operation should be a receiver for any operation in the interval. The second and third parts of the valid path requirement concern write atomicity and coherence and are not usually relevant to the compiler.

Next consider the control requirement. The critical set part requires that operations ordered by a control path should be executed in that order. A control path consists of $\xrightarrow{ctl}$ edges from reads to writes and conflict order edges from writes to reads. Such a path can be maintained by ensuring that a read that controls a write is not reordered with respect to that write. For register allocation, there are no constraints on the first type of interval since such an interval does not reorder a read preceding a write. There are no constraints on the second type of interval either since whenever such an interval reorders a read preceding a write, the read returns the value of its own processor's write. Such a read does not occur on a control path, and so it is not constrained by the critical set part. Thus, the critical set part of the control requirement does not impose any restrictions on register allocation.

For the finite speculation part, part (a) is satisfied if the critical set part is obeyed as described above, and the following is assumed for register allocation intervals of type 2: if a memory read from a potentially unbounded loop is replaced by a register read, then the flush write for the interval must be in the loop. For part (b) of the finite speculation part, an additional requirement is that for every loop that may not terminate in some sequentially consistent execution of the input program, no instruction instance that writes shared-memory or an output interface following the loop is reordered with respect to the loop instruction instances.

The write termination part is usually not relevant to the compiler, given that a memory write substituted by a register write is always flushed. Similarly, the loop coherence part is also usually not relevant to the compiler.

Recall from the discussion on data-race-free-0 that the compiler must also obey the general requirements of an execution; i.e., the uniprocessor correctness condition and the condition that only a finite number of sub-operations are ordered before any other by the execution order (Definition 5.5). The constraints on the compiler for this are the same as those for data-race-free-0 and are repeated here from Section 5.4 for completeness.

Traditional uniprocessor compilers obey the major part of the uniprocessor correctness condition constraints. However, additional care is required in the presence of register allocation in multiprocessors. The uniprocessor correctness condition is satisfied if an execution always has an end operation for every register allocation interval of the second type. Specifically, whenever an operation from an unbounded loop is allocated in a register, the corresponding end operation should also be in the loop. The condition for execution order described above is relevant to the compiler in the presence of register allocation. This condition can be met if whenever an operation from an unbounded loop is allocated in a register, then the start or end memory operation of the allocation interval is also in the loop.

The compiler must also ensure that it correctly translates the information on valid paths present in its input high-level program to its output low-level program. The translation procedure is specific to specific models, depending on the type of information communicated and the hardware and software mechanisms for communicating the information. Note that with register allocation, the compiler needs to be careful if communicators are substituted with register operations; in such a case, the corresponding start or end memory operation should have the equivalent properties for communicating critical paths as the register allocated communicator. For example, if the register allocated communicator in an interval of the first type receives for all following operations, then so must the start read.

Finally, if hardware optimizes loops that are guaranteed to terminate in sequentially consistent executions by speculatively executing operations beyond such loops (as discussed in the previous sub-section), then the compiler should translate any information required for this optimization consistently as well.

7.6. Relation with Previous Work

This section discusses the relationship of our work with several previous studies.

7.6.1. Relation with Work by Shasha and Snir

Shasha and Snir have proposed a compile time algorithm that allows parallel and out of program order execution of memory operations [ShS88]. Their scheme aims to statically identify a minimal set of pairs of operations within a process, such that delaying the issue of one of the elements of each pair until the completion of the other is sufficient for sequential consistency. The algorithm uses the program order relation P, and a conflict relation C which is true for any pair of conflicting operations. The algorithm requires finding the ‘‘minimal’’ cycles in the graph of $P \cup C$. Such cycles are called critical cycles and the operations on the P edges of critical cycles are called critical pairs. They show that imposing delays on only the critical pairs is sufficient to ensure sequential consistency.

The algorithm, however, depends on detecting conflicting data operations at compile time; therefore, its success depends on data dependence analysis techniques, which may be quite pessimistic. Furthermore, the algorithm presented in detail assumes straightline code with no branch instructions. The paper then identifies two extensions to accommodate code with branches. The first technique requires imposing delays between basic blocks and considers each pair of basic blocks from different processors in isolation to determine critical pairs. The second technique requires assuming that every possible execution path of each processor can execute concurrently with every possible execution path of other processors; the critical pair analysis is applied to all such combinations irrespective of whether they are possible on sequentially consistent systems. The paper mentions that a polynomial time algorithm (in the number of nodes in the graph) can be used for their scheme (with bounded nesting of loops and conditionals), but no details of the algorithm are given. Subsequently, Midkiff et al. have shown a detailed and more practical algorithm to apply the above work to programs with branches and loops [MPC89], but this also requires a global data dependence analysis.

Our condition for sequential consistency in Section 7.1.1, and the reasoning for relaxing the program order constraints in Section 7.1.2 uses reasoning similar to the algorithm by Shasha and Snir, but requires that the programmer provide the information to the system. We believe it would be difficult for programmers to use the work in [ShS88] (and [MPC89]) because it would require programmers to look at all possible execution paths of a process, including those not possible with sequentially consistent executions. We overcame this problem by imposing the control requirement on implementations. Furthermore, our work allows the programmer to exploit the knowledge of dependences that result in a fixed, known order of execution of certain conflicting operations. We exploit this knowledge through the notions of synchronization loops and self-ordered loops. These allow more optimizations by using more information about the program than just the conflict relation used in [ShS88]. (The work in [MPC89] partly exploits the above type of knowledge by fixing the orientation of conflict order arcs between synchronization operations whose order of execution is known.) Our work also leads to aggressive implementations (such as for data-race-free-0 in Chapter 5) that do not necessarily require the system to impose delays on program order edges of critical cycles. Finally, we also examine the optimizations of executing writes non-atomically and eliminating acknowledgements in cache-based systems; the work in [ShS88] implicitly considers systems with only one copy of a line (i.e., writes are atomic).

7.6.2. Relation with Work by Collier

Collier has developed a general framework to define memory models using his abstraction of a shared-memory system (described in Chapters 2 and 5) [Col84-92]. We have adopted Collier's abstraction for system-centric specifications of our models. Collier defines architectures (or memory models) as sets of rules, where each rule is a restriction on the order of execution of certain sub-operations (see Chapter 2 for examples). Our work focuses on systems that can provide the illusion of sequential consistency by using information from the programmer. The contribution of this chapter is in exposing a mapping between information and possible system optimizations that can preserve this illusion, and in using the mapping to develop memory models. Collier's framework does not make apparent such a mapping, and his models are not expressed in terms of sequential consistency (with an exception of few models where writes to all locations are seen in the same order by all processors). Our system-centric specification for sequential consistency in terms of acyclic program/conflict graph, however, is a direct consequence of Collier's specifications. Furthermore, Collier's abstraction for shared-memory systems has been an invaluable aid for reasoning about non-atomic systems, specifying such systems, and proving the correctness of our system-centric specifications.

7.6.3. Relation with Work by Bitar

Bitar has proposed the ‘‘weakest memory access order’’ [Bit92]. He proposes a weakest order for the processor to issue memory operations and for the memory modules to execute memory operations. For the processor, he associates a tree of locks and unlocks with every memory operation. A memory operation can be issued as soon as the ‘‘first’’ lock is obtained and must complete only before the first unlock. However, he does not precisely state what these locks and unlocks are and what their exact relationship is to the memory operation under consideration. Further, we also show that requiring some operations of a processor to be executed in program order cannot constitute a weakest requirement. As long as critical paths are executed safely, all operations of a single processor may be executed in parallel.

For the memory module, Bitar uses the theory of Shasha and Snir and mentions that only conflicting operations that could be part of a critical cycle should be executed by the memory in the correct order. Other operations can be executed in any order. To determine what the critical cycles are, he says that the program should be labeled properly but he does not clarify what that means for the programmer and how critical cycles can be identified. In contrast, our work shows how programs can be ‘‘labeled properly’’, and using only information from sequentially consistent executions.

Finally, he extends his work to cache based systems, but does not give a precise formulation of the requirements for such systems. In particular, he does not consider the optimizations of non-atomic writes and eliminating acknowledgements.

7.6.4. Relation with Data-Race-Free and PLpc models

We first discuss how our framework relates to the work on the data-race-free and PLpc memory models, and then discuss how the new models of Section 7.3 compare with the data-race-free and PLpc models.

The notions of synchronization loops, unessential operations, and self-ordered loops are generalizations of similar notions used for PLpc. The distinctions that make the definitions in this chapter more general are the following. First, this chapter separates the notions of synchronization loops and self-ordered loops; PLpc combines the two ideas. Second, synchronization loops and self-ordered loops of this chapter are not restricted to a single exit read per loop as in PLpc. The model using the signal-await construct in Section 7.3.2 would not be possible without the above extension.

The generic system-centric specifications of Sections 7.4 and 7.5 are generalizations of the specifications for data-race-free-1 in [AdH92] and PLpc in [AGG93]. Both of these models express their conditions essentially in the form of a valid path requirement and a control requirement. (The control requirement for data-race-free-1

[AdH92] is more conservative than Condition 7.21, while that for PLpc [AGG93] is similar to Condition 7.21.) However, the data-race-free-1 and PLpc specifications do not explicitly show the relationship between the two requirements and the information the models derive from the programmer. We have generalized the work on these two models to make explicit this relationship in a manner that exposes the design space and allows system designers to directly transform optimizations and information about programs into useful memory models and implementations. Specifically, by giving a common proof of correctness of a generic system-centric specification (that is relatively easy to translate to implementations), we have eliminated the need for the complex proofs that were earlier necessary to prove the correctness of the data-race-free-1 and PLpc implementations.

Our generalization has made possible new memory models that allow optimizations to be applied to more cases than allowed by either the data-race-free or PLpc models. Each of the eight sub-sections of Section 7.3 describes at least one such model. These models allow for greater overlapping of memory operations of a process and more writes can be executed non-atomically. Further, Section 7.3.1.4 describes how to eliminate acknowledgements, an optimization not considered by the work describing the data-race-free and PLpc models.

7.6.5. Relation with a Framework for Specifying System-Centric Requirements

Jointly with others, we have proposed a framework for uniformly specifying previous hardware-centric models and system constraints for future memory consistency models [GAG93]. This framework uses an extension of Collier's abstraction that also models the equivalent of a write buffer in a processor. The following first discusses how the extension differs from the original abstraction by Collier and why we chose the original abstraction for this work. We then discuss how the methodology for specifying system-centric requirements in [GAG93] relates to the methodology of this chapter.

The extended abstraction explicitly models the equivalent of a write buffer in a processor. A write operation now involves an additional initiation sub-operation that can be viewed as the write being placed in the write buffer of its processor. As with Collier's model, in an *n* processor system, a write also can have *n* memory sub-operations, each representing the write updating the respective memory copy. A read returns the value of the ‘‘last’’ conflicting write placed in its processor's write buffer if such a write exists; if there is no such write, then the read returns the value in its processor's memory copy.

The advantage of explicitly modeling a write buffer is that as soon as the read returns the value of a write from a write buffer, the read sub-operation can be considered to have occurred; the sub-operation of the write itself may occur later in the execution order. Collier's abstraction allows a system to have write buffers, but does not allow a read to return the value of a write whose conflicting sub-operation is after the read in the execution order. Thus, with Collier's abstraction, when a read returns the value of a write from a write buffer, the read sub-operation cannot be considered to have occurred until the write sub-operation occurs. In a hardware cache-coherent system, the write sub-operation would occur only when the write gets ownership of the requested line. Meanwhile, other conflicting writes from other processors could also occur and the read would be ordered after them all in the execution order. Thus, although the above writes happen after the read returns its value in real time, they must be viewed as having happened before the read. Specifically, if the read were a receiver, it would have to ensure that it receives all the necessary information about operations that are sent by the above writes. Similar interactions are possible when allocating shared-memory locations in registers and when a processor writes and subsequently reads from a cache line before getting ownership [GAG93, GGH93].

Thus, the extended abstraction allows more direct modeling of certain interactions than the original abstraction by Collier. However, we have found that specifying systems with Collier's abstraction makes it easier to reason about the mapping between program information and optimizations that will give sequential consistency, and so we chose to use that abstraction for our work. There are some interactions that can be represented easily with the extended abstraction but seem difficult (and we believe not possible) to represent cleanly with Collier's abstraction. However, we have so far found that we can always represent systems with Collier's abstraction that have, for all practical purposes, the same valid paths as systems with the additional interactions and are more aggressive than the above systems. Thus, if only SCNF models are considered, Collier's abstraction seems to be sufficient, and has the additional advantage of not imposing constraints on the system that are not likely to be exploited by the programmer. In conclusion, while we believe the use of the extended abstraction will aid implementors, we consider the original Collier's abstraction as more appropriate for reasoning about when a system will appear sequentially consistent. It would be interesting to formalize results that would allow translating specifications from one abstraction to another easily. Theorem H.1 in Appendix H formalizes one type of translation for some systems and is used to prove results for the PLpc models [AGG93].

We next discuss the methodology for specifying system constraints in [GAG93]. The methodology is an extension of the data-race-free-1 specifications and has also been used for system-centric specifications of PLpc [AGG93]. The methodology for the system-centric specifications of this Chapter is based on the above work. Specifically, [GAG93] proposes specifying systems as constraints on conflicting sub-operations, and shows how such specifications are more aggressive but semantically equivalent to previous specifications of hardware-centric models. The observation that *all* system constraints (other than finite speculation) can be expressed in terms of constraints on conflicting sub-operations was made by Kourosh Gharachorloo while working on the PLpc model and led to the work in [GAG93]; the data-race-free-1 model made this observation only for part of the system constraints. Effectively, [GAG93] proposes that system specifications should give the ordering paths that are executed safely by the system (along with a write termination requirement, a requirement for the initiation sub-operation of a write, and finite speculation). However, unlike the work in this chapter, the work in [GAG93] is concerned only with specifying systems in the most aggressive manner. This chapter tries to determine the relationship between the ordering paths executed safely by the system and information from the program that will give sequential consistency to such a system, and to propose new SCNF models based on this relationship. The work in [GAG93] does not provide insight for such a relationship, and uses the work of the current chapter to ensure that specifications of previous hardware-centric models appear sequentially consistent to data-race-free/PL/PLpc programs.

7.6.6. Relation of New Models with Other Models

Section 7.3 proposed several new models that are related to recently proposed models not unified by the data-race-free and PLpc models. These relationships were discussed along with the relevant models in Section 7.3; the following briefly summarizes them. The following discusses models related to the optimization of executing memory operations out of program order, pipelining operations, executing writes non-atomically, and eliminating acknowledgements in the above order.

Gibbons and Merritt have relaxed release consistency by allowing a release to be associated with a subset of the preceding operations [GiM92]. They show that it is sufficient for a release (a sender in our terminology) to wait for only the preceding data operations that are associated with it to complete, for preceding synchronization writes to complete, and for preceding synchronization reads that return the value of a synchronization write of another processor to complete. Section 7.3.1.1 shows how to generalize the above observation by allowing a receiver (an acquire in the terminology of [GiM92]) to be associated with operations it receives for as well, and allow more optimizations with acquires. Further, we also allow a sender to not wait for the preceding synchronization operations that it is not associated with. Finally, Gibbons and Merritt do not state any control requirement; as stated, their conditions seem to allow the anomaly of figure 7.11(b). Another contribution of the work by Gibbons and Merritt is that they effectively allow program order to be a partial order per process. We have adopted their approach in our work. With this, some of the reordering optimizations in the previous sections could be achieved if the programmer simply indicated that the considered operations are not ordered by program order. Nevertheless, our work shows how programmers (and memory model designers) can reason when program order can be safely relaxed, identifies mechanisms to communicate this information, and identifies common cases where the program order can be safely relaxed.

The entry consistency model allows more data operations to be executed out-of-order (with respect to other data or synchronization operations) than the SCNF models discussed in previous chapters. (Section 7.3.1.1 discusses why we believe entry consistency does not allow reordering of two synchronization operations). Our models allow more pairs of synchronization operations to be executed out of order as well. Further, entry consistency requires data accesses to always be protected by a lock to appear sequentially consistent; we remove this restriction. Finally, entry consistency is not accompanied by a proof of correctness. Our work provides such a proof and in doing so shows that an additional control requirement and added constraints on synchronization operations are necessary for an entry consistent system to appear sequentially consistent. To eliminate the restriction on the use of locks by entry consistency, the Midway system allows data to be declared as either release, processor, or entry consistent [BZS92]. Our work extends such a model as well by proposing more optimizations (e.g., as in section 7.3.2.3).

The model for allowing reordering of operations across barriers (Section 7.3.2.2) is strongly related to the notion of fuzzy barriers [Gup89]. Our work is a straightforward generalization and formalization of the fuzzy barrier work that allows programmers to give the necessary information for the optimization.

For the optimization of pipelining memory operations in Section 7.3.1.2, we have used Carlton's approach [Car91], but remove the restriction of only lock-based synchronization and the restriction of requiring a lock and data associated with it to be in the same memory module for every lock. We also provide the proof of correctness for this optimization.

To the best of our knowledge, the optimization of non-atomic writes has been considered only for systems already allowed by data-race-free/PLpc models and by Collier [Col84-92]. As discussed above, the only case for which Collier seems to allow non-atomic writes and guarantee sequential consistency is when writes to all locations are seen in the same order by all processors. The implementation described requires a restricted (ring based) network. Our models do not impose the above restrictions. Compared to data-race-free-1 and PLpc, we have shown models that allow more writes to be non-atomic (e.g., the signal constructs of Section 7.3.2.1).

The optimization of not requiring acknowledgements on certain accesses has been considered by Collier [Col84-92], Landin et al. [LHH91], and Carlton [Car91]. Collier's scheme has the limitation described above for non-atomic writes. Landin et al. also required a restricted (race-free) network. Carlton assumes only lock-based synchronization and restricts locks and the data they protect to be in the same memory module. Our models allow this optimization without the above restrictions; for example, we show that for CS_unlocks, acknowledgements are not required irrespective of the underlying hardware.

For all the optimizations, our approach of considering common programming constructs and specifying memory models in terms of how such constructs should be used leads to easier-to-use models (even if for some cases, the considered optimizations are already allowed by some previous models).

Finally, our framework for reasoning about the design space of memory models can be viewed as a unification of all of the above models.

7.7. Conclusions

Previous chapters of this thesis argued that memory models be proposed using the SCNF approach, and proposed several SCNF models that unified many of the earlier models. Since the work on those SCNF models, other researchers have proposed several other models. Rather than develop yet another new SCNF model that would unify the other models and/or allow more optimizations, this chapter seeked to explore the design space of SCNF memory models. We developed a framework that provides insight into the optimizations and program information that a memory model can exploit, and provides a relatively straightforward mapping between an optimization and the information (from sequentially consistent executions) that allows the optimization to be performed without violating sequential consistency. We used this framework to design several new memory models and showed that there are optimizations and information about common programming constructs that are not fully exploited by current programming models. We used our framework to characterize the design space in a manner that provides qualitative insight into the programmability and performance of the various points of the design space. The following discusses three potential limitations of this work for SCNF models.

First, our work mainly concerns the design process for a memory model. The final decision on which memory model to use for a system requires a quantitative analysis of the cost/performance ratio for the various models. The cost is the system support and programmer effort required for the specific model. The performance benefit depends on how often the optimizations exploitable by the model can be safely applied for the application program domain of the system. Our work only provides a qualitative insight for this step. Based on our experience with a few programs so far, the concluding chapter of this thesis conjectures key optimizations and mechanisms that may be useful to support in the future to adequately address the 3-P criteria of performance, programmability and portability for memory models. More detailed quantitative analysis is required to evaluate our conjecture, and is a subject of future work.

Second, this work may be criticized as being too complex. However, we maintain that the subject is inherently complex as demonstrated by other formal treatments of the subject, and the inaccuracies detected in many of the informal treatments. In contrast to previous formal treatments, the analysis for each of the new memory models developed in this chapter is simpler and more intuitive. The major complexity of this work is in the proof of Appendix E, which need no longer be confronted by most users, designers, and implementors of memory models. An exception to this is discussed in the next limitation below.

The third limitation of this work is that the part of the design space exposed by our framework depends on our definition of the critical set. If we could eliminate more ordering paths from the critical set, it may be possible to allow more optimizations than allowed currently by our framework. In that case, designers of memory models need to ensure that our system-centric specification and proof hold for the new critical set as well, and may have to confront the full complexity of our proof in Appendix E. It would be interesting to determine a characterization of a minimal critical set and a characterization of the type of optimizations not allowed by the current critical set. A related limitation concerns the low-level specification of the control requirement. While the current requirement seems adequate for now, it is possible that future systems may be able to exploit a more relaxed condition. Relaxing the low-level control requirement also requires confronting the full complexity of the proof in Appendix E. It would be useful to develop alternate proof techniques to simplify the proof in Appendix E so that more aggressive specifications of a critical set and the control requirement can be easily incorporated.

Chapter 8

Detecting Data Races On Data-Race-Free Systems

SCNF memory models seek to provide a sequentially consistent interface to the programmer. They achieve this by guaranteeing the appearance of sequential consistency if the programmer writes valid programs (i.e., the program obeys certain conditions specified by the model). It is possible, however, that during program development, the program is not valid because of the presence of a bug. The SCNF models do not guarantee sequential consistency for such programs. To be practically useful, however, SCNF-based systems must provide support so that even while debugging, programmers can assume sequential consistency. This chapter investigates such support for the data-race-free-0 and data-race-free-1 models.[33]

For brevity, this chapter uses data-race-free to denote both data-race-free-0 and data-race-free-1; it uses happens-before ($\xrightarrow{hb}$) to denote $\xrightarrow{hb0+}$ when considering data-race-free-0 and to denote $\xrightarrow{hb1}$ when considering data-race-free-1; it uses synchronization-order ($\xrightarrow{so}$) to similarly denote $\xrightarrow{so0+}$ or $\xrightarrow{so1}$; it uses race or data race to denote the concepts according to definition 4.9 when considering data-race-free-0 and according to definition 6.3 when considering data-race-free-1. This chapter uses the term *pairable synchronization* for data-race-free-1 systems consistent with the definition in Chapter 6, and for data-race-free-0 systems to refer to all synchronization operations. In using the work of this chapter, the above terms should be used consistently for each model.

Data-race-free systems guarantee sequential consistency to data-race-free programs; i.e., programs for which sequentially consistent executions do not exhibit data races. We would like to determine when programs are data-race-free and detect the data races in non-data-race-free programs so that programmers can assume sequential consistency when designing and debugging their programs.

The problem of detecting data races is not unique to data-race-free systems. Even on sequentially consistent systems, the presence of data races makes it hard to reason about a program and is usually considered to be a bug. Much current research in parallel programming has therefore been devoted to detecting data races in programs written for sequentially consistent systems. *Static techniques* perform a compile-time analysis of the program text to detect a superset of all possible data races that could potentially occur in all possible sequentially consistent executions of the program [BaK89, Tay83b]. In general, static analysis must be conservative and slow, because detecting data races is undecidable for arbitrary programs [Ber66] and is NP-complete for even very restricted classes of programs (e.g., those containing no branches) [Tay83a]. *Dynamic techniques*, on the other hand, use a tracing mechanism to detect whether a particular sequentially consistent execution of a program actually exhibited a data race [AlP87, ChM91, DiS90, HKM90, NeM90, NeM91]. While dynamic techniques provide precise information about a single execution, they provide little information about other executions, a serious disadvantage. For these reasons, the general consensus among researchers investigating data race detection is that tools should support both static and dynamic techniques in a complementary fashion [EmP88].

Rather than start from scratch, we seek to apply the data race detection techniques from sequentially consistent systems to data-race-free systems. Static techniques can be applied to programs for data-race-free systems unchanged, because they do not rely on executing the program. Dynamic techniques, however, depend on executing a program. If a program is data-race-free, then all executions of it on a data-race-free system will be sequentially consistent, and the dynamic techniques will correctly conclude that no data races occurred. If a program is not data-race-free, on the other hand, an execution of it on a data-race-free system may not be sequentially con-

[33] Most of this chapter (except Section 8.5) is taken verbatim from a paper jointly written with others [AHM91]. The material is copyrighted by ACM and reproduced here with the permission of all the co-authors.

sistent. Applying the dynamic techniques to such an execution may produce unpredictable results.

This chapter develops a system-centric specification for data-race-free systems that allows data races to be dynamically detected. The key observation we use is that many currently practical hardware implementations of the data-race-free models give sequential consistency at least until the first data races (those not affected by others). We formalize this observation with a condition that all executions on data-race-free systems have a *sequentially consistent prefix* that extends to the first data races (or the end of the execution). With this specification, we show how to use a dynamic approach that either (a) correctly determines no data races occurred and concludes that the execution was sequentially consistent, or (b) identifies the first data races that could have occurred in a sequentially consistent execution with approximately the same accuracy as on a sequentially consistent system. Furthermore, since many currently practical hardware implementations of data-race-free already exhibit the required sequentially consistent prefix, we argue that the new specification is often obeyed for free from the hardware perspective. An aggressive optimizing compiler could violate our condition; however, since debugging already imposes constraints on optimizing compilers, it is not clear how significant our added constraint would be for compilers.

Alternatively, one could use dynamic techniques on data-race-free systems by having data-race-free systems support a slower sequentially consistent mode that is used for debugging (e.g., while debugging, all operations could be distinguished as synchronization operations). The results of this chapter, however, show that such a slower mode is not necessary for detecting data races. Furthermore, we expect that our results will also allow other debugging tools for sequentially consistent systems to be used unchanged on data-race-free systems. If there are no data races in the execution, then the execution is sequentially consistent and other debugging tools can be directly applied. If there are data races, then the presence of the sequentially consistent prefix allows the tools to be applied to the part of the execution that contains the first bugs of the program.

The actual technique for detecting the data races in an execution on a system that obeys our specification was primarily developed by other authors of this joint work. This chapter summarizes the technique and briefly mentions its limitations; a more detailed description appears in [AHM91].

The rest of this chapter is organized as follows. Section 8.1 discusses the potential problems in applying dynamic data race detection techniques to data-race-free systems. Section 8.2 develops the system-centric specification to overcome these problems. Section 8.3 shows how many practical implementations of data-race-free systems already obey the specification of Section 8.2. Section 8.4 summarizes how data races can be dynamically detected on a system obeying our specification, and briefly mentions the limitations of such a technique. Section 8.5 discusses related work and Section 8.6 concludes the chapter.

8.1. Problems in Applying Dynamic Techniques to Data-Race-Free Systems

This section describes the problems that could potentially limit the use of dynamic data race detection techniques on data-race-free systems. We call executions that obey system-centric specifications of the data-race-free models as data-race-free executions.

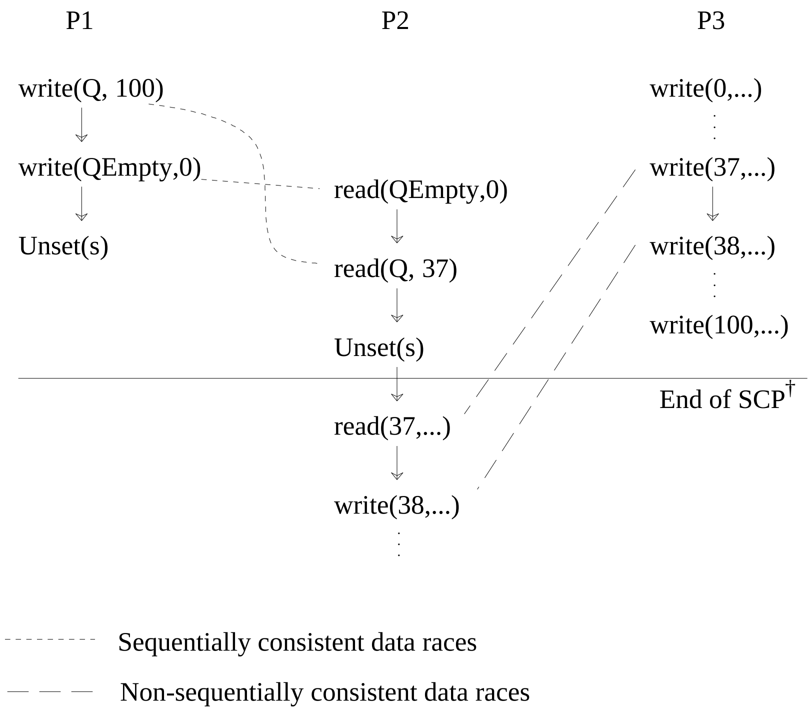
Existing dynamic data race detection techniques for sequentially consistent systems [AlP87, ChM91, DiS90, HKM90, NeM90, NeM91] instrument the program to record information about the memory operations of its execution. This information allows the happens-before relation for the execution to be constructed, thereby allowing the detection of data races. There are two approaches for recording and using this information. The *post-mortem* techniques generate trace files containing the order of all pairable synchronization operations to the same location, and the memory locations accessed between two pairable synchronization operations in a given process. These trace files are analyzed after the execution; the ordering of the pairable synchronization operations allows the synchronization-order relation of the execution to be constructed from which the happens-before relation can be constructed. The pairs of conflicting operations (at least one of which is data) that are not ordered by the happens-before relation are reported as data races. The *on-the-fly* techniques do not produce explicit trace files, but buffer partial trace information in memory and detect data races as they occur during the execution.

The difficulty in applying the above techniques to data-race-free executions is that such executions may not be sequentially consistent (if the program is not data-race-free). For such an execution, we first need to formalize the notion of a data race. This can be done in a manner analogous to that for sequentially consistent executions by using the happens-before relation for the execution. Note that since in general, the pairable synchronization

```
P1                              P2                             P3

compute addr of region          { missing Test&Set }           work on region
    on which to work                                               0...100
                                if (QEmpty=False) then
{ missing Test&Set }
                                      addr := Dequeue();
Enqueue(addr);
                                else ...
QEmpty := False;
                                Unset(S)
Unset(S)
                                work on region
                                    addr...addr+100
```

(a)



†SCP is defined in Section 8.2.

(b)

Figure 8.1. (a) Program fragment, and (b) example happens-before graph showing data races and SCP.

operations of a data-race-free execution are not constrained to be executed in a sequentially consistent manner, the synchronization-order relation and hence the happens-before relation may contain cycles and hence not be partial orders. Nevertheless, the current dynamic techniques for sequentially consistent executions can still be applied to find the data races of a data-race-free execution.

Although data races of a data-race-free execution can be easily detected with the current techniques, there are two potential problems that need to be resolved for the reported data races to be meaningful. The first problem is that for arbitrary data-race-free systems, it is theoretically possible for a data-race-free execution to not exhibit data races and yet not be sequentially consistent. Fortunately, as we shall see later, the implementations of the data-race-free models proposed to date do not exhibit this problem.

The second problem is that data races that may be reported from data-race-free executions may never occur on any sequentially consistent execution of the program. This could easily occur with the current data-race-free systems. Figure 8.1 illustrates a program fragment and one data-race-free execution in which such non-sequentially consistent data races occur. In this program, processor P1 places the starting address of a region for processor P2 to work on in a shared queue and resets the QEmpty flag. Processor P2 checks to determine if work is available, and dequeues an address if the queue is not empty. Processor P3 works independently on some part of the address space. Since P1 and P2 both access the shared queue, operations manipulating the queue are enclosed in critical sections, implemented with the Test&Set and Unset instructions. However, due to an oversight, the Test&Set instructions were omitted and the program is not data-race-free (assuming that the Test&Set and Unset are the only operations distinguished as pairable synchronization operations). A sequentially consistent execution of this program will exhibit data races between the accesses to the queue and the variable QEmpty. Because the program is not data-race-free, a data-race-free execution does not have to appear sequentially consistent. One such execution is shown in Figure 8.1(b), where $op(x,a)$ represents a read or a write memory operation to location x that respectively returns or stores the value a. In this execution, although processor P2 finds QEmpty to be reset, it does not read the new value, 100, enqueued by P1. Instead it reads an old value, in this case 37. The region that processor P2 starts working on now overlaps with the region accessed by P3. This gives rise to many data races between the operations of P2 and P3 as shown. On a sequentially consistent system, P2 could never have returned the value 37, and hence these races would never have occurred. Nevertheless, naively using the dynamic techniques would report all of these data races.

For debugging programs for data-race-free systems, we are only interested in detecting data races that would also occur in a sequentially consistent execution of the program. Further, the motivation of detecting data races was to allow programmers to reason in terms of sequential consistency. Therefore, reporting data races that cannot occur in a sequentially consistent execution can be confusing to the programmer and can complicate the task of debugging. The next section develops a system-centric specification that addresses this problem.

8.2. A System-Centric Specification for Dynamic Data Race Detection

This section develops a system-centric specification that addresses the limitations of dynamic data race detection described in the previous section. The specification ensures that for any execution on a system that obeys this specification, a set of data races that would also occur in some sequentially consistent execution can be identified. A later section will show how dynamic detection can be used to report this set.

Intuitively, our specification requires that an implementation guarantee sequential consistency until a data race actually occurs in the execution. Further, once a data race occurs, sequential consistency should be violated only in parts of the execution that are affected by the data race. This will ensure that every execution has a sequentially consistent prefix that contains the first data races (those not affected by others) of the execution. Dynamic techniques can then potentially be applied to this sequentially consistent prefix to report the first races. We introduce the following terminology to formalize our condition. As in Section 7.5.2, the following terminology involves determining when some operation in an execution also occurs in another execution; recall from Section 7.5.2 that we only consider the instruction instance, address accessed and value written by a write (not value returned by a read) in determining if an operation occurs in another execution. Below, $\langle x,y \rangle$ denotes a race between memory operations x and y.

Definition 8.1: A *prefix* of an execution E is a subset of the memory operations of E such that if y is in the prefix and $x \xrightarrow{hb} y$ in E, then x is in the prefix.

Definition 8.2: A prefix of an execution E of a program *Prog* is a *sequentially consistent prefix* or *SCP* of E iff

(1) it is also the prefix of a sequentially consistent execution *Eseq* of program *Prog*, and

(2) if x and y are in the prefix, then $\langle x,y \rangle$ is a data race in E only if $\langle x,y \rangle$ is also a data race in *Eseq*.[34]

An example of an SCP is shown in Figure 8.1(b). Thus, for an execution E of any program, the operations of a processor in its SCP are also the initial operations of the processor in some sequentially consistent execution, *Eseq*, of the program. Further, a data race involving operations in the SCP occurs in E only if it also occurs in *Eseq*. This implies that the set of data races that have their operations in a particular SCP is a valid set of sequentially consistent data races to report.

To enable the identification of data races in an SCP, we propose a condition that in an execution, either a data race has its operations in a specific SCP of the execution, or the data race is affected by another data race with operations in the SCP, where ‘‘affected’’ is defined as follows.

Definition 8.3: A race $\langle x,y \rangle$ *affects* a memory operation z, written $\langle x,y \rangle \xrightarrow{A} z$, iff

(1) z is the same memory operation as x or y, or

(2) $x \xrightarrow{hb} z$, or $y \xrightarrow{hb} z$, or

(3) there exists a race $\langle x',y' \rangle$ such that $\langle x',y' \rangle \xrightarrow{A} z$, and either $\langle x,y \rangle \xrightarrow{A} x'$ or $\langle x,y \rangle \xrightarrow{A} y'$.

A race $\langle x,y \rangle$ affects another race $\langle x',y' \rangle$, written $\langle x,y \rangle \xrightarrow{A} \langle x',y' \rangle$, iff $\langle x,y \rangle \xrightarrow{A} x'$ or $\langle x,y \rangle \xrightarrow{A} y'$.

This implies that data races that are not affected by any other data race (intuitively the first data races) should always be in an SCP of the execution, i.e., they should also occur in a sequentially consistent execution. Thus, the data races that are not affected by any others constitute a valid set of data races that can be reported.

The system-centric specification follows.[35] We say that a race $\langle x,y \rangle$ occurs in an SCP if the operations x and y are in the SCP.

Condition 8.4: For any execution E of a program *Prog*,

(1) if there are no data races in E, then E appears sequentially consistent, and

(2) there exists an SCP of E such that a data race in E either occurs in the SCP, or is affected by another data race that occurs in the SCP.

Condition 8.4(1) ensures that if no data races are reported, then the programmer can safely assume that the systems is sequentially consistent, overcoming the first problem cited in the previous section. Condition 8.4(2) ensures that if a data race is detected in E, then there is a data race in E that affects this data race that also occurs in some sequentially consistent execution of the program. Thus, the set of data races that are not affected by any other data race in E form a valid reportable set of data races that also occur in some sequentially consistent execution.

[34] The definition of an SCP in [AHM91] required $\langle x,y \rangle$ to be a data race in E *if and only if* it is also a data race in *Eseq*. The *if and only if* clause needs to be replaced by *only if*; it is required to prove the theorems of the paper but does not affect any results.

[35] The notion of a system-centric specification is used a little differently in this chapter. Specifically, so far a system obeys a system-centric specification as long as the result of the run of a program on the system is the same as the result of some execution allowed by the specification. The work in since this chapter, however, requires tracing the memory operations during a run of the program; therefore, this work requires that the system obey the requirements of the specification in real time.

8.3. Data-Race-Free Systems Often Obey Condition for Dynamic Data Race Detection

This section discusses why we expect many practical hardware implementations of data-race-free-1 to already obey Condition 8.4, and how the condition might restrict an optimizing compiler. Recall that Appendix F shows that all low-level system-centric specifications of data-race-free-0 in this thesis obey the generic high-level valid path and low-level control requirements of Chapter 7, where the valid paths of data-race-free-0 are $\xrightarrow{hb0}$ paths between conflicting synchronization operations and $\xrightarrow{hb0+}$ paths between other conflicting operations. A similar result holds for data-race-free-1 as well (with $\xrightarrow{hb0+}$ paths replaced by $\xrightarrow{hb1}$ paths.) The following theorem states that the above requirements along with an additional condition also obey Condition 8.4. The following uses the control ($\xrightarrow{ctl}$) relation as defined in Section 7.5.2. Recall also (from Appendix F) that the valid paths for data-race-free-0 are $\xrightarrow{hb0}$ paths between conflicting synchronization operations and $\xrightarrow{hb0+}$ paths between other conflicting operations; the valid paths for data-race-free-1 are the same except that $\xrightarrow{hb0+}$ is replaced by $\xrightarrow{hb1}$.

> **Theorem 8.5:** Condition 8.4 for dynamic detection of data races is obeyed by all executions that obey the generic high-level valid path requirement (Condition 7.16) assuming valid paths for data-race-free models as described above, and the generic low-level control requirement (Conditions 7.21) with the following additional restrictions: (1) the $\xrightarrow{vpo}$ relation in properties (b) and (c) for the control relation (Definition 7.19) should be replaced by $\xrightarrow{po}$, (2) if $R \xrightarrow{po} O$ in E where R is a pairable read, then $R \xrightarrow{ctl} O$ in E, and (3) if $R \{ \xrightarrow{ctl} \}+ W$ in E, then $R(i) \xrightarrow{xo} W(j)$ for all i, j in E.

The proof of the above theorem appears in Appendix I. The following gives the intuition for the correctness of the above theorem and then explains how the added constraint on control reads of the above theorem affects practical implementations.

Condition 8.4(1) requires that if an execution has no data races, then it should appear sequentially consistent. In an execution without data races, all conflicting synchronization operations are ordered by $\xrightarrow{hb0}$ and all other conflicting operations are ordered by $\xrightarrow{hb0+}$ (or $\xrightarrow{hb1}$ for data-race-free-1); therefore, all critical paths are valid paths of the data-race-free model. It follows that the execution obeys Condition 7.6 for sequential consistency, assuming write termination and finite speculation. Appendix I shows that the control requirement ensures that the execution also appears to obey the write termination and finite speculation assumptions, thereby obeying Condition 8.4(1).

Condition 8.4(2) requires that every data race in a data-race-free execution that is not affected by any other data race should also occur in a specific sequentially consistent execution of the same program. The data races that are not affected by any others are intuitively, the first data races in an execution. Therefore, Condition 8.4(2) can be obeyed by ensuring that an execution provides sequential consistency until a data race actually occurs. Even then, a violation of sequential consistency should be allowed to occur only for operations that are directly affected by the data race. Intuitively, this is true for all data-race-free implementations that obey the condition of theorem 8.5 for the following reason. The data-race-free implementations proposed to date are allowed to violate sequential consistency only for executions that exhibit data races. Practically, however, it is not possible to predict whether an execution will exhibit a data race until a data race actually occurs in the execution. Therefore, if a processor does not execute operations until the course of the execution preceding the operation is already determined, it follows that the data-race-free implementation will provide sequential consistency for all operations until the first data race. Further, it will violate sequential consistency only in the parts of the execution that are affected by the data races. The added constraint of theorem 8.5 ensures that a processor does not execute operations until the course of the execution preceding the operation is already determined, thereby obeying condition 8.4.

Figure 8.2 illustrates the need for the added constraint on control reads as follows. Assume all operations in the figure are data operations. In every sequentially consistent execution of the illustrated program, the reads of X and Y by processors P_1 and P_2 should return the initial value 0 and the italicized data operations are not executed. Without the additional requirement, however, the following execution is possible: (1) P_1 and P_2 execute their writes of Y and X respectively, (2) P_1 and P_2 execute their reads of X and Y respectively, which return the value

Initially X = Y = 0

```
P1                      P2
if (X == 1) {           if (Y == 1) {
   data operations         data operations
}                       }
Y = 1                   X = 1
```

Figure 8.2. Motivation for extra constraint in theorem 8.5.

1, (3) P_1 and P_2 execute their italicized data operations. This execution does not appear sequentially consistent. The only data races in this execution that could occur in a sequentially consistent execution are due to the operations on *X* and *Y*. However, any prefix that includes these races must include the italicized data operations and so is not a sequentially consistent prefix. Note, however, that in the above execution also, the ‘‘first’’ data race would occur on a sequentially consistent execution and all the violations of sequential consistency occur only because of the data race; the problem encountered is that the execution up to the data race no longer consists of operations that could occur on a sequentially consistent execution.

We next discuss how the added constraint on control reads of the above theorem affects implementations of data-race-free systems proposed to date. Previous chapters have proposed system-centric specifications for the data-race-free models that do not require the additional constraint. When compared to the low-level system-centric specifications, the additional requirement can limit performance in two ways. First, it does not allow writes to be executed speculatively. Although the aggressive specifications allow writes to be executed speculatively (assuming the control path of Section 7.5.2 is maintained), this is not currently practical since it could potentially result in global rollbacks. The second limitation of the additional constraint is that it requires a write to wait until it is known which preceding operations will be executed, which addresses they will access, and which values they will write. The low-level specifications require a write to wait only if the preceding operations could be one of certain operation categories (e.g., if the write is data, then it needs to wait to determine if a preceding operation could be a pairable read synchronization or could conflict with the write). Practically, however, we expect that most hardware will obey the conservative condition of theorem 8.5. Thus, we conclude that the additional constraint is already obeyed by most practical data-race-free hardware. For an optimizing compiler that reorders memory operations, it may be possible to exploit the more aggressive condition. However, debugging with optimizing compilers already often imposes constraints on the compiler, and it is not clear if the above constraint would be a significant restriction.

8.4. Detecting Data Races on Data-Race-Free Systems

This section summarizes how Condition 8.4 can be used to dynamically detect sequentially consistent data races on data-race-free systems, and briefly mentions the limitations of our technique. A detailed description of this material appears in [AHM91].

A post-mortem approach can be used to locate sets of data races, where each set contains at least one data race that belongs to a specific SCP. We can achieve this by instrumenting the program to record the operations (including locations accessed) of every process, and the order of execution of conflicting synchronization operations. This allows constructing the happens-before graph for the execution, which indicates the operations that were involved in a race in the execution. To determine the data races in an SCP, we need to augment the happens-before graph with doubly directed edges between any pair of operations that formed a race in the execution. This graph represents the affects relation between races. Since the graph can have cycles, it does not yet give us the first data races (i.e., the data races from the SCP). To determine the first data races, we need to use the strongly connected components of the graph to partition the data races. The first partitions containing data races (i.e., the partitions with data races that are not ordered after any other partitions with data races in the graph) contain data races from the SCP. It is straightforward to prove that Condition 8.4 ensures that there is a first partition iff there is a data race in the execution, and that each first partition contains at least one data race belonging to a specific SCP. Thus, the first partitions contain the data races which should be given to the programmer.

A limitation of the above approach is that for first partitions with more than one data race, we cannot determine which are the sequentially consistent data races. Nevertheless, the number of data races and the portion of the program that need to be examined are significantly reduced, making debugging much easier. The other limitations of this work are that we do not detect sequentially consistent data races that are not in the first partitions, and the race detection overhead is quite high. We argue in [AHM91] how all the limitations of our technique are analogous to the limitations of dynamic techniques for sequentially consistent systems.

8.5. Related Work

To the best of our knowledge, the only other work related to debugging on non-sequentially consistent systems is by Gharachorloo and Gibbons [GhG91]. This work proposes additional hardware support for a release consistent system that allows runtime detection of violations of sequential consistency (due to programs not properly labeled [GLL90]). The scheme imposes an additional constraint on release consistent (RCsc) hardware requiring acquires to wait for all previous operations. Since release consistent (RCsc) systems obey data-race-free-1 and since data-race-free-1 programs are also properly labeled programs, it follows that the above scheme is applicable to data-race-free systems as well. The advantages of the above scheme compared to our work are the following. First, the debugging overhead for the scheme in [GhG91] is in the additional hardware support and the additional constraint on acquires. This is far less than that incurred by our tracing mechanism. The second advantage is that since the debugging hardware is always enabled, the above scheme can signal violations of sequential consistency for every run of the program on the system. In contrast, our scheme allows detecting data races that occur only for executions that have the debugging support on; since this support involves relatively large overhead, keeping it on is not practical for every execution. The disadvantages of the above scheme are the following. First, the above scheme cannot detect the first data races that may be responsible for the bug in the program. Therefore, unlike our scheme, the scheme in [GhG91] does not (yet) help programmers to reason with sequential consistency while debugging. The second disadvantage is that the debugging overhead due to the additional hardware constraint is incurred with the above scheme even after the program is debugged. The only additional system constraint (for all executions) imposed by our scheme is already obeyed by most currently practical systems.

8.6. Conclusions

The data-race-free-0 and data-race-free-1 models provide high performance by guaranteeing sequential consistency only to programs that do not exhibit data races on sequentially consistent hardware. To allow programmers to use the intuition and algorithms already developed for sequentially consistent systems, it is important to determine when a program is data-race-free; when a program is not data-race-free, it is important to identify the parts where data races could occur.

Detecting data races is also crucial for programs written for sequentially consistent systems. Static techniques for sequentially consistent systems can be directly applied to data-race-free systems as well. Dynamic techniques, on the other hand, may report data races that could never occur on sequentially consistent systems. This can complicate debugging because programmers can no longer assume the model of sequential consistency.

We have shown that a post-mortem dynamic approach can be used to detect data races effectively even on data-race-free systems. The key observation we make is that most data-race-free hardware preserves sequential consistency at least until the first data races (those not affected by any others). We formalize this condition by using the notion of a sequentially consistent prefix. For an execution on a system that obeys this condition, we can either (1) correctly report no data races and conclude the execution to be sequentially consistent or (2) report the first data races that also occur on a sequentially consistent execution (within the limitation discussed in Section 8.4). Since our condition is already met by currently practical data-race-free hardware, our technique can practically exploit the full performance of the data-race-free hardware.

It would be interesting to investigate on-the-fly techniques for data-race-free systems in the future. It would also be interesting to determine if the techniques of this chapter would extend to other SCNF models.

Chapter 9

Conclusions

The abstraction of shared-memory provides important advantages for programming parallel systems. The naive shared-memory model of sequential consistency, however, restricts exploiting many performance enhancing optimizations in hardware and software that have been successfully employed for uniprocessors.

Alternative models for higher performance have been proposed, but suffer from two important disadvantages. First, programming with many of the alternative models is difficult, often requiring programmers to be aware of hardware features such as write buffers and caches. Second, the multitude of current memory models (more than a dozen were proposed in the last four years!), besides being intimidating, thwarts portability. Even so, it is difficult to believe that the future will not bring more models, resulting in an ever-increasing number of shared-memory interfaces for programmers to deal with (this thesis itself indicates another dozen system implementations that could be viewed as new memory models!).

A much needed attribute in the design and specification of memory models today is a unifying framework that can express the rich variety of current system optimizations and help envisage future optimizations, both in a way that would lead to models that adequately satisfy the ***3P*** criteria of ***p***rogrammability, ***p***ortability, and ***p***erformance. This thesis fulfills this need by

- establishing a unifying methodology (SCNF) for specifying memory models that meet the 3P criteria,
- proposing four SCNF memory models (data-race-free-0, data-race-free-1, PLpc1, and PLpc2) that unify several previous hardware-centric models,
- exposing a large part of the design space of SCNF models by formalizing the complex relationship between system optimizations and program information necessary to design SCNF models, and
- making preliminary progress in debugging with relaxed models by showing a technique to detect meaningful data races on data-race-free systems.[36]

Section 9.1 further discusses the key features of this thesis, and Section 9.2 discusses relevant issues not addressed by this thesis.

9.1. Thesis Summary

The key feature of SCNF memory models is that they all provide a constant and simple view of the system to the programmer. This immediately alleviates the problems of programmability and portability. For performance, SCNF models require more active cooperation of the programmer. Typically, they require programmers to make explicit some information about the behavior of the program, assuming the simple system view. Since the programmer writes the program assuming the simple system view, it is natural to expect that the programmer will have this information. Our approach does not give any guarantee about the system behavior if the programmer gives incorrect information. An important effect is that system designers can exploit the information they seek to the maximal extent, thereby optimizing for the expected case (i.e., correct information). In contrast, previous models provide guarantees for *all* programs, potentially sacrificing some optimizations for the more frequent programs.

For the choice of the base system view for all models, we have chosen the model of sequential consistency. This is a natural choice considering the simplicity of sequential consistency, and considering that a large body of algorithms has been developed (often implicitly) assuming sequential consistency. Our general approach outlined

[36] The data race detection work is joint work with others [AHM91]; PLpc1 and PLpc2 are derived from joint work with others on the PLpc model [GAG92].

above, however, is valid with any other base model as well.

The thesis demonstrates the effectiveness of the SCNF methodology by developing four SCNF models (data-race-free-0, data-race-free-1, PLpc1, and PLpc2) based on popular commercial and academic hardware-centric models (weak ordering, release consistency (RCsc), release consistency (RCpc), processor consistency, total store ordering, partial store ordering, IBM 370, and Alpha). Each SCNF model allows programming with sequential consistency and allows more implementations (and potentially higher performance) than the corresponding hardware-centric models. The key features of these SCNF models are that they exploit increasing amounts of information about a program, and they allow the programmer to provide conservative information. This ensures the following for the four models.

– Any program can be run with any of the above models (assuming conservative defaults).

– In moving from a less aggressive to a more aggressive model, programmers can add new information incrementally; meanwhile, they can continue to see the performance benefits of the original model.

– A program written for a more aggressive model can be run on a less aggressive model with all the performance benefits of the latter model; extra information has no penalty and is simply ignored.

Thus, the four SCNF models provide four levels of exploitable information and optimizations. Programmers can write programs for any specific level; running them at higher or lower level systems simply exploits the appropriate level of information and optimizations.

The SCNF approach addresses the 3P criteria well. A cost, however, is the increased complexity of designing the memory model for a system. The relationship between an optimization and the program information that will allow the optimization to be used safely (i.e. without violating sequential consistency) is complex. The key reason for this complexity is that the information can only be from sequentially consistent executions of the program. The thesis alleviates this complexity by developing the control requirement. The requirement is difficult to formalize and prove correct, but is fortunately obeyed by currently practical systems. In particular, it is obeyed by all systems where writes are not executed speculatively, each write is eventually seen by all processors, and writes to the same location are seen in the same order by all processors. The control requirement is a key contribution of this work since most previous work either ignored the issue, or addressed it only informally, or imposed fairly conservative conditions (e.g., processors block on reads, or cannot execute any memory operations speculatively).[37]

Obeying the control requirement allows the use of a straightforward mapping between optimizations and information from sequentially consistent executions. To determine when an optimization is safe, we need to simply determine when the optimization violates the critical paths of a sequentially consistent execution. The corresponding model requires the programmer to make the relevant parts of the above paths explicit, so that the optimization is not applied to that path. Thus, the design space of SCNF models can be succinctly characterized as follows:

– An SCNF model is characterized by a set of ordering paths called its valid paths.

– A system obeys an SCNF model if it executes the valid paths ‘‘safely’’ and obeys the control requirement.

– A program obeys an SCNF model if the critical paths of every sequentially consistent execution are valid paths.

The thesis illustrated these concepts through several new memory models, exploiting new optimizations and old optimizations in new cases for potentially higher performance, and exploiting commonly used programming constructs for easier programmability.

An impediment to the effective use of the SCNF approach is that initially, while debugging, programmers might unknowingly give incorrect information. In this case, sequential consistency is not guaranteed. We have taken a preliminary step in overcoming this problem for the data-race-free models. Programs written for these models are incorrect if they have data races. We have adopted the approach of extending the data race detection work for sequentially consistent systems to data-race-free systems. The key issue is that for dynamic data race detection techniques, we must expose to the programmer only those data races that could also have occurred in

[37] Some of the formalization of the control requirement was done jointly with others in the course of developing the PLpc model [AGG93].

sequentially consistent executions. We develop a system condition to allows this, and show that currently practical data-race-free systems obey this condition. It should be understood that the debugging problem is not any worse with the SCNF models than with the earlier hardware-centric models; the reason it is not an important issue with the earlier models is that they require programmers to reason with non-sequentially consistent executions all the time.

9.2. What next?

There are at least two key, relevant issues not addressed by this thesis. The first issue is the choice of the best model for a system. The answer to this question depends on several variables such as the application domain and other system design constraints. A study that would give a satisfactory answer, especially considering future generations of aggressive processors, is beyond the scope of this work. We, however, make a few conjectures below.

First, considering programmability, experience with a few programs suggests that distinguishing operations on the following bases is reasonably straightforward for programmers:

- non-communicators (i.e., data or non-race) vs. communicators (i.e., synchronization or race)
- communicators that order non-communicators (i.e., paired or true synchronization) vs. other communicators (i.e., unpaired synchronization)

Other distinctions (e.g., loop vs. non-loop operations, and distinctions between other types of communicators) are easy to make for common synchronization scenarios (e.g., locks and barriers), but difficult for general cases. This suggests that from the programmability perspective, any other information would be best camouflaged in terms of the behavior of commonly used programming constructs. Typically, this would involve having library routines with easy-to-obey restrictions on their use, as illustrated by Chapter 7; programmers can use the special routines whenever they meet the necessary restrictions.

Second, considering portability, if all models always provide a ‘‘conservative’’ or ‘‘don’t-care’’ option, then porting programs for correctness is trivial. Specifically, memory models that provide direct mechanisms to distinguish only the types of operations listed above, and seek the remaining information through the use of special constructs guarantee easy portability.

Third, considering performance, earlier detailed hardware simulation studies for release consistency (RCpc) and weak ordering have reported upto 41% speedup in hardware (compared to sequential consistency) [GGH91a, ZuB92]. These studies do not fully exploit non-blocking reads; a study that does exploit such reads indicates better performance, but is trace-based [GGH92]. In most cases, weak ordering performs as well as release consistency (RCpc), but there are some cases where it gives significantly worse performance. However, the above studies analyze programs written for sequential consistency and are not data-race-free-0 or PLpc. To the best of our knowledge, these studies consider only the explicit calls to synchronization libraries (and a few other races) as synchronization. As indicated in Chapter 4, for many programs, these calls do indeed represent most of the races in the program. Therefore, it is reasonable to conclude from these studies that distinguishing between communicators and non-communicators is worthwhile. However, it is difficult to draw any conclusions regarding the distinction between unpaired and paired synchronization, and any other further distinctions, from these studies. More detailed studies, specifically considering future more aggressive processors, larger systems, and a wider breadth of applications (including applications with more fine-grained sharing) are needed to assess the impact of further distinctions. Finally, the above studies are for hardware implemented shared-memory. For software implemented shared virtual memory systems, we expect the differences between different models to be more significant.

The second key issue not addressed in this thesis concerns alternative approaches for specifying memory models that may possibly satisfy the 3P criteria better than SCNF. There are at least three other alternatives worth consideration.

The first alternative is to use an SCNF-like framework, but use a base model different from sequential consistency. Although sequential consistency provides a simple and intuitive view of the system, often programmers do not seem to rely on all of its power. For example, if two processors write (via operations that race) to the same location, should they really be treated as a synchronization? Most programs seem to use only races involving a write and a read, where the write executes before the read, to order other accesses. In fact, since often interactions occurring due to a race between two writes are unintended, it may be difficult for programmers to provide the necessary information to preserve sequential consistency that require considering such interactions. The model of causality, however, only considers interactions due to write/read pairs [AHJ90, HuA90]. Should causality be considered as the base model? Our framework would still be applicable; only now programmers would provide information about the program behavior on causal systems and systems would ensure causality. Correspondingly, in Chapter 7, the program/conflict graph would be replaced by the program/causal-conflict graph.

Potential disadvantages of using causality are the following. First, for naive programmers, it is difficult to envision an analog as simple as figure 1.1 to describe causality. But perhaps, if write atomicity were added, one could describe it simply enough. Second, programs like Dekker's algorithm in Chapter 1 which do rely on synchronizations due to operation pairs other than a write followed by a read will simply not be allowed. Undoubtedly, there are useful programs (e.g., operating systems) that rely on the correctness of such interactions. A pertinent question here is whether writers of such (relatively complex) code could deal with reasoning with more system-centric constraints. These constraints could potentially provide some weak guarantees for such interactions.

The second alternative of a base model is ‘‘no model at all.’’ Thus, the programmer assumes the system is completely unconstrained (except perhaps the guarantee of preserving some intra-processor dependences and write termination). Then programmers would need to explicitly indicate all the ordering paths that they want preserved. (An equivalent converse is to require programmers to explicitly state which ordering paths need not be preserved.) The advantage of such a model is that it does not impose unnecessary constraints on the system; therefore, programmers can potentially tune their programs for maximal performance. This would specifically be useful for users of asynchronous algorithms which need very weak guarantees to converge on the correct answer. The relaxed memory order model for SPARC V9 is an example of such a model. Further, with more powerful hardware primitives such as full/empty bits [Smi82] that enforce implicit synchronization, a system with no model at all may be viable [Goo93].

Finally, we could continue to use SCNF, but also provide an alternative, high-level system-centric specification. Thus, programmers who prefer the simplicity of sequential consistency can continue to reap the advantages of the SCNF methodology, and programmers who like to maximize their performance by dealing directly with the idiosyncrasies of specific systems can also continue to do so. The tradeoff is that if in the future, more aggressive system implementations that obey the SCNF model but not the system-centric specification are envisaged, then either those implementations cannot be exploited or the programs that relied on the system-centric specification would not be portable to the new system. For this reason, the system-centric specification given should be at a high-level. We recommend Collier's model as the abstract system model, and the high-level valid path requirement coupled with the low-level control requirement as the system-centric specification. Note that release consistency (RCsc) uses a similar approach; however, the system-centric specification is fairly conservative (e.g., it does not allow implementations such as lazy release consistency, delayed consistency, and other implementations described in Chapter 5).

There are several other unresolved issues including the correct programming language support for distinguishing different operations, possible techniques to verify whether software obeys necessary constraints, and tools to aid debugging where verification is not possible or not exact. Interactions between relaxed memory models and other latency hiding or reducing techniques are also important to study further [GHG91]. Note that most of the optimizations due to memory models are essentially trying to exploit the implicit, fine-grained parallelism in a single thread. One way to make the memory model a less significant issue is for programming languages to provide the maximum flexibility for programmers to express the fine-grained parallelism that they are already aware of. This is expressed in our work by making the program order of each process partial, as recommended by Gibbons and Merritt [GiM92]. Finally, from a more theoretical standpoint, it would also be interesting to make the critical set and control requirement more aggressive, and the proof of correctness of the control requirement simpler.

Thus, while this thesis has established the necessary unifying framework in which to reason about memory models, more work is certainly required before we converge on the best memory models for shared-memory systems.

References

[AdH90a] S. V. ADVE and M. D. HILL, Implementing Sequential Consistency in Cache-Based Systems, *Intl. Conf. on Parallel Processing*, August 1990, I47-I50.

[AdH90b] S. V. ADVE and M. D. HILL, Weak Ordering - A New Definition, *Proc. 17th Annual Intl. Symp. on Computer Architecture*, May 1990, 2-14.

[AHM91] S. V. ADVE, M. D. HILL, B. P. MILLER and R. H. B. NETZER, Detecting Data Races on Weak Memory Systems, *Proc. 18th Annual Intl. Symp. on Computer Architecture*, May 1991, 234-243.

[AGG91] S. V. ADVE, K. GHARACHORLOO, A. GUPTA, J. L. HENNESSY and M. D. HILL, Memory Models: An Evaluation And A Unification, *Presentation in the Second Workshop on Scalable Shared-Memory Multiprocessors*, Toronto, May 1991.

[AdH92] S. V. ADVE and M. D. HILL, Sufficient Conditions for Implementing the Data-Race-Free-1 Memory Model, Computer Sciences Technical Report #1107, University of Wisconsin, Madison, September 1992.

[AGG93] S. V. ADVE, K. GHARACHORLOO, A. GUPTA, J. L. HENNESSY and M. D. HILL, Sufficient System Requirements for Supporting the PLpc Memory Model, Computer Sciences Technical Report #1200, University of Wisconsin, Madison, December 1993. Also available as Technical Report #CSL-TR-93-595, Stanford University, December 1993.

[AdH93] S. V. ADVE and M. D. HILL, A Unified Formalization of Four Shared-Memory Models, *IEEE Transactions on Parallel and Distributed Systems 4*, 6 (June 1993), 613-624.

[ABM89] Y. AFEK, G. BROWN and M. MERRITT, A Lazy Cache Algorithm, *Proc. 1st ACM Symp. on Parallel Algorithms and Architectures*, July 1989, 209-222.

[ABM93] Y. AFEK, G. BROWN and M. MERRITT, Lazy Caching, *ACM Transactions on Programming Languages and Systems 15*, 1 (January 1993), 182-205.

[ASH88] A. AGARWAL, R. SIMONI, M. HOROWITZ and J. HENNESSY, An Evaluation of Directory Schemes for Cache Coherence, *15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii, June 1988, 280-289.

[ALK90] A. AGARWAL, B. LIM, D. KRANZ and J. KUBIATOWICZ, APRIL: A Processor Architecture for Multiprocessing, *Proc. 17th Annual Symposium on Computer Architecture*, June 1990, 104-114.

[AHJ90] M. AHAMAD, P. W. HUTTO and R. JOHN, Implementing and Programming Causal Distributed Shared Memory, College of Computing Technical Report GIT-CC-90-49, Georgia Institute of Technology, November 1990.

[ABJ93] M. AHAMAD, R. BAZZI, R. JOHN, P. KOHLI and G. NEIGER, The Power of Processor Consistency, *Proc. Symposium on Parallel Algorithms and Architectures*, 1993.

[ASU86] A. AHO, R. SETHI and J. ULLMAN, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[AlP87] T. R. ALLEN and D. A. PADUA, Debugging Fortran on a Shared Memory Machine, *Proc. Intl. Conf. on Parallel Processing*, August 1987, 721-727.

[ACC90] R. ALVERSON, D. CALLAHAN, D. CUMMINGS, B. KOBLENZ, A. PORTERFIELD and B. SMITH, The Tera Computer System, *Proc. International Conference on Supercomputing*, Amsterdam, June 1990, 1-6.

[ArB86] J. ARCHIBALD and J. BAER, Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model, *ACM Trans. on Computer Systems 4*, 4 (November 1986), 273-298.

[AtF92] H. ATTIYA and R. FRIEDMAN, A Correctness Condition for High-Performance Multiprocessors, *Proc. Symp. on Theory of Computing*, 1992, 679-690.

[ACF93] H. ATTIYA, S. CHAUDHURI, R. FRIEDMAN and J. WELCH, Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies, *Proc. Symp. on Parallel Algorithms and Architectures*, 1993.

[BaZ91] J. BAER and R. N. ZUCKER, On Synchronization Patterns in Parallel Programs, *Intl. Conference on Parallel Processing*, 1991, II60-II67.

[BaK89] V. BALASUNDARAM and K. KENNEDY, Compile-time Detection of Race Conditions in a Parallel Program, *3rd Intl. Conf. on Supercomputing*, June 1989, 175-185.

[Bel85] C. G. BELL, Multis: A New Class of Multiprocessor Computers, *Science 228*(April 1985), 462-466.

[BCZ90] J. K. BENNETT, J. B. CARTER and W. ZWANAEPOEL, Adaptive Software Cache Management for Distributed Shared Memory Architectures, *Proc. 17th Annual Intl. Symp. on Computer Architecture*, Seattle, May 1990.

[Ber66] A. J. BERNSTEIN, Analysis of Programs for Parallel Processing, *IEEE Trans. on Electronic Computers EC-15*, 5 (October 1966), 757-763.

[BeG81] P. A. BERNSTEIN and N. GOODMAN, Concurrency Control in Distributed Systems, *Computing Surveys 13*, 2 (June, 1981), 185-221.

[BeZ91] B. N. BERSHAD and M. J. ZEKAUSKAS, Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors, Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[BZS92] B. N. BERSHAD, M. J. ZEKAUSKAS and W. A. SAWDON, The Midway Distributed Shared Memory System, *Compcon*, 1992.

[BiJ87] K. BIRMAN and T. JOSEPH, Exploiting Virtual Synchrony in Distributed Systems, *Proc. 11th ACM Symp. Operating Systems and Principles*, November 1987, 123-138.

[BNR89] R. BISIANI, A. NOWATZYK and M. RAVISHANKAR, Coherent Shared Memory on a Distributed Memory Machine, *Proc. Intl. Conf. on Parallel Processing*, August 1989, I-133-141.

[BiR90] R. BISIANI and M. RAVISHANKAR, PLUS: A Distributed Shared-Memory System, *Proc. 17th Annual Intl. Symp. on Computer Architecture*, May 1990, 115-124.

[Bit92] P. BITAR, The Weakest Memory-Access Order, *Journal of Parallel and Distributed Computing 15*, 4 (August 1992), 305-331.

[BMW85] W. C. BRANTLEY, K. P. MCAULIFFE and J. WEISS, RP3 Process-Memory Element, *International Conference on Parallel Processing*, August 1985, 772-781.

[CRA82] CRAY, Cray X-MP Series Mainframe Reference Manual, Publication Number HR-0032, CRAY Research Inc., November 1982.

[Car91] M. CARLTON, Implementation Issues for Multiprocessor Consistency Models, *Presentation in the Second Workshop on Scalable Shared-Memory Multiprocessors*, Toronto, May 1991.

[CBZ91] J. B. CARTER, J. K. BENNETT and W. ZWAENEPOEL, Implementation and Performance of Munin, *Proc. Symposium on Operating System Principles*, October 1991, 152-164.

[CeF78] L. M. CENSIER and P. FEAUTRIER, A New Solution to Coherence Problems in Multicache Systems, *IEEE Trans. on Computers C-27*, 12 (December 1978), 1112-1118.

[ChV88] J. CHEONG and A. V. VEIDENBAUM, A Cache Coherence Scheme With Fast Selective Invalidation, *Proceedings of the 15th Annual International Symposium on Computer Architecture 16*, 2 (June 1988), 299-307.

[Che90] H. CHEONG, Compiler-Directed Cache Coherence Strategies for Large-Scale Shared-Memory Multiprocessor Systems, Ph.D. Thesis, Dept. of Electrical Engineering, University of Illinois, Urbana-Champaign, 1990.

[ChM91] J. CHOI and S. L. MIN, Race Frontier: Reproducing Data Races in Parallel Program Debugging, *Proc. 3rd ACM Symp. on Principles and Practice of Parallel Programming*, April 1991.

[Col84-92] W. W. COLLIER, *Reasoning about Parallel Architectures*, Prentice-Hall, Englewood Cliffs, New Jersey, 1992. Parts of this work originally appeared in the following technical reports by W.W. Collier: ‘‘Architectures for Systems of Parallel Processes’’ (27 January 1984, Technical Report 00.3253), ‘‘Write Atomicity in Distributed Systems’’ (19 October 1984, Technical Report 00.3304), ‘‘Reasoning about Parallel Architectures’’ (1985), IBM Corp., Poughkeepsie, N.Y.

[CKM88] R. Cytron, S. Karlovsky and K. P. McAuliffe, Automatic Management of Programmable Caches, *Proc. 1988 Intl. Conf. on Parallel Processing*, University Park PA, August 1988, II-229-238.

[DeM88] R. DeLeone and O. L. Mangasarian, Asynchronous Parallel Successive Overrelaxation for the Symmetric Linear Complementarity Problem, *Mathematical Programming 42*, 1988, 347-361.

[Dec81] Dec., VAX Architecture Handbook, 1981.

[DiS90] A. Dinning and E. Schonberg, An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection, *Proc. ACM SIGPLAN Notices Symp. on Principles and Practice of Parallel Programming*, March 1990, 1-10.

[DSB86] M. Dubois, C. Scheurich and F. A. Briggs, Memory Access Buffering in Multiprocessors, *Proc. 13th Annual Intl. Symp. on Computer Architecture 14*, 2 (June 1986), 434-442.

[DSB88] M. Dubois, C. Scheurich and F. A. Briggs, Synchronization, Coherence, and Event Ordering in Multiprocessors, *IEEE Computer 21*, 2 (February 1988), 9-21.

[DuS90] M. Dubois and C. Scheurich, Memory Access Dependencies in Shared-Memory Multiprocessors, *IEEE Trans. on Software Engineering SE-16*, 6 (June 1990), 660-673.

[DWB91] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee and Y. Chen, Delayed Consistency and Its Effects on the Miss Rate of Parallel Programs, *Supercomputing ’91*, November 1991, 197-206.

[DKC93] S. Dwarkadas, P. Keleher, A. L. Cox and W. Zwaenepoel, Evaluation of Release Consistent Software Distributed Shared Memory on Emerging Network Technology, *Proc. 20th Intl. Symp. on Computer Architecture*, 1993, 144-155.

[EmP88] P. A. Emrath and D. A. Padua, Automatic Detection of Nondeterminacy in Parallel Programs, *Proc. SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988, 89-99. Also appears in *SIGPLAN Notices 24(1)* (January 1989).

[FOW87] J. Ferrante, K. J. Ottenstein and J. D. Warren, The Program Dependence Graph and its Use in Optimization, *ACM Transactions on Programming Languages and Systems 9*, 3 (July 1987), 319-349.

[FoW78] S. Fortune and J. Wyllie, Parallelism in Random Access Machines, *Proc. Tenth ACM Symposium on Theory of Computing*, 1978, 114-118.

[FrS92] M. Franklin and G. S. Sohi, The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism, *Proc. 19th Annual Intl. Symp. on Computer Architecture*, May 1992, 58-67.

[Fri93] R. Friedman, Implementing Hybrid Consistency with High-Level Synchronization Operations, *Proc. Principles of Distributed Computing*, August 1993.

[GLL90] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy, Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. 17th Annual Intl. Symp. on Computer Architecture*, May 1990, 15-26.

[GGH91a] K. Gharachorloo, A. Gupta and J. Hennessy, Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors, *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991, 245-257.

[GGH91b] K. Gharachorloo, A. Gupta and J. Hennessy, Two Techniques to Enhance the Performance of Memory Consistency Models, *Proc. Intl. Conf. on Parallel Processing*, 1991, I355-I364.

[GhG91] K. Gharachorloo and P. B. Gibbons, Detecting Violations of Sequential Consistency, *Proc. Symp. on Parallel Algorithms and Architectures*, July 1991, 316-326.

[GAG92] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy and M. D. Hill, Programming for Different Memory Consistency Models, *Journal of Parallel and Distributed Computing 15*, 4 (August 1992), 399-407.

[GGH92] K. Gharachorloo, A. Gupta and J. Hennessy, Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors, *Proc. 19th Intl. Symp. on Computer Architecture*, 1992, 22-33.

[GAG93] K. GHARACHORLOO, S. V. ADVE, A. GUPTA, J. L. HENNESSY and M. 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, December 1993.

[GGH93] K. GHARACHORLOO, A. GUPTA and J. HENNESSY, Revision to ‘‘Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors’’, Technical Report CSL-TR-93-568, Stanford University, April 1993.

[GMG91] P. B. GIBBONS, M. MERRITT and K. GHARACHORLOO, Proving Sequential Consistency of High-Performance Shared Memories, *Proc. Symp. on Parallel Algorithms and Architectures*, July 1991, 292-303.

[GiM92] P. B. GIBBONS and M. MERRITT, Specifying Nonblocking Shared Memories, *Proc. Symp. on Parallel Algorithms and Architectures*, 1992, 306-315.

[Goo89] J. R. GOODMAN, Cache Consistency and Sequential Consistency, Technical Report #61, SCI Committee, March 1989. Also available as Computer Sciences Technical Report #1006, University of Wisconsin, Madison, February 1991.

[Goo93] J. R. GOODMAN, Private Communication, 1993.

[GGK83] A. GOTTLIEB, R. GRISHMAN, C. P. KRUSKAL, K. P. MCAULIFFE, L. RUDOLPH and M. SNIR, The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer, *IEEE Trans. on Computers*, February 1983, 175-189.

[Gup89] R. GUPTA, The Fuzzy Barrier: A Mechanism for High Speed Synchronization of Processors, *Proc. Third Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, April 1989, 54-63.

[GHG91] A. GUPTA, J. HENNESSY, K. GHARACHORLOO, T. MOWRY and W. WEBER, Comparative Evaluation of Latency Reducing and Tolerating Techniques, *Proc. 18th Ann. Intl. Symp. on Computer Architecture,* May 1991, 254-263.

[Gus92] D. B. GUSTAVSON, The Scalable Coherent Interface and Related Standards Projects, *IEEE Micro 12*, 2 (February 1992), 10-22.

[HeW90] M. P. HERLIHY and J. M. WING, Linearizability: A Correctness Condition for Concurrent Objects, *ACM Trans. on Programming Languages and Systems 12*, 3 (July 1990), 463-492.

[HLR92] M. D. HILL, J. R. LARUS, S. K. REINHARDT and D. A. WOOD, Cooperative Shared Memory: Software and Hardware Support for Scalable Multiprocessors, *Proc. 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 1992, 262-273.

[HKT92] S. HIRANANDANI, K. KENNEDY and C. TSENG, Compiling Fortran D for MIMD Distributed-Memory Machines, *Communications of the ACM 35*, 8 (August 1992), 66-80.

[HKM90] R. HOOD, K. KENNEDY and J. MELLOR-CRUMMEY, Parallel Program Debugging with On-the-fly Anomaly Detection, *Supercomputing ’90*, November 1990, 74-81.

[Hor92] J. HORNING, Private Communication, *Digital Systems Research Center*, April 1992.

[HuA90] P. W. HUTTO and M. AHAMAD, Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories, *Proc. 10th Intl. Conf. on Distributed Computing Systems*, 1990, 302-311.

[IBM83] IBM, IBM System/370 Principles of Operation, Publication Number GA22-7000-9, File Number S370-01, May 1983.

[Jef85] D. R. JEFFERSON, Virtual Time, *ACM Trans. on Programming Languages and Systems 7,* 3 (July 1985), 404-425.

[KCZ92] P. KELEHER, A. L. COX and W. ZWAENEPOEL, Lazy Consistency for Software Distributed Shared Memory, *Proc. 19th Intl. Symp. on Computer Architecture*, 1992, 13-21.

[Kni86] T. KNIGHT, An Architecture for Mostly Functional Languages, *Proc. ACM Conf. on LISP and Functional Programming*, 1986, 105-112.

[KNA93] P. KOHLI, G. NEIGER and M. AHAMAD, A Characterization of Scalable Shared Memories, GIT-CC-93/04, Georgia Institute of Technology, January 1993.

[Kro81] D. KROFT, Lockup-Free Instruction Fetch/Prefetch Cache Organization, *Proc. Eighth Symp. on Computer Architecture*, May 1981, 81-87.

[Lam78] L. LAMPORT, Time, Clocks, and the Ordering of Events in a Distributed System, *Communications of the ACM 21*, 7 (July 1978), 558-565.

[Lam79] L. LAMPORT, How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, *IEEE Trans. on Computers C-28*, 9 (September 1979), 690-691.

[LHH91] A. LANDIN, E. HAGERSTEN and S. HARIDI, Race-Free Interconnection Networks and Multiprocessor Consistency, *Proc. 18th Annual Intl. Symp. on Computer Architecture*, May 1991, 106-115.

[LCW93] J. R. LARUS, S. CHANDRA and D. A. WOOD, CICO: A Practical Shared-Memory Programming Performance Model, Computer Sciences Technical Report #1171, University of Wisconsin-Madison, August 1993. Presented at the Workshop on Portability and Performance for Parallel Processing, July, 1993. To appear: Ferrante & Hey eds., Portability and Performance for Parallel Processors, John Wiley & Sons, Ltd..

[LeM92] T. J. LEBLANC and E. P. MARKATOS, Shared Memory vs. Message Passing in Shared-Memory Multiprocessors, *4th IEEE Symp. Parallel and Distributed Processing*, December 1992.

[LeR91] J. LEE and U. RAMACHANDRAN, Architectural Primitives for a Scalable Shared-Memory Multiprocessor, *Proc. 3rd Annual ACM Symp. Parallel Algorithms and Architectures*, July 1991, 103-114.

[LLG90] D. LENOSKI, J. LAUDON, K. GHARACHORLOO, A. GUPTA and J. HENNESSY, The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor, *Proc. 17th Annual Symposium on Computer Architecture, Computer Architecture News 18*, 2 (June 1990), 148-159, ACM.

[Li88] K. LI, IVY: A Shared Virtual Memory System for Parallel Computing, *Proc. Intl. Conf. on Parallel Processing*, 1988, 94-101.

[Li89] K. LI and P. HUDAK, Memory Coherence in Shared Virtual Memory Systems, *ACM Trans. on Computer Systems 7*, 4 (November 1989), 321-359.

[LiS88] R. J. LIPTON and J. S. SANDBERG, PRAM: A Scalable Shared Memory, Technical Report CS-Tech. Rep.-180-88, Princeton University, September 1988.

[LyT88] N. A. LYNCH and M. R. TUTTLE, An Introduction to Input/Output Automata, Technical Report MIT/LCS/TM-373, MIT Laboratory for Computer Science, November 1988.

[MeS91] J. M. MELLOR-CRUMMEY and M. L. SCOTT, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Transactions on Computer Systems*, February 1991, 21-65.

[MPC89] S. MIDKIFF, D. PADUA and R. CYTRON, Compiling Programs with User Parallelism, *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[MoG91] T. MOWRY and A. GUPTA, Tolerating Latency Through Software-Controlled Prefetching, *Journal of Parallel and Distributed Computing*, June 1991, 87-106.

[Mul89] S. MULLENDER, *Distributed Systems, Chapter 15*, Addison-Wesley, ACM Press, 1989.

[NaT92] B. NARENDRAN and P. TIWARI, Polynomial Root-Finding: Analysis and Computational Investigation of a Parallel Algorithm, *Proceedings of the 4th Annual Symposium on Parallel Algorithms and Architectures*, 1992, 178-187.

[NeM90] R. H. B. NETZER and B. P. MILLER, Detecting Data Races in Parallel Program Executions, *Research Monographs in Parallel and Distributed Computing, MIT Press, 1991.* Also available as *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing, August 1990.*

[NeM91] R. H. B. NETZER and B. P. MILLER, Improving the Accuracy of Data Race Detection, *Proc. 3rd ACM Symp. on Principles and Practice of Parallel Programming*, April 1991.

[Pap86] C. PAPADIMITRIOU, *The Theory of Database Concurrency Control*, Computer Science Press, Rockville, Maryland 20850, 1986.

[Pat83-86] J. PATEL, Mulitprocessor Cache Memories, Seminar at Texas Instruments Research Labs (Dallas, Dec. 1983), Intel (Aloha, Oregon, April 1984), Digital Equipment (Hudson, Mass., June 1984), IBM (Yorktown, Oct. 1984), IBM (Poughkeepsie, Aug. 1986).

[PeL92a] K. PETERSEN and K. LI, An Evaluation of Multiprocessor Cache Coherence Based On Virtual Memory Support, Technical Report Tech. Rep.-401-92, Princeton University, December 1992.

[PeL92b] K. PETERSEN and K. LI, Cache Coherence for Shared Memory Multiprocessors Based On Virtual Memory Support, Technical Report Tech. Rep.-400-92, Princeton University, December 1992.

[PBG85] G. F. PFISTER, W. C. BRANTLEY, D. A. GEORGE, S. L. HARVEY, W. J. KLEINFELDER, K. P. MCAULIFFE, E. A. MELTON, V. A. NORTON and J. WEISS, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, *International Conference on Parallel Processing*, August 1985, 764-771.

[ReK79] D. P. REED and R. K. KANODIA, Synchronization with Eventcounts and Sequencers, *Communications of the ACM 22*, 2 (February 1979), 115-123.

[ReT86] R. RETTBERG and R. THOMAS, Contention is No Obstacle to Shared-memory Multiprocessors, *Communications of the ACM 29*, 12 (December 1986), .

[RuS84] L. RUDOLPH and Z. SEGALL, Dynamic Decentralized Cache Schemes for MIMD Parallel Processors, *Proc. Eleventh International Symposium on Computer Architecture*, June 1984, 340-347.

[SUN91] SUN, The SPARC Architecture Manual, Sun Microsystems Inc., No. 800-199-12, Version 8, January 1991.

[SUN93] SUN, The SPARC Architecture Manual, Sun Microsystems Inc., Version 9, 1993.

[San90] J. SANDBERG, Design of the PRAM Network, Technical Report CS-Tech. Rep.-254-90, Princeton University, April 1990.

[ScD87] C. SCHEURICH and M. DUBOIS, Correct Memory Operation of Cache-Based Multiprocessors, *Proc. Fourteenth Annual Intl. Symp. on Computer Architecture*, Pittsburgh, PA, June 1987, 234-243.

[ScD88] C. SCHEURICH and M. DUBOIS, Concurrent Miss Resolution in Multiprocessor Caches, *Proc. of the 1988 Intl. Conf. on Parallel Processing*, University Park PA, August, 1988, I-118-125.

[Sch89] C. E. SCHEURICH, Access Ordering and Coherence in Shared Memory Multiprocessors, Ph.D. Thesis, Department of Computer Engineering, Technical Report CENG 89-19, University of Southern California, May 1989.

[ShR91] G. SHAH and U. RAMACHANDRAN, Towards Exploiting the Architectural Features of Beehive, Technical Report GIT-CC-91/51, College of Computing, Georgia Institute of Technology, November 1991.

[ShS88] D. SHASHA and M. SNIR, Efficient and Correct Execution of Parallel Programs that Share Memory, *ACM Trans. on Programming Languages and Systems 10*, 2 (April 1988), 282-312.

[SFC91] P. S. SINDHU, J. FRAILONG and M. CEKLEOV, Formal Specification of Memory Models, Technical Report CSL-91-11 [P91-00112], Xerox Corporation, December 1991.

[SWG92] J. P. SINGH, W. WEBER and A. GUPTA, SPLASH: Stanford Parallel Applications for Shared-Memory, *Computer Architecture News 20*, 1 (March 1992), 5-44.

[Sit92] R. SITES, EDITOR, Alpha Architecture Reference Manual, 1992.

[Smi82] B. SMITH, Architecture and Applications of the HEP Multiprocessor Computer System, *Proc. of the Int. Soc. for Opt. Engr.*, 1982, 241-248.

[SwS86] P. SWEAZEY and A. J. SMITH, A Class of Compatible Cache Consistency Protocols and their Support by the IEEE Futurebus, *Proc. Thirteenth International Symposium on Computer Architecture*, Tokyo, Japan, June 1986, 414-423.

[Tay83a] R. N. Taylor, Complexity of Analyzing the Synchronization Structure of Concurrent Programs, *Acta Informatica 19*(1983), 57-84.

[Tay83b] R. N. Taylor, A General-Purpose Algorithm for Analyzing Concurrent Programs, *Communications of the ACM 26*, 5 (May 1983), 362-376.

[TiK88] P. Tinker and M. Katz, Parallel Execution of Sequential Scheme with ParaTran, *Proc. ACM Conf. on LISP and Functional Programming*, July 1988, 28-39.

[ToH90] J. Torellas and J. Hennessy, Estimating the Performance Advantages of Relaxing Consistency in a Shared-Memory Multiprocessor, *Intl. Conference on Parallel Processing*, 1990.

[WeG89] W. Weber and A. Gupta, Exploring the Benefits of Multiple Hardware Contexts in a Multiprocessor Architecture: Preliminary Results, *The 16th. Annual Intl. Symp. on Computer Architecture*, May 1989, 273-280.

[WiL92] A. W. Wilson and R. P. LaRowe, Hiding Shared Memory Reference Latency on the Galactica Net Distributed Shared Memory Architecture, *Journal of Parallel and Distributed Computing 15*, 4 (August 1992), 351-367.

[Zuc91] R. N. Zucker, A Study of Weak Consistency Models, Dissertation Proposal, University of Washington, 1991.

[Zuc92] R. N. Zucker, Relaxed Consistency And Synchronization in Parallel Processors, Ph.D. Thesis, Technical Report 92-12-05, University of Washington, December 1992.

[ZuB92] R. N. Zucker and J. Baer, A Performance Study of Memory Consistency Models, *19th Intl. Symp. on Computer Architecture*, 1992, 2-12.

Appendix A: Equivalence of Definitions of Race for Data-Race-Free-0 Programs

This appendix proves that using any of the definitions of a data race in figure 4.2 or in definition 4.4 results in the same set of data-race-free-0 programs. Alternatively, we need to prove the following for any sequentially consistent execution, E_s, of a program *Prog*: if an operation in E_s needs to be distinguished as a synchronization operation by some definition above, then it also needs to be distinguished as a synchronization operation by all of the above definitions. The following discusses each of the five alternative definitions.

Alternative 1 (Definition 4.7): Clearly, a data race according to alternative 1 is also a data race according to all other definitions. Therefore, all operations that need to be distinguished as synchronization by alternative 1 also need to be distinguished as synchronization by other definitions.

Alternative 3 (Definition 4.12): Consider a pair of conflicting operations, X and Y, that form a race with alternative 3; i.e., X and Y are not ordered by the program/conflict graph. Then alternative 3 requires that these operations be distinguished as synchronization. We show that these operations need to be distinguished as synchronization by other definitions as well as follows. Consider the operations in E_s that are ordered before Y in the program/conflict graph. Consider a modification of the execution order of E_s where the above operations are moved to just before X, retaining their original relative order. Consider the resulting order until before X. This resulting order is still consistent with program order, and the last conflicting write before any read is still the same as with the original execution order. Now consider the resulting order appended with X and Y (in any order) and make X and Y return the value of the last conflicting write before them. (If X or Y is part of a read-modify-write, then include both the read and write above.) It follows that the resulting order is the prefix of an execution order of some sequentially consistent execution where the operations of the processors of X and Y preceding X and Y are the same as for E_s, and X and Y access the same addresses as before. Further, if X and Y are not parts of read-modify-writes, then they occur consecutively in the resulting order. If they are parts of read-modify-writes, then one of the arrangements (i.e., either X before Y or Y before X) has X and Y consecutive. Consider any sequentially consistent execution with the above prefix. From the assumptions of how operations are distinguished (Section 5.1.3 and Condition 7.18) and by the above observations, it follows that X and Y form a race by alternative 1 in the new execution and are distinguished in the same way as in E_s. It therefore follows that X and Y should be distinguished as synchronization according to all definitions in the new execution, and therefore in E_s. Thus, any operation that needs to be distinguished as synchronization by alternative 3 needs to be distinguished similarly by all other definitions.

Definition 4.4: This definition requires two conflicting operations, X and Y, not ordered by $\xrightarrow{hb0}$ to be distinguished as synchronization. Suppose X and Y are not ordered by any path in the program/conflict graph described by alternative 3, where the path has at least one program order arc. Then X and Y need to be distinguished as synchronization by alternative 3, and therefore (by the above case), by all other definitions. Suppose X and Y are ordered by a path in the program/conflict graph of alternative 3, where the path has at least one program order arc. Then consider a longest such path. Between any operations of a $\xrightarrow{co}$ arc on this path, there is no path of the above type. Therefore, alternative 3 requires the operations on the $\xrightarrow{co}$ arcs of this path to be distinguished as synchronization. But then the above path is a $\xrightarrow{hb0}$ path from X to Y, a contradiction.

Alternative 4 (Definition 4.14): The proof for alternative 4 is similar to that for alternative 3.

Alternative 2 (Definition 4.9): The proof for alternative 2 can be derived from the result for alternative 4 similar to the way the proof for definition 4.4 is derived from the result for alternative 2.

Appendix B: Modified Uniprocessor Correctness Condition

We assume the notion of a control flow graph and that of a loop defined for a control flow graph as in [ASU86]. Specifically, we only consider loops where an instruction of the loop cannot fork off multiple instruction instances in an execution; i.e., each set of next instructions of an instruction in the loop contains only one instruction (note that this does not prohibit branch instructions which have more than one *sets* of next instructions). We use the following definitions.

An *instance* of *loop* L in an execution is a sequence of instruction instances such that all instruction instances are from instructions in loop L, the first instruction instance in the sequence is the next instruction instance of an instruction instance not from loop L (or is an initial instance that is not the next instance of any instance), at least one next instruction instance of all but the last instruction instance in the sequence is from an instruction in loop L.

An instance of a loop is *incomplete* or *does not terminate* in an execution iff the number of instruction instances in the loop instance is infinite.

We modify the definition of an instruction instance to include the following. For every incomplete loop instance L in execution *E*, one instruction instance of L *may* specify two (sets of) next instructions, where one next instruction is from the loop and the other is not from the loop.

We modify the definition of program order on instruction instances to also include the following. For instruction instances i_1 and i_2 in an execution, $i_1 \xrightarrow{po} i_2$ if i_1 is from an incomplete loop instance L, and i_2 is an additional next instance of an instruction instance from L as allowed by the above modification to the definition of instruction instances.

The modified uniprocessor correctness condition assumes the above modifications to the definition of an instruction instance and program order and is as follows. (It differs from the original condition in points (2) and (3).)

> **Condition B.1:** *Uniprocessor correctness condition:* The set of instruction instances *I* of an execution must obey the following conditions.
>
> (1) For every initial instruction of a process, exactly one instance of the instruction must be present in *I* such that it is not the next instruction instance of another instruction instance. Such an instruction instance is called an initial instruction instance.
>
> (2) An instruction instance is present in *I* only if it is the next instruction instance of another instruction instance, or it is an initial instruction instance (as specified by (1) above).
>
> (3) If an instruction instance is present in *I* and does not follow an instruction instance from an incomplete loop instance, then its next instruction instance must be present in *I*.
>
> (4) The program order relation is acyclic. Specifically, an instruction instance is the next instruction instance of at most one instruction instance, an initial instruction instance cannot be a next instruction instance of any instruction instance, and a merge instruction instance for an instruction instance *i* must be after *i* by program order.
>
> (5) If *i* is an initial instruction instance, then the value returned by *i* for its read of a state variable of its process is the initial value of that state variable as specified by the program. If *i* is not an initial instruction instance, then the value returned by *i* for its read of a state variable of its process is the value specified by the last instruction instance ordered before *i* by program order that modified that state variable.[38]

[38] Note that part (5) implies that an instruction instance *i* following an incomplete loop instance L cannot be in the execution if an infinite number of the instructions instances in L could access the processor state accessed by *i* in a conflicting manner. The execution order condition imposes a similar condition for memory operations.

(6) The components other than the values of the state variables and memory locations read by an instruction instance *i* are determined by applying the functions specified by the instruction of *i* to the values of the state variables read by *i* and the values returned by the various reads of *i*.

(7) The value read from an external input interface by an instruction instance *i* must be a value written by an external observer or an initial value.

Appendix C: Correctness of Condition 7.12 for Sequential Consistency

This appendix proves the correctness of Condition 7.12 and also discusses an aggressive extension for the definition of a critical set. For reference, we repeat Condition 7.12 below, and also give the definition of a critical set incorporating non-atomic writes. Recall that *last* and *after* used below refer to the ordering due to execution order.

> **Definition C.1:** A *critical set* for an execution is a set of ordering paths that obey the following properties. Let X and Y be any two consecutive conflicting operations such that there is an ordering path or race path from X to Y. Ignore all unessential operations.
>
> (1) If either X is a read or Y is not a self-ordered read, and if there is an ordering path from X to Y, then one such path is in the critical set.
>
> (2) If Y is an exit read from a synchronization loop that is not self-ordered, Y is issued by processor P_i, and if there is no ordering path from X to Y, then let $W(i)$ be the last write sub-operation (including the hypothetical initial write) before $X(i)$ that conflicts with $Y(i)$ and writes an exit value of Y. If $W(i)$ exists, then let $W'(i)$ be a write after $W(i)$ such that there is an ordering path from W' to Y that ends in a program order arc. If W' exists, then one ordering path from the last such W' to Y that ends in a program order arc must be in the critical set.
>
> (3) If Y is a self-ordered read, then let W and W' be as defined in (2) above. If W' exits, then one ordering path from *any* W' to Y that ends in a program order arc must be in the critical set.
>
> For every execution, we consider one specific critical set, and call the paths in that set as *critical paths*.

> **Condition 7.12:** *System-Centric Specification of Sequential Consistency:* The execution should have a critical set such that if an ordering path from X to Y is in the critical set, then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i (assuming finite speculation, write termination, and loop coherence).

Proof:

Consider an execution E that obeys the above specification. Ignore all unessential operations in E. Consider two consecutive conflicting operations X and Y that have an ordering path between them. The following shows $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i in E.

Case 1: X is a read or Y is not a self-ordered read.

X and Y satisfy part (1) in definition 7.11. Therefore, an ordering path from X to Y is critical. Therefore, $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i.

Case 2: X is a write and Y is a self-ordered read.

X and Y satisfy part (3) in definition 7.11. Let Y be issued by processor P_i. Suppose for a contradiction that $Y(i) \xrightarrow{\text{xo}} X(i)$. If an ordering path corresponding to X,Y is chosen critical, then let W' be the write such that an ordering path from W' to Y is chosen critical. Then $W'(i) \xrightarrow{\text{xo}} Y(i)$. Therefore, W' must be different from X. $W'(i)$ is after (by $\xrightarrow{\text{xo}}$) the last conflicting write before $X(i)$ that wrote an exit value of Y. Therefore, $W'(i)$ and all other conflicting write sub-operations between $W'(i)$ and $Y(i)$ (by execution order) must write non-exit values. Therefore, $Y(i)$ is unessential, a contradiction. If no W' is chosen, then there is no write before $Y(i)$ that writes an exit value. Therefore, again Y is unessential, a contradiction.

We have proved so far that (after ignoring unessentials), if there is an ordering path from X to Y and X and Y are consecutive conflicting operations, then $X(i) \xrightarrow{\text{xo}} Y(i)$. It follows immediately that if X and Y are any conflicting operations and there is an ordering path from X to Y, then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i. Thus, if E does not have any unessentials, then it obeys Condition 7.6 and so the above specification is correct. If E

has unessentials and all its synchronization loops terminate, then by the loop coherence assumption, an execution that is the same as E but without the unessentials of E has the same result as E. This execution obeys Condition 7.6 and therefore again the above specification is correct. Thus, the only remaining case is where E has synchronization loops that do not terminate.

Let processor P_i have a synchronization loop that does not terminate in E. By the finite speculation assumption, P_i does not execute any operations or instructions following the loop operations and instructions. Consider the $\xrightarrow{\text{xo}}$ of E without the unessential operations. Any prefix of this $\xrightarrow{\text{xo}}$ must be a prefix of a sequentially consistent execution of program *Prog*. We know that a synchronization loop always terminates in every sequentially consistent execution of the program. Therefore, it follows that at some point in the above $\xrightarrow{\text{xo}}$, the shared-memory locations should have values that satisfy the predicate of at least one of the synchronization loops that did not terminate and these values should not change in the rest of the $\xrightarrow{\text{xo}}$ (otherwise, we can construct a sequentially consistent execution where all the above loops would not terminate). It follows that even in E, the reads of this synchronization loop will see the exit values and will terminate the loop (assuming write termination). □

We next discuss an aggressive extension of the definition of a critical set. The extension applies to part (3) of the definition when Y is an exit read that is part of a read-modify-write, and is based on the following observation. If the first write W_2 after W mentioned in part (3) is also part of a read-modify-write from a synchronization loop that writes a non-exit value of Y and if the value written by Y's read-modify-write is a non-exit value of W_2's loop, then Y cannot execute just before W_2 because that makes W_2 unessential. Therefore, making an ordering path from W to Y critical is also sufficient. Furthermore, if the conflicting writes before W also write non-exit values, then making an ordering path from those writes to Y critical is also sufficient. Extending this observation to all writes that obey the above pattern leads to the following formalization for a possible critical path when Y is a read-modify-write from a self-ordered loop. This observation is exploited by the PLpc models. The following assumes atomic writes as in definition 7.11; the extension to non-atomic writes is analogous to definition C.1 given at the beginning of this appendix.

Definition C.2: *Extension to definition 7.11:*

Parts (1) and (2) of definition 7.11 remain unchanged and part (3) need be applied only when Y is not part of a read-modify-write. When Y is part of a read-modify-write, either part (3) or the following can be applied.

Let W be the last write (including the hypothetical initial write) before X such that W conflicts with Y, W writes an exit value of Y, and the following is true. If W_1 (conflicting with X) is between W and X and writes an exit value of Y, then the first conflicting write, W_2, after W_1 writes a non-exit value of Y, W_2 is from a synchronization loop, and the write of Y's read-modify-write writes a non-exit value for W_2's loop. If W exists and if there is an ordering path from any write after W to Y that ends in a program order arc, then one such path is in the critical set.

The proof given above extends easily to incorporate the extension as well. Specifically, for the new case, if $Y(i) \xrightarrow{\text{xo}} X(i)$, then the first conflicting essential write, W_2, after $Y(i)$ must be from a synchronization loop and Y's write writes a non-exit value for that loop. Thus, W_2 must be unessential, a contradiction.

The proof of the current low-level control requirement (Condition 7.21) in Appendix E considers the possibility of the above types of critical paths; however, it makes two assumptions. The first is that the exit read of the loop of W_2 mentioned above forms a race in some sequentially consistent execution without unessential operations. Without this assumption, the proof of Appendix E is valid with the above extension only if the following additional condition on the properties of the control relation is imposed. (Below, assume E_s is a sequentially consistent execution without unessential operations.) The additional condition is: an exit read R_2 of the loop of W_2 mentioned above should control any operation O that follows it such that (a) $R_2 \xrightarrow{\text{vpo}} O$ in some E_s, or (b) O is an exit read from a self-ordered loop, or (c) O forms a race in some E_s and is an exit read of a synchronization loop.

The second assumption in the proof in Appendix E is that property (a) of the control relation (definition 7.19(a)) should hold if all control reads of O in E are in E_s, and all such reads return the same value in E and E_s

with the possible exception of reads which have critical paths of the above type and such that *O* is not in the loop instance of the exception reads. This assumption is reasonable because the properties of a synchronization loop ensure that the value returned by a read of the above type cannot affect whether a later operation following this loop instance will be executed in E_s, or what the exit values of an exit read from another loop instance will be, or what the write of an exit read's read-modify-write from another loop instance will write.

Appendix D: A Constructive Form of the Control Relation

This appendix constructively defines a relation for an SCNF model that obeys the properties for the control relation in Definition 7.19. The underlying concepts for this work were first developed for the data-race-free-1 and PLpc models [AdH92, AGG93]; the following is a generalization of those concepts for all SCNF models expressed in our framework. The formalization and description of the relation below are similar to that of the corresponding relation developed for PLpc [AGG93] (and later used for other models [GAG93]). The original relation is called the reach ($\xrightarrow{\text{rch}}$) relation [AGG93] and we continue to use the same name here.

To obey property (g) of the control relation, the reach relation assumes that in any execution, no instruction instance of a synchronization loop can change the way an exit read is distinguished for the purposes of determining whether it is on a valid path. This additional assumption is only required if the presence of a preceding instruction can be used to determine whether an operation is on a valid path (as with MB of Alpha), and this preceding instruction and the operation are in different basic blocks within the loop which are not guaranteed to execute together in an execution.

Section D.1 defines the reach relation and Section D.2 proves that it obeys the properties of the control relation given in Definition 7.19.

D.1 The Reach Relation

We assume the internal state of a process that can be read and written by an instruction instance consists of registers and private memory of the process. The register read set of an instruction instance consists of the registers and private memory locations whose values the instruction instance reads, and the register write set consists of the registers and private memory locations that the instruction instance updates. For simplicity, we assume that an instruction instance accesses at most one shared-memory location, and the shared-memory operations of any instruction instance that accesses shared-memory consist of either exactly one read or exactly one write or exactly one read-modify-write. This assumption is obeyed by most RISC architectures. The assumption is for simplicity only; the relation below can be easily extended for a more complicated instruction set.

We first define the notion of local data/distinction dependence ($\xrightarrow{\text{dd}}$) and branch dependence[39] ($\xrightarrow{\text{bd}}$) that will be used to develop the $\xrightarrow{\text{rch}}$ relation. For two instruction instances A and B in an execution, $A \xrightarrow{\text{dd}} B$ if B reads a value that A writes into its register write set. (The interaction due to shared-memory data dependence is handled later.) Also, $A \xrightarrow{\text{dd}} B$ if the presence of A is used by the system to determine if B is on a valid path (as specified in part (c) of Condition 7.18).[40] The following definition of branch dependence assumes that the program order is a total order per process; it can be extended in a straightforward manner when this order is partial. The notion of branch dependence is borrowed from Ferrante et al. [FOW87], and is defined in terms of a control flow graph and dominators [ASU86] as follows. 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 is 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 *branch dependent* on instruction instance A if the following conditions hold: (i) $A \xrightarrow{\text{po}} B$ 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'.

[39] The branch dependence relation is called control dependence in [AGG93, GAG93]; we have already used the term *control* in other places and so use a different term here.

[40] This is the only way in which the data/distinction dependence relation differs from the corresponding relation (called data dependence) in [AGG93, GAG93]. The work in [AGG93, GAG93] currently does not allow distinguishing valid paths as specified by part (c) of Condition 7.18, but could easily do so.

To allow for possibly non-terminating sequentially consistent executions, we need to augment the control flow graph and the resulting $\xrightarrow{\text{bd}}$ relation with additional arcs. Informally, consider a loop in the program (as assumed by Appendix B) that does not terminate in some sequentially consistent execution. Consider an instruction instance *i* that is not in the loop and is program ordered after an instance *L* of the loop. Then we require $(\xrightarrow{\text{bd}})+$ to order *i* after the instances of the branch instructions of *L* that jump back into the loop and cause the loop to execute infinite iterations. More formally, let a branch instruction be one that specifies more than one set of next instructions (e.g., conditional branch instructions), or a single set of next instructions where one of the next instructions is not the immediately following instruction in the static sequence of instructions specified by the program (e.g., unconditional jumps). Let an instance of a branch instruction be said to change the program flow if one of the next instruction instances is not from an instruction immediately following the branch in the static sequence of instructions of the program. Then the condition for augmenting the control flow graph for a loop *L* that does not terminate in some sequentially consistent execution is as follows. Let *C* be a branch instruction that could be executed infinite times in some sequentially consistent execution E_s. Suppose an infinite number of successive instances of *C* change the control flow of the program in some E_s. Add an auxiliary edge from every such instruction *C* to the EXIT node. This ensures that any such branch 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 $(\xrightarrow{\text{bd}})+$. The modification described above is not necessary if all sequentially consistent executions of the program will terminate, or if there are no memory operations or instructions that affect the output interface that are ordered after possibly non-terminating loops in the control flow graph.

We next use $\xrightarrow{\text{bd}}$ and $\xrightarrow{\text{dd}}$ to define two relations, the uniprocessor reach dependence ($\xrightarrow{\text{udep}}$) and the multiprocessor reach dependence ($\xrightarrow{\text{mdep}}$) below, and then define the $\xrightarrow{\text{rch}}$ relation in terms of these two relations. Below, E_s denotes a sequentially consistent execution. The following extends the $\xrightarrow{\text{vpo}}$ relation discussed in Chapter 7 to instruction instances in the obvious way.

Definition D.1: *Uniprocessor Reach Dependence:* Let *X* and *Y* be instruction instances in an execution *E* of program *Prog*. $X \xrightarrow{\text{udep}} Y$ in *E* iff $X \xrightarrow{\text{po}} Y$, and either

(a) $X \xrightarrow{\text{bd}} Y$ in *E*, or

(b) $X \xrightarrow{\text{dd}} Y$ in *E*, or

(c) *X* and *Y* are in another execution E_s of the program Prog, $X\ (\xrightarrow{\text{bd}})+\ Z \xrightarrow{\text{dd}} Y$ in E_s, and *Z* is not in *E*.

Definition D.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. $X \xrightarrow{\text{mdep}} Y$ in *E* iff any of the following is true.

(a) $X \xrightarrow{\text{po}} Y$ in *E* and *X* and *Y* are in another execution E_s of *Prog*, where $X\ \{\xrightarrow{\text{udep}}\}+\ Z \xrightarrow{\text{vpo}} Y$ in E_s, *Z* is an instruction instance that accesses shared-memory, for any $A \xrightarrow{\text{dd}} B$ constituting the $\{\xrightarrow{\text{udep}}\}+$ path from *X* to *Z* in E_s, *B* is also in *E*, and either *Z* is not in *E*, or *Z* is in *E* but accesses different addresses in *E* and E_s, or *Z* generates a write to the same address in *E* and E_s but with a different value in *E* and E_s.

(b) $X \xrightarrow{\text{po}} Y$ in *E*, $X\ \{\xrightarrow{\text{udep}} \cup \xrightarrow{\text{mdep}}\}+\ Z$ in *E*, and $Z \xrightarrow{\text{vpo}} Y$ in *E*.

(c) $X \xrightarrow{\text{po}} Y$ in *E*, *Y* generates an exit read of a synchronization loop, and $X\ \{\xrightarrow{\text{udep}}\}+\ Z$, where *Z* is a branch instruction in the loop.[41]

[41] Part (c) is not required if for every instance of a synchronization loop, the exit values of the loop instance and the values written by the exit writes of the loop instance are the same for every execution in which the loop instance occurs (see Section 7.5.2).

(d) $X \xrightarrow{\text{po}} Y$ in E and $X \{ \xrightarrow{\text{udep}} \cup \xrightarrow{\text{mdep}} \}+ Z \xrightarrow{\text{po}} Y$ in E where (i) X is from an instance L of a loop that does not terminate in E, L terminates in every E_s, X generates a read R in E where R forms a race in some sequentially consistent execution without unessential operations and R is not the last such read in L, and (ii) Z is a branch instruction in L that can change the program flow to the beginning of L, and (iii) Y generates an operation that is either an exit read of a self-ordered loop or that could form a race in some sequentially consistent execution without unessential operations.

(e) $X \xrightarrow{\text{po}} Y$ or X is the same as Y. Further, X generates read R which is either an exit read from a self-ordered loop or it is an exit read from a synchronization loop that could form a race in some sequentially consistent execution without unessential operations. Further, Y generates an operation O (different from R) such that either $R \xrightarrow{\text{vpo}} O$ in some E_s or O is an exit read of a self-ordered loop, or O could form a race in some sequentially consistent execution without unessential operations.

(f) $X \xrightarrow{\text{po}} Y$ in E or X is the same as Y in E, X generates read R and Y generates operation O (different from R) in E, R and O are in another execution E_s of *Prog*, $R \xrightarrow{\text{vpo}} O$ in E_s, and $W \xrightarrow{\text{co}} R$ is on a valid path in E_s only if W's sub-operation is the last essential conflicting sub-operation before R's sub-operation.

Definition D.3: *Reach Relation:* For two (possibly identical) instruction instances X and Y in E, $X \xrightarrow{\text{rch}} Y$ in E iff $X (\xrightarrow{\text{udep}} \cup \xrightarrow{\text{mdep}})+ Y$. For two different memory operations X' and Y' in E from instruction instances X and Y in E respectively, $X' \xrightarrow{\text{rch}} Y'$ in E iff $X' \xrightarrow{\text{po}} Y'$ and $X (\xrightarrow{\text{udep}} \cup \xrightarrow{\text{mdep}})+ Y$.

The reach relation is a transitive closure of the uniprocessor reach dependence ($\xrightarrow{\text{udep}}$) and the multiprocessor reach dependence ($\xrightarrow{\text{mdep}}$) relations. The uniprocessor component largely corresponds to uniprocessor data and branch dependence, while the multiprocessor component corresponds to dependences that are present due to the memory consistency model and due to aggressive speculative execution beyond unbounded loops. The uniprocessor dependence component also incorporates dependences that determine how an operation is distinguished for valid paths. The components that make up $\xrightarrow{\text{udep}}$ and $\xrightarrow{\text{mdep}}$ are defined for a given execution E. Both relations also require considering other sequentially consistent executions of the program, and determining if an instruction instance in one execution is in the other execution, as described in Section 7.5.2.

D.2 Proof of Correctness of the Reach Relation

We say an instruction instance in execution E_1 executes in execution E_2, or is in E_2, if it is matched up with some instruction instance in E_2 as explained in Section 7.5.2. We say an instruction instance in E_1 fully executes in E_2 if it executes in E_2, it accesses the same registers and memory locations in E_1 and E_2, it reads and writes the same values in the corresponding registers and memory locations in E_1 and E_2, and its operations are distinguished (for valid paths) similarly in E_1 and E_2. We prove that the reach relation obeys the various properties of the control relation in definition 7.19 below. Recall that E_s denotes a sequentially consistent execution.

Property (a): If for every read R that controls an operation O in E, R is in E_s and returns the same value in E and E_s, then O is in E_s. Further, if O is an exit read of a synchronization loop, then the exit value of O and the value written by the exit write corresponding to O (if any) are the same in E and E_s.

Proof:

Consider an operation O and a sequentially consistent execution E_s for which property (a) is not obeyed. Consider all $\{ \xrightarrow{\text{udep}} \}+$ paths to O in E, using only E_s as the other execution for the third part of the $\xrightarrow{\text{udep}}$ relation. We show that all instruction instances on these paths in E must be fully executed in E_s, and therefore, O executes in E_s.[42]

[42] Recall that appendix C mentions a slight modification to property (a) to incorporate the extended definition of critical paths (Definition C.2): control reads of O that use the extended definition may not return the same value in E and E_s (if

Suppose some instruction instance on such a path is not fully executed in E_s. Choose a first such instruction instance in program order. Let this instruction instance be i.

Case 1: i is not in E_s.

Let c be the last (by program order) instruction instance before i in E that executes in E_s. Then c must be a branch instruction, $c \; \{ \xrightarrow{\text{bd}} \}+ \; i$ in E, and c reads different values in E and E_s. But then $c \; \{ \xrightarrow{\text{udep}} \}+ \; i$, and so c cannot have different values since i was a first such instruction instance to have different values and c precedes i, a contradiction.

Case 2: i is in E_s.

There must be a difference in one of the values of i, or some instruction instance used to distinguish i is not fully executed. The difference in values cannot be the value returned by a shared-memory read since this read is before O by $\xrightarrow{\text{rch}}$ and so returns the same value in E and E_s. So either i read a different value written by a preceding instruction instance into its register write set, or a preceding instruction instance that distinguishes an operation of i is not fully executed. Let this preceding instruction instance be j_1 in E and j_2 in E_s.

Sub-case 2a: j_1 is not in E_s.

$j_1 \xrightarrow{\text{dd}} i$ in E and so j_1 is on a $\{ \xrightarrow{\text{udep}} \}+$ path to O in E and so must be in E_s, a contradiction.

Sub-case 2b: j_2 is not in E.

Then consider the first instruction instance k_2 before j_2 (by program order) in E_s that is in E. Call it k_1 in E. $k_2 \; \{ \xrightarrow{\text{bd}} \}+ \; j_2 \xrightarrow{\text{dd}} i$ in E_s. But then k_2 is on a $\{ \xrightarrow{\text{udep}} \}+$ path to O in E. So it cannot return a different value in E and E_s, a contradiction.

Sub-case 2c: j_1 and j_2 are both in E and E_s.

Then both j_1 and j_2 are on a $\{ \xrightarrow{\text{udep}} \}+$ path to O. They must both have the same values. Further, since they write the same register or private memory locations, they must be ordered by program order and so the same one must be the last one before i. So i must read the same value. □

It remains to prove that if all the reach reads of O return the same value, and if O is an exit read of a synchronization loop, then the exit values of O are the same and the value written by the exit write corresponding to O (if any) is the same. This follows directly from part (c) and since all reads that reach an exit write also reach the corresponding exit read.[43]

Property (b): If for every read R that controls an operation O in E, R is in E_s and returns the same value in E and E_s, and if $O' \xrightarrow{\text{vpo}} O$ in E_s, then $O' \xrightarrow{\text{po}} O$ in E.

Proof:

Consider an operation O, an operation O', and an E_s for which property (b) is not obeyed. Consider all paths to O' in E_s such that these paths followed by $O' \xrightarrow{\text{vpo}} O$ in E_s imply a $\xrightarrow{\text{mdep}}$ relation in E. Further, for any instruction instance i on these paths, also consider all instruction instances in E_s that are also in E and are before i by $\{ \xrightarrow{\text{udep}} \}+$ in E. We show that all instruction instances considered above in E_s must be fully executed in E. Thus, O' is in E and $O' \xrightarrow{\text{po}} O$ in E. Suppose not. Choose any first instruction instance by program order that violates the above. Let this instruction instance be i.

such a read and O are not in the same loop instance). This modification does not affect the proof of correctness of property (a) in this appendix because the above control reads are from synchronization loops and therefore cannot be on $\{ \xrightarrow{\text{udep}} \}+$ paths to O. Thus, such reads are not considered by this proof and so their value does not matter.

[43] The modification due to Appendix C does not affect this part of property (a) for reasons similar to that for the first part.

Case 1: i is not in E.

Let c be the last (by program order) instruction instance before i in E_s that is in E. Then c must be a branch instruction, c $\{ \xrightarrow{bd} \}+$ i in E_s, and c has a different value in E and E_s. But c must be one of the considered instruction instances; therefore, c cannot have different values in the two executions, a contradiction.

Case 2: i is in E.

Then one of the values of i must be different, or i is distinguished differently. Similar, to property (a), the difference in values cannot be due to a shared-memory read (because of part (a) of the definition of $\xrightarrow{mdep}$.) So either i must have read the wrong value from a register or private memory that was updated by a preceding instruction instance, or a preceding instruction instance that distinguishes i is not fully executed in E. Let this preceding instruction instance be j_1 in E and j_2 in E_s.

Sub-case 2a: j_2 is not in E.

Consider the last instruction instance before j_1 in E_s (by program order) that is in E. This instruction instance must be before j_1 by $\{ \xrightarrow{bd} \}+$, returns different values, but must be on the chosen paths. Then as before this is a contradiction.

Sub-case 2b: j_1 is not in E_s.

Then consider the first instruction instance k_1 before j_1 in E (by program order) that is in E_s. Call it k_2 in E_s. k_1 $\{ \xrightarrow{bd} \}+$ $j_1 \xrightarrow{dd} i$ in E. But then k_2 must be on one of the chosen paths. But k_1 and k_2 have different values, a contradiction.

Sub-case 2c: j_1 and j_2 are in E and E_s.

Then both j_1 and j_2 are on a path being considered and so must have the same values. Only one of them must be last and so i must have same values.

□

Property (c): If R controls an operation O' in E and if $O' \xrightarrow{vpo} O$ in E, then R controls O in E.

Proof:

Follows directly from part (b) of the definition of $\xrightarrow{mdep}$. □

Property (d): Consider an instance of a loop such that it does not terminate in E, it terminates in every E_s, and its termination in E_s depends on the value returned by one or more of its shared-memory reads R that could be involved in a race in some sequentially consistent execution without unessential operations. Then a read of the type R above that is not the last such read from its loop instance by program order in E must control any operation O in E such that O is not in R's loop instance and either O is an exit read of a self-ordered loop or O forms a race in some sequentially consistent execution without unessential operations.

Proof:

Follows directly from part (d) of the definition of $\xrightarrow{mdep}$. □

Property (e): R controls O in E if (i) R is either an exit read from a self-ordered loop or R is an exit read from a synchronization loop that forms a race in some sequentially consistent execution without unessential operations, and (ii) either $R \xrightarrow{vpo} O$ in some sequentially consistent execution, or O could form a race in some sequentially consistent execution without unessential operations, or O is an exit read of a self-ordered loop.

Proof:

Follows directly from part (e) of the definition of $\xrightarrow{mdep}$. □

Property (f): If W, R, and O are in E and E_s, $W \xrightarrow{co} R \xrightarrow{po} O$ is on a valid path in E_s, and the above $\xrightarrow{co}$ arc is on the valid path only if W is the last conflicting essential write before R (by execution order), then $R \xrightarrow{ctl} O$ in E.

Proof:

Follows directly from part (f) of the definition of $\xrightarrow{\text{mdep}}$. □

Property (g): If *R* is an exit read of a synchronization loop instance that does not terminate in *E*, *R* controls *W* in *E*, and *W* is not from *R*'s loop instance, then all exit reads in *R*'s loop instance that are from the same static operation as *R* control *W* in *E*.

Proof:

Follows by inspection and from the assumption stated in the beginning of this appendix. □

Appendix E: Correctness of Low-Level System-Centric Specification of Control Requirement

This appendix proves that the low-level system-centric specification of the control requirement for a generic SCNF model *M* (Condition 7.21) is correct. We prove this by proving the following theorem.

> **Theorem E.1:** An execution *E* of program *Prog* appears sequentially consistent if *Prog* is a valid program for model *M* and *E* obeys the low-level control requirement (Condition 7.21) for model *M*.

The proof proceeds by assuming that theorem E.1 is incorrect and shows a contradiction. Informally, the proof first shows that there must be a sequentially consistent execution (say E_s) that is ‘‘similar’’ to *E* in a ‘‘maximum number of ways.’’ The proof then shows a transformation on E_s that results in another sequentially consistent execution that is similar to *E* in ‘‘more ways’’ than E_s, implying a contradiction. The following first outlines the overall structure of the proof and gives some terminology.

Overall Structure of Proof

The proof consists of five main steps. Below, E_s denotes any sequentially consistent execution of program *Prog* without unessential operations.

Step E.1: We first define an execution E_m such that E_m has the same result as *E*, E_m obeys the low-level control requirement (Condition 7.21) for model *M*, and the execution order of E_m has some special properties.

Step E.2: For every E_s, we define certain types of well-behaved sub-operations in E_m, called ok and semi-ok sub-operations. In particular, these sub-operations are in E_s as well, and ok reads return the same values in E_m and E_s.

Step E.3: We show that if theorem E.1 is incorrect, then there must be a finite prefix (called *Prefix*) of the execution order of E_m such that for every E_s, there is a sub-operation in *Prefix* that is not ok for E_s.

Step E.4: Step E.3 implies that there must be an E_s for which the number of ok sub-operations in *Prefix* is greater than or equal to the corresponding number for any other sequentially consistent execution of *Prog*. We pick such an E_s (called E_1). We also show the existence of a sub-operation called $K(i)$ in *Prefix* that is not ok for E_1 but well-behaved in a certain way.

Step E.5: We define a transformation for E_1 that produces another sequentially consistent execution of *Prog* such that all sub-operations in *Prefix* that are ok for E_1 are ok for the new execution. In addition, $K(i)$ is also ok for the new execution. This is a contradiction since E_1 had the maximum number of ok sub-operations in *Prefix*.

Terminology and Notation

We continue to use the terms *precede* and *follow* to indicate program order, unless stated otherwise.

Other than the terms precede and follow, unless stated otherwise, all references to an ordering of sub-operations of an execution (e.g., first, before, after, etc.) pertain to the execution order of the execution. For a sequentially consistent execution, we also consider an execution order on operations, and the above comment applies to operations for such an execution as well.

Below, E_s denotes a sequentially consistent execution of program *Prog* without unessential operations. Further, a read of a read-modify-write in E_s is immediately followed by the sub-operations of its corresponding write in the execution order of E_s.

R and *W* denote read and write operations respectively. *O*, *X*, *Y*, and *Z* denote any operations. We use subscripts to differentiate between different operations. This appendix also sometimes overloads the above symbols to denote sub-operations. The specific usage is either clear from the context, or is applicable to both operations and sub-operations.

Valid paths refer to valid paths for model *M*.

The control requirement refers to the low-level specification of the control requirement (Condition 7.21) for model *M*.

Proof of Theorem E.1

Step E.1: Construction of execution E_m.

> **Definition :** *Control+ reads:* The *control+ reads* of an operation in an execution are the reads ordered before the operation by the ($\xrightarrow{\text{ctl}}$)+ relation of the execution. (The ($\xrightarrow{\text{ctl}}$)+ relation is specified in definition 7.19.)
>
> **Definition :** *Mate reads and writes:* Consider a read *R* and write *W* from a read-modify-write in an execution. Then *R* is the mate read of *W* and *W* is the mate write of *R* in the execution.
>
> **Definition :** $\xrightarrow{CTL}$ *relation and CTL reads:* In an execution, $R \xrightarrow{\text{CTL}} O$ or *R* is a CTL read of *O* if either *R* is a control+ read of *O*, or if *R* and *O* are from a synchronization loop and *R* is the mate read of *O*.
>
> **Definition :** $\xrightarrow{CTL*}$ *relation:* In an execution, $R(i) \xrightarrow{\text{CTL*}} O(j)$ if $R \xrightarrow{\text{CTL}} O$, *O* is a write, and either *R* returns the value of another processor's write or *R* is a mate read of *O*.

Recall that the $\xrightarrow{\text{co}}$ symbol is overloaded to define a relation on operations and sub-operations. The relation on sub-operations is as follows: $X(i) \xrightarrow{\text{co}} Y(i)$ if $X(i)$ and $Y(i)$ conflict and $X(i) \xrightarrow{\text{xo}} Y(i)$.

Lemma E.1.1: The relation ($\xrightarrow{\text{CTL*}} \cup \xrightarrow{\text{co}}$)+ on the sub-operations of an execution is acyclic.

Proof:

Suppose for a contradiction that there is a cycle in the relation ($\xrightarrow{\text{CTL*}} \cup \xrightarrow{\text{co}}$)+ on the sub-operations of some execution. Viewing the relation as a graph with edges due to $\xrightarrow{\text{CTL*}}$ and $\xrightarrow{\text{co}}$, consider a cycle involving the fewest number of sub-operations. There are two cases discussed below.

Case 1: For every $R_1(i) \xrightarrow{\text{CTL*}} W_1(j)$ on the cycle, R_1 ($\xrightarrow{\text{ctl}}$)+ W_1.

Let $W(i) \xrightarrow{\text{co}} R(i)$ be one of the $\xrightarrow{\text{co}}$ arcs on the cycle. Viewing the relation as a graph, the path from $R(i)$ to $W(i)$ is a control path. The critical set part of the control requirement ensures that $R(i) \xrightarrow{\text{xo}} W(i)$ for all *i*. Therefore, $W(i) \xrightarrow{\text{co}} R(i)$ cannot be on the cycle, a contradiction.

Case 2: There is one $R_1(i) \xrightarrow{\text{CTL*}} W_1(j)$ on the cycle such that R_1 is the mate read of W_1.

Then we have $W_2(i) \xrightarrow{\text{co}} R_1(i) \xrightarrow{\text{po}} W_1(j) \xrightarrow{\text{co}} R_2(j)$ on the cycle, where W_1, R_1, W_2, R_2 are all to the same location. By the assumptions for valid paths (condition 7.18), it follows that $R_1(i) \xrightarrow{\text{xo}} W_1(i)$. Therefore, $W_2(i) \xrightarrow{\text{xo}} W_1(i)$. By the loop coherence part of the control requirement, it follows that $W_2(j) \xrightarrow{\text{xo}} W_1(j)$. It therefore follows that $W_2(j) \xrightarrow{\text{xo}} R_2(j)$. Thus, $W_2(j) \xrightarrow{\text{co}} R_2(j)$, implying there is a cycle in the graph with fewer sub-operations than the chosen cycle, a contradiction. □

Lemma E.1.2: There is an execution $E_m{}'$ of program *Prog* such that:

(i) $E_m{}'$ obeys the control requirement for model *M*.

(ii) All components of $E_m{}'$ except its execution order (i.e., *I*, *O*, *V*, O_s in definition 5.2) are the same as *E*.

(iii) The execution order of $E_m{}'$ obeys the following properties: (a) if $R(i) \xrightarrow{\text{CTL*}} W(j)$ in $E_m{}'$, then $R(i) \xrightarrow{\text{xo}} W(j)$ in $E_m{}'$, and (b) conflicting sub-operations are ordered similarly by the execution orders of E and $E_m{}'$.

Proof:

Lemma E.1.1 ensures that there is a total order on the sub-operations of E that is consistent with the $\xrightarrow{\text{CTL*}}$ and $\xrightarrow{\text{co}}$ (on sub-operations) relations of E. The finite speculation part of the control requirement ensures that only a finite number of sub-operations can be ordered before another sub-operation by the $\xrightarrow{\text{CTL*}}$ relation. The execution order condition on an execution (Condition 5.3) ensures that only a finite number of sub-operations can be ordered before another sub-operation by the $\xrightarrow{\text{co}}$ relation. It follows that there must be a total order on the sub-operations of E that is consistent with the $\xrightarrow{\text{CTL*}}$ and $\xrightarrow{\text{co}}$ (on sub-operations) relations of E, and that orders only a finite number of sub-operations before any other sub-operation. It follows that the new total order qualifies to be an execution order of an execution $E_m{}'$ whose other components are the same as E. Further, this execution order obeys part (iii) of the lemma. To show that $E_m{}'$ obeys the rest of the lemma, we need to show that $E_m{}'$ obeys the control requirement.

The only constraints of the control requirement are on the order of conflicting sub-operations (i.e., the critical set part and the loop coherence parts), and on whether certain sub-operations and instruction instances can be present (i.e., the finite speculation and write termination parts). These aspects are the same for E and $E_m{}'$ and so $E_m{}'$ obeys the control requirement. □

Let E_m be the same as $E_m{}'$ (where $E_m{}'$ is as defined in lemma E.1.2) except that E_m does not contain any operations that are unessential in $E_m{}'$ and are from loop instances that terminate in $E_m{}'$. (Correspondingly, E_m does not contain sub-operations of the above operations or instruction instances that generate the above operations.) By the loop coherence part of the control requirement and the definition of unessential operations, it follows that E_m is an execution of program *Prog*. From lemma E.1.2, it follows that the result of E_m is the same as that of E. Therefore, E_m does not appear sequentially consistent. The rest of the proof uses the execution E_m to arrive at a contradiction.

Step E.2: Definition and properties of ok and semi-ok sub-operations and operations.

This step defines certain types of sub-operations and operations that we will use throughout the proof, called ok and semi-ok sub-operations and operations. We use the following definitions and observation.

Definition : For any E_s, define its *critical+ paths* as follows.

(1) The paths in one critical set of E_s that are also valid paths, and each such valid path (say between operations X and Y) is the longest valid path in E_s between X and Y. (Such a critical set exists since *Prog* is a valid program for model M.)

(2) All race paths in E_s between consecutive conflicting operations.

Two conflicting sub-operations are said to be ordered by a critical+ path if their corresponding operations are ordered by that critical+ path.

Definition : The *valid+ paths* of E_m are the following.

(1) The valid paths of E_m such that each such valid path (say between operations X and Y) is the longest valid path in E_s between X and Y and is also a valid path of E.

(2) All paths of the type $X \xrightarrow{\text{co}} Y$ in E_m where one of X or Y is a read operation.

Two conflicting sub-operations are said to be ordered by a valid+ path if their corresponding operations are ordered by that valid+ path.

Observation E.2.1: Suppose there is a valid+ path from X to Y in E_m. Further suppose that if the path is of type (1) above, then either X and Y are from the same processor, or X is a write, or X is a read that returns the value of its own processor's write in E. Then $X(i) \xrightarrow{\text{xo}} Y(i)$ for all i in E_m.

Proof: For paths of type (1), the observation follows directly from the valid path requirement (Condition 7.16). For paths of type (2), the observation follows directly from the definition of $\xrightarrow{\text{co}}$. □

Definition : The *critical+ operation* before operation Y in E_s is the last operation before Y in E_s that has a critical+ path to Y in E_s. If X is a critical+ operation before operation Y in E_s, then $X(i)$ is the critical+ sub-operation before $Y(i)$ in E_s (assuming both $X(i)$ and $Y(i)$ are defined).

Definition : *Reasonable sub-operations:* A sub-operation $Y(i)$ in E_m is *reasonable* for E_s if it obeys the following:

(1) Y is an essential operation in E_m.

(2) $Y(i)$ is in E_s.

(3) If there is a critical+ path from X to Y in E_s, then there is a valid+ path from X to Y in E_m and X is essential in E_m.

Definition : *Ok sub-operations:*

A write sub-operation in E_m is *ok* for E_s if it is reasonable for E_s, its control+ reads in E_m are ok for E_s, and the last conflicting write sub-operation before it in E_s is ok for E_s.

A read sub-operation in E_m is *ok* for E_s if it is reasonable for E_s, its control+ reads in E_m are ok for E_s, the critical+ write sub-operation before it in E_s is ok for E_s, and the last essential conflicting write sub-operation before it in E_m is ok for E_s.

We will show later that the above definition of ok sub-operations unambiguously defines each sub-operation as ok or not ok.

Definition : *Ok operations:* An operation in E_m is ok for E_s if all its sub-operations in E_m are ok for E_s.

Definition : *Semi-ok sub-operations and operations:* A sub-operation $O(i)$ or an operation O in E_m are *semi−ok* for E_s if the control+ reads of O in E_m are ok for E_s.

Properties of ok sub-operations:

Lemma E.2.1 describes the properties of ok sub-operations. It uses the following definitions.

Definition : *Special read:* A read R in E_s is a *special read* for E_s if there is a critical+ path from some W to R (in E_s) of the type in definition C.2 of Appendix C. Call this critical+ path a *special critical+ path* for R.

Definition : *Bad read-modify-write:* Consider a special read R in E_s. Let the special critical+ path for R be from W to R. Then for every W_1 between W and R (in E_s) that writes an exit value of R in E_s, there must be a read-modify-write immediately after W_1 (by definition C.2 of Appendix C). Call such read-modify-writes and their constituent operations and sub-operations in E_s as *bad* for R.

Lemma E.2.1: Below, ignore all unessential sub-operations in E_m. Below, we use R, W, etc. to represent sub-operations.

(1) Consider a write W that is in E_s and E_m, and ok for E_s. Then all writes that conflict with W and are before W in E_s are in E_m, are ok for E_s, and are before W in E_m in the same order as in E_s. Further, the operation of W is also ok for E_s.

(2) Consider a write W that is in E_s and E_m, and ok for E_s. If there are writes that conflict with W and are before W in E_s, then all sub-operations of all these writes and W are in E_m.

(3) Consider a read R that is in E_s and E_m. If the last write W before R in E_m that conflicts with R is ok for E_s, then W is before R in E_s.

(4) Consider a read R that is in E_s and E_m, is ok for E_s, and is an exit read of a synchronization loop. Then the exit values of R and the corresponding values to be written by the write of R's read-modify-write (if any) are the same in E_m and E_s.

(5) Consider a read R that is in E_s and E_m, and is ok for E_s. Then the last write before R that conflicts with R is the same in E_m and E_s (and so R returns the same value in E_m and E_s), or R is a special read in E_s such that the following is true. The last write W before R in E_m that conflicts with R is before R in E_s, the conflicting write immediately after W in E_s is bad for R in E_s, and the read of the bad read-modify-write is either not in E_m or is not ok for E_s.

(6) Consider a read R that is in E_s and E_m and is ok for E_s. If R returns the value of an unessential write in E, then the write is from a different processor in E.

Proof:

For a contradiction, consider the first sub-operation O in E_s that violates the conditions of the lemma. There are six cases depending on the part of the lemma that is violated.

Case 1: O violates part (1) of the lemma.

Let W_s be the last write in E_s before O that conflicts with O. There must be a critical+ path from W_s to O in E_s and so there is a valid+ path from W_s to O in E_m. Therefore, by observation E.2.1, W_s must be before O in E_m. Since O is ok for E_s, W_s must be ok for E_s too (by definition of ok sub-operations). By assumption, W_s obeys the lemma. Therefore, all conflicting writes before W_s in E_s are ok for E_s and in the required order before W_s in E_m. It follows that O cannot violate the first part of part (1) of the lemma. For the second part, since W_s is ok and obeys the lemma, it follows that all sub-operations of the operation of W_s are ok. It immediately follows that all sub-operations of the operation of O are ok and so the operation of O is ok, proving the second part.

Case 2: O violates part (2) of the lemma.

The proof for this case is similar to that for Case 1 as follows. Let W_s be the last write in E_s before O that conflicts with O. There must be a critical+ path from W_s to O in E_s and so there is a valid+ path from W_s to O in E_m. Therefore, by observation E.2.1, all sub-operations of W_s and O are in E_s. Since O is ok for E_s, W_s must be ok for E_s too (by definition of ok sub-operations). By assumption, W_s obeys the lemma. Therefore, all sub-operations of all conflicting writes before W_s in E_s are in E_m. It follows that O cannot violate part (2) of the lemma.

Case 3: O violates part (3) of the lemma but not part (6).

O must be a read R, the last write sub-operation before R in E_m that conflicts with R (say W_m) must be ok for E_s, and W_m must be after R in E_s. Let the first write that conflicts with R and is after R in E_s be W_s (W_s may be the same as W_m). The following first derives two facts that must be true for this case, and then shows that they lead to a contradiction.

Fact (a): R does not return the value of its own processor's write in E.

W_m is after R in E_s and before R in E_m; therefore (by the assumptions for valid paths and the valid path requirement), W_m cannot be from the same processor as R. Therefore, if R returns the value of W_m in E, then R does not return the value of its own processor's write in E. Otherwise, R returns the value of an unessential write in E. Since R does not violate part (6), it follows that R returns the value of another processor's write in E.

Fact (b): W_s must be before R in E_m and is ok for E_s.

If W_s is the same as W_m, then W_s is ok for E_s and before R in E_m (by the statement for this case). Otherwise, by the proof in Case (1) above, W_s must be ok for E_s and must be before W_m in E_m. Therefore, in all cases, W_s is ok for E_s and must be before R in E_m.

We show next that facts (a) and (b) lead to a contradiction. There must be a critical+ path from R to W_s in E_s. Since W_s is ok for E_s (by fact(b)), it follows that there is a valid+ path from R to W_s in E_m. Since (by fact (a)) R does not return the value of its own processor's write in E_m, it follows (from observation E.2.1) that R must be before W_s in E_m. This contradicts fact(b) above. Thus, O cannot violate part (3) of the lemma.

Case 4: O violates part (4) of the lemma.

Below, we use the term ‘‘exit values’’ of an exit read to refer to the exit values that the read needs to return to exit the loop, and to the values that the write of the read’s read-modify-write (if any) will write corresponding to an exit value of the read. By property (a) of the control relation, if all control reads (except possibly special reads) of O in E_m are in E_s and return the same value in E_m and E_s, then the exit values of O are the same in E_s and E_m. Since O is ok for E_s, all control reads of O in E_m are ok for E_s. It follows that all control reads of O are in E_s (by definition of ok sub-operations). These reads must be before O in E_s. Therefore, they obey the lemma and so a control read of O in E_m either returns the value of the same write in E_s and E_m or is a special read in E_s. It follows that the exit read and exit write values of O must be the same in E_s and E_m.

Case 5: O violates part (5) of the lemma.

Then O is a read R. Let the last write before R in E_m that conflicts with R be W_m. Then W_m is before R in E_s (by the proof of part (3)), and W_m is ok for E_s (by definition of ok sub-operations). Let the last write before R in E_s that conflicts with R be W_s. Let the critical+ write before R in E_s be W_c. Then W_c is before R in E_m (by observation E.2.1) and W_c is ok for E_s (by the definition of ok sub-operations). Only the following cases are possible.

Sub-case (5a): W_m is the same as W_s.

The lemma follows directly.

Sub-case (5b): W_m is not the same as W_s, W_c is the same as W_s.

Then W_s is ok for E_s and before R in E_m. Therefore, W_s is before W_m in E_s. We have seen above that W_m is before R and so before W_s in E_s. But then for W_s to be ok for E_s, part (2) of the lemma requires that W_m be before W_s in E_m, a contradiction.

Sub-case (5c): W_m is not the same as W_s, W_c is not the same as W_s.

Then R must be from a self-ordered loop. We show in the following three paragraphs that R must be a special read in E_s, and there is a bad read-modify-write for R after W_m in E_s.

There are two possible cases: W_c is different from W_m or W_c is the same as W_m. The following two paragraphs examine these cases.

Suppose W_c is different from W_m. Then W_c must be before W_m in E_s (because both are ok for E_s, W_c is before R in E_m, and so W_c is before W_m in E_m). W_m writes the same value in E_m and E_s (since W_m is ok). This value is an exit value for R in E_m; therefore by part (4) of the lemma, this value is also an exit value for R in E_s. This implies that R is a special read in E_s and there is a bad read-modify-write for R after W_m in E_s.

Suppose W_c is the same as W_m. Then again W_m writes an exit value for R in E_s. Since W_m is the critical+ write for R in E_s and W_m is not the same as W_s in E_s, it follows that R must be a special read in E_s and there is a bad read-modify-write after W_m for R in E_s.

Thus, we have shown so far that R is a special read for E_s, W_m is before R in E_s, and there is a bad read-modify-write for R after W_m in E_s. Denote the read and write of the bad read-modify-write by R_2 and W_2. If R_2 is not in E_m or not ok for E_s, then the proposition is proved. Therefore, assume that R_2 is ok for E_s. There are two cases depending on whether R_2 is after W_m in E_m or not.

Sub-case (5c1): R_2 is after W_m in E_m.

Let R’s processor be P_i and R_2’s processor be P_j. Let R’s mate be W. In E_m, $R(i) \xrightarrow{\text{xo}} W(j)$ (by construction of E_m).

First note that $R_2(j)$ cannot be between $R(i)$ and $W(j)$ in E_m, as proved in the following paragraph.

Suppose $R_2(j)$ is between $R(i)$ and $W(j)$. Then by the definition of an execution, $W_2(j)$ must be before $W(j)$ (otherwise, $W(j)$ is between $R_2(j)$ and $W_2(j)$ violating the condition for read-modify-writes). But then by loop coherence, $W_2(i)$ must also be before $W(i)$ in E_m. But $W(i)$ is after $R(i)$ in E_m. Thus, $W_2(i)$ is between $R(i)$ and $W(i)$. This is not allowed by the definition of an execution.

Similarly, $R(i)$ cannot be between $R_2(j)$ and $W_2(i)$.

We divide the rest of the argument of this sub-case into two further sub-cases depending on whether $R_2(j)$ is before or after $R(i)$ in E_m as follows.

Suppose $R_2(j)$ is before $R(i)$ in E_m. Then $W_2(i)$ must also be before $R(i)$ in E_m. Further, since $R_2(j)$ is after $W_m(j)$ and $W_2(j)$ is after $R_2(j)$, it follows (by the loop coherence part of the control requirement) that $W_2(i)$ is after $W_m(i)$ in E_m. Therefore, $W_2(i)$ must be between $W_m(i)$ and $R(i)$ in E_m. This is a contradiction since $W_m(i)$ is the last conflicting write before $R(i)$ in E_m.

Suppose $R_2(j)$ is after $R(i)$ in E_m. Then it must be after $W(j)$ in E_m. By definition of a special read and its bad operations, the value written by $W(j)$ is not an exit value for R_2 in E_s. By part (4) of the lemma, the value of $W(j)$ is not an exit value for R_2 in E_m as well. Therefore, there must be another write $W_3(j)$ between $W(j)$ and $R_2(j)$ in E_m and W_3 conflicts with R_2. Since R_2 is ok for E_s, by part (3) of the lemma, the last such W_3 must be ok for E_s and must be between W_m and R_2 in E_s, a contradiction.

Sub-case (5c2): R_2 is before W_m in E_m.

Then since by assumption, R_2 obeys the lemma, it must be that there is a bad read-modify-write before R_2 (and before W_m) in E_s such that the read of that bad read-modify-write is not ok for E_s. The read of the bad read-modify-write is a control read of its mate (by property (e) of the control relation); therefore, the write of the bad read-modify-write is not ok, and the write is before W_m in E_s. But then by part (1) of the lemma, W_m cannot be ok for E_s, a contradiction.

Case 6: O violates part (6) of the lemma.

O must be a read that returns the value of an unessential write *W* in *E*, and this write is from the same processor as *O*. Recall that a synchronization loop with an exit read-modify-write is allowed to have only a single exit read; further, the value written by an exit write is the same as that returned by the read of its read-modify-write. From the above observations and because *O* is essential in *E*, it follows that *O* cannot be from the same synchronization loop as *W*. If *W*'s loop instance terminates in *E*, then there must be another write between *W* and *O* in *E* corresponding to the read that returned the exit value. Thus, *O* cannot return the value of *W* in *E*. If *W*'s loop does not terminate in *E*, then there are infinite writes to the same location as *O* and all of these writes are before *O* by $\xrightarrow{\text{po}}$ in *E*. By the valid path requirement, all the sub-operations of the infinite writes that conflict with *O*'s sub-operation must be before *O* in *E*, a contradiction to the execution order condition (Condition 5.3). This covers all the sub-cases for this case. □

Properties of semi-ok sub-operations:

Lemma E.2.2: Consider an operation O in E_m that is semi-ok for E_s. Let operation O' be such that $O' \xrightarrow{\text{vpo}} O$ in E_m. Then O and O' are in E_s and O’ is semi-ok for E_s.

Proof

By property (a) of the control relation, if all control reads of O in E_m are in E_s and return the same value in E_m and E_s (with the possible exception of special reads), then O is in E_s. All control reads of O in E_m are ok for E_s. By Lemma E.2.1, all of these reads are in E_s, and all of these reads except possibly some special reads in E_s return the same value in E_m and E_s. It follows that O must be in E_s.

By property (c) of the control relation, all control+ reads of O' in E_m are control+ reads of O in E_m. These reads are ok for E_s. It follows that O' is semi-ok for E_s.

Lemma E.2.3: Consider an operation O in E_m that is semi-ok for E_s and an operation O' that is before O by $\xrightarrow{\text{vpo}}$ in E_s. Then, O' is in E_m and O' is semi-ok for E_s.

Proof:

By property (b) of the control relation, if all control reads of O in E_m are in E_s and return the same value in E_m and E_s, then O' is in E_m. All control reads of O are ok for E_s. They are therefore in E_s, and all but the special reads return the same value in E_m and E_s. By the argument in lemma E.2.2, the value of a special read cannot affect whether an operation will be executed as long as the loop of the special read terminates. It follows that O' is in E_m. By property (c) of the control relation, it follows that all control+ reads of O' in E_m are also control+ reads of O in E_m. Therefore, all control+ reads of O' are ok for E_s. It follows that O' is semi-ok for E_s. □

Lemma E.2.4: If X in E_m is non-semi-ok for E_s, then there must be a control+ read of X in E_m that is semi-ok and non-ok for E_s.

Proof:

There must be some control+ read of X in E_m that is non-ok for E_s. Consider the first such read R by program order in E_m. Then the control+ reads of R are also control+ reads of X, which are ok for E_s. Therefore, the control+ reads of R are ok for E_s, and so R is semi-ok for E_s, proving the proposition. □

Lemma E.2.5: Consider a read R that is in E_s and E_m and is semi-ok for E_s. If R returns the value of an unessential write in E, then the write is from a different processor in E.

Proof:

The proof is identical to that for Case 6 in lemma E.2.1. □

We next show that the definition of ok sub-operations unambiguously defines each sub-operation to be ok or not ok. Suppose for a contradiction that there is an ambiguous sub-operation in E_m. Then it must also be in E_s; otherwise, it is not ok. Consider the first sub-operation O in E_s that is ambiguous. (Note that determining whether certain valid+ paths exist in E_m and whether an operation is essential in E_m is unambiguous. Therefore, an ambiguity arises only if disambiguating a sub-operation requires determining if another operation is ok.)

Suppose O is a write. Then the decision of whether O is ok depends only on sub-operations before O in E_s. All the sub-operations before O are unambiguously known to be ok or not ok; therefore, O cannot be ambiguous, a contradiction.

Suppose O is a read. Then the decision of whether O is ok depends only on sub-operations before O in E_s and on whether the last essential conflicting write sub-operation (W_m) before O in E_m is ok. Thus O is ambiguous only if W_m is ambiguous. This is possible only if W_m is in E_s and is after O in E_s. However, by lemma E.2.1, if W_m is O in E_s, then W_m cannot be ok. Thus, O cannot be ambiguous.

Step E.3: A finite prefix (*Prefix*) of the execution order of $\mathbf{E_m}$.[44]

Let the *finite part* of E_m be the operations that do not follow infinite operations in E_m. (Recall that *follow* refers to program order.)

Lemma E.3.1: There must be a finite prefix of the execution order of E_m such that for every E_s, there is at least one essential sub-operation O in the prefix that is not ok for E_s. Further, if O is a read, then it is from the finite part of E_m.

Proof:

Assume that there is no finite prefix of the execution order of E_m of the type mentioned in the lemma. Then for any finite prefix of the execution order of E_m, there exists an E_s such that all essential reads in the prefix from the finite part of E_m and all essential writes in the prefix are ok for E_s. We show that then E_m must appear sequentially consistent, a contradiction. We show this through a series of results as follows. (Results 2, 3, 4, and 5 are relevant only if E_m obeys the modified uniprocessor correctness condition of Appendix B, but not the unmodified condition in Chapter 3.)

[44] The use of the term *prefix* in this appendix is unrelated to that of Chapter 8 or appendix I. This appendix uses prefix to imply an initial sub-sequence of the execution order such that if O is in a prefix, then all sub-operations before O in the execution order are also in the prefix.

Result 1: Consider an essential operation W in E_m such that $W(i)$ is not in E_m. Then there cannot be any essential operation O in E_m such that $O(i)$ could conflict with $W(i)$, and O is either a read from the finite part of E_m or a write.

Suppose for a contradiction that there is an O as described above. If there is a write that qualifies for O, then choose O to be a write. Consider the prefix of the execution order of E_m that consists of the sub-operations of O and W in E_m. If O can be a special read, then include the sub-operations of the mate W_1 of O as well. Then there is an E_s for which the sub-operations of O, W and W_1 are ok. There are three cases as follows.

Case 1: There is a critical+ path between O and W in E_s.

Then there must be a valid+ path between O and W in E_m. From observation E.2.1, either $W(i)$ is in E_m or O is a read that returns the value of its own processor's write W_2 in E_m. The former case is a contradiction. The latter case is also a contradiction since it requires W_2 to be chosen as O.

Case 2: O and W are consecutive conflicting operations in E_s and there is no critical+ path between O and W in E_s.

O must be a read from a self-ordered loop and W must write an exit value of O in E_s. By assumption, W is the only essential write in E_m that conflicts with O. Therefore, O cannot be a special read in E_s. From lemma E.2.1, the last conflicting write sub-operation before O's sub-operation must be the same in E_m and E_s. It follows that $W(i)$ must be in E_m, a contradiction.

Case 3: O and W are not consecutive conflicting operations, and O is before W in E_s.

There must be a conflicting write between O and W in E_s. From lemma E.2.1, it follows that all sub-operations of W must be in E_m, a contradiction.

Case 4: O and W are not consecutive conflicting operations, and O is after W in E_s.

If O is a write, then again from lemma E.2.1, it follows that all sub-operations of O and W are in E_m, a contradiction. Therefore, O must be a read. We know by lemma E.2.1 that if O is not a special read in E_s, then the last conflicting write before O in E_s must be ok for E_s. Therefore, it follows that all sub-operations of W are in E_m, a contradiction. If O is a special read in E_s, then its mate is ok for E_s; therefore, again all sub-operations of W are in E_m, a contradiction.

Result 2: Consider a loop instance L in the finite part of E_m such that L does not terminate in E_m but terminates in every E_s. Further, L is not a synchronization loop (that is exploited by model M in the sense discussed in Section 7.4). Then there are infinite reads in E_m that determine the termination of L in some sequentially consistent execution and form a race in that execution.

Suppose there is a loop instance L by processor P_i that does not obey the above result. Let the number of possible states of a processor be *NS*. Consider a sequentially consistent execution of *Prog*, E_{s1}, such that all essential operations of P_i in E_m until and including the first NS+1 iterations of L (as ordered by program order) are ok for E_{s1}. (Such an E_{s1} is possible because there is a finite prefix of the execution order of E_m that contains the operations of P_i until and including the first NS+1 iterations of loop L in E_m.) It follows that all the first NS+1 iterations of L must be in E_{s1}. Suppose none of the reads that determine the termination of L in E_{s1} form a race in E_{s1}. Then the termination of L depends only on the internal state of the processor at the beginning of each iteration, and the state of shared-memory at the beginning of the first iteration. The state of the processor at the beginning of an iteration must be the same for at least two of the first NS+1 iterations of L in E_{s1}. However, this implies that loop L cannot terminate in E_{s1}, a contradiction. Thus, the termination of L in E_{s1} must depend on a read R_1 which is involved in a race in E_{s1}, and this read must be in E_m. Let the number of iterations of L in E_{s1} be N1. Next consider a sequentially consistent execution of *Prog*, E_{s2}, for which the number of iterations of L in E_m that are ok is at least N1+NS+1 or 2NS+1, whichever is greater. Applying the same reasoning as for E_{s1}, there must be a race read in E_{s2} different from R_1 that determines the termination of loop L and is in E_m. Extending this argument, it follows that there are infinite reads in E_m that determine the termination of loop L in some sequentially consistent execution and form a race in that execution.

Result 3: There cannot be a write in the infinite part of E_m that forms a race in some sequentially consistent execution (without unessentials).

> Suppose for a contradiction that there is a write W in the infinite part of E_m that forms a race in some sequentially consistent execution (without unessentials). There must be a loop instance L preceding W in E_m such that L does not terminate in E_m.
>
> L must be guaranteed to terminate in all sequentially consistent executions; otherwise, by the finite speculation part of the control requirement, W cannot be in E_m.
>
> If L is from a synchronization loop such that if at least one exit read of the synchronization loop is involved in a race or if the loop is self-ordered, then by property (e) of the control relation, the infinite exit reads of L control W in E_m, a contradiction to the finite speculation part of the control requirement. Note that if L is from a synchronization loop, but the loop does not obey the above properties, then the synchronization loop cannot be exploited by model M in the sense discussed in Chapter 7.4.
>
> Then by result 2 and property (d) of the control relation, there are infinite reads that control W in E_m. This violates the finite speculation part of the control requirement, a contradiction.

Result 4: If X and W conflict in E_m, X is in the finite part of E_m, W is in the infinite part of E_m, and X and W are both essential in E_m, then $X(i) \xrightarrow{\text{xo}} W(i)$ for all i in E_m.

> Suppose there are X and W in E_m that do not obey the above result. Consider the first sub-operation in E_m that qualifies for X and consider a corresponding W. Let the processor that executes X be P_i. From result 1, we know that $W(i)$ is in E_m. By result 3, W cannot form a race in any sequentially consistent execution of *Prog*.
>
> Consider a prefix of the execution order of E_m that contains all sub-operations of W and X and the following. If X could be a special read in any E_s, then the prefix contains all sub-operations of the mate of X. The prefix also contains all sub-operations of all essential operations from the finite part of E_m that are ordered before X by $(\xrightarrow{\text{po}} \cup \xrightarrow{\text{co}})+$ of E_m. Call X and the set of essential operations from the finite part of E_m that are ordered before X by $(\xrightarrow{\text{po}} \cup \xrightarrow{\text{co}})+$ of E_m as set S. There must be an E_s such that all essential write sub-operations of the above prefix and all essential read sub-operations of the above prefix that are from the finite part of E_m are ok for E_s. Consider this E_s below.
>
> $X(i)$ must be ok for E_s, and so $W(i) \xrightarrow{\text{xo}} X(i)$ in E_s and W and X are consecutive conflicting operations in E_s. Further, W and X cannot form a race in any sequentially consistent execution. Therefore, there must be a path from W to X consisting of alternating $\xrightarrow{\text{po}}$ and $W_1 \xrightarrow{\text{co}} R_1$ arcs in E_s, where W_1 and R_1 form a race in some sequentially consistent execution of *Prog*. We show below that all operations on the above path must be from the finite part of E_m, must be from set S, and all their sub-operations must be ok for E_s, a contradiction since W is not from the finite part.
>
> > Suppose there is some operation on the above path that is not from the finite part of E_m or is not from set S or whose sub-operations are not all ok for E_s. Consider the last such operation O on the path. There are two cases discussed below.
> >
> > Suppose O is the first operation of a $\xrightarrow{\text{po}}$ arc. Then the next operation on the $\xrightarrow{\text{po}}$ arc is in the finite part of E_m and from set S and all its sub-operations are ok for E_s. If O is in E_m, then O must be in the finite part of E_m and so is in S and all its sub-operations are ok for E_s, a contradiction. Therefore, O must not be in E_m. But then some read in E_s that is in E_m and in the finite part of E_m and in S is not ok for E_s, a contradiction.
> >
> > Suppose O is the first operation of a $\xrightarrow{\text{co}}$ arc. Then the next operation on the arc (and its mate if it is a special read) is ok for E_s and in S and in the finite part of E_m. It follows that O must be in E_m. Further, O forms a race in some sequentially consistent execution and it is a write; therefore, O cannot be in the infinite part of E_m (by result 3). Therefore, O must also be in S and is ok for E_s, a contradiction.

Subsequent results use the following definition.

Definition : Consider an unessential read R in E_m. If there is a write sub-operation in the finite part of E_m such that this write and all conflicting writes after this write in the finite part of E_m write exit values for R in E_m, then call the operation of the first such write sub-operation in E_m as the *stable write* for R in E_m. If there is no such write, then the stable write for R is undefined.

Result 5: Consider an unessential read sub-operation R in the finite part of E_m that (in E_m) conflicts with an essential write sub-operation in the infinite part of E_m. Then there must be some exit read R_1 of the loop instance of R such that the stable write of R_1 is not defined in E_m.

> Suppose, for a contradiction, that in E_m, there is a read that violates the above. Let R be such a read that is the first such read of its processor (say P_i). Let W be the first essential write sub-operation in the infinite part of E_m that conflicts with R in E_m. By result 3, W cannot form a race in any sequentially consistent execution (without unessentials) of *Prog*.
>
> Consider a prefix of the execution order of E_m that contains all sub-operations of W, R, the stable write for R in E_m, the stable writes in E_m for all exit reads in R's loop instance that precede R, the mate write for R in E_m (if any), and all sub-operations of all essential operations from the finite part of E_m that are ordered before the above listed operations by ($\xrightarrow{\text{po}} \cup \xrightarrow{\text{co}}$)+ in E_m. Let the set of above mentioned operations be called the set S.
>
> There must be an E_s such that all essential write sub-operations of the above prefix and all essential read sub-operations of the above prefix that are from the finite part of E_m are ok for E_s and an iteration of the loop instance of R is in E_s. Consider such an E_s that additionally also obeys the following: E_s should have the minimum possible number of reads such that the read is R or an exit read of the loop instance of R that precedes R, and a conflicting write before this read in E_s is not in S. Call the above conditions on E_s the minimality conditions for E_s.
>
> From Result 3, W and R cannot form a race in any sequentially consistent execution. Further, since all sub-operations of W must be ok for E_s, W cannot be after R in E_s because R is unessential in E_m. Therefore, W must be before R in E_s. Similarly, all stable writes of all exit reads of R's loop instance that precede or include R are before the corresponding exit reads in E_s. Further, suppose in E_s, W' $\xrightarrow{\text{co}}$ R' where R' is either R or an exit read of R's loop instance that precedes R in E_s, and there is no path from W' to R' in the program/causal-conflict graph (see figure 4.2 for the definition) of E_s that contains at least one program order arc. Then the minimality conditions for E_s mentioned above imply that W' must be from S. (Otherwise, there is another sequentially consistent execution that obeys the minimality conditions on E_s and where R' is before W', thereby implying that the original E_s does not obey the minimality conditions.) Call the above the minimality inference.
>
> Consider the last conflicting write W' before R. If W' is from S and is not W, then W' is from the finite part of R and is essential in E_m. Therefore, W' must be ok for E_s. Further, W is before W' in E_s. Therefore W must be before W' in E_m. By result 4, it follows that W cannot be from the infinite part, a contradiction.
>
> If W' is from S and is W, then since W and R cannot form a race in any E_s, it follows that there must be a path from W to R consisting of alternating $\xrightarrow{\text{po}}$ and W_1 $\xrightarrow{\text{co}}$ R_1 arcs in E_s, where W_1 and R_1 form a race in some sequentially consistent execution (without unessentials) of *Prog*.
>
> If W' is not from S, then from the minimality inference, it follows that there must be a path from W' to R consisting of alternating $\xrightarrow{\text{po}}$ and W_1 $\xrightarrow{\text{co}}$ R_1 arcs in E_s, where W_1 and R_1 form a race in some sequentially consistent execution (without unessentials) of *Prog*.
>
> Thus, it follows that either W' is not from S or it is W, and there is a path from W' to R consisting of alternating $\xrightarrow{\text{po}}$ and W_1 $\xrightarrow{\text{co}}$ R_1 arcs in E_s, where W_1 and R_1 form a race in some sequentially consistent execution (without unessentials) of *Prog*. However, using the minimality inference and reasoning similar to that for result 4, we can show that all operations on the above path must be from S and must be from the finite part of E_m, a contradiction.

Result 6: There is no cycle in the program/conflict graph of E_m that involves only essential operations from the finite part of E_m.

> Suppose there is a cycle in the program/conflict graph of E_m that involves only essential operations from the finite part of E_m. Then one such cycle must be finite since there are only a finite number of processors, $\xrightarrow{\text{po}}$ is transitive, and there can be only a finite number of operations ordered before any operation by $\xrightarrow{\text{co}}$. Consider the prefix of E_m that consists of all the sub-operations of all operations on the above cycle. Further, if there are any reads on the cycle that could be special in some E_s, then include the sub-operations of their mates in the prefix as well. There must be an E_s where all of the above sub-operations are ok for E_s. Therefore, all $\xrightarrow{\text{co}}$ arcs on the above cycle are also $\xrightarrow{\text{co}}$ arcs in E_s. This implies a cycle in the program/conflict graph of E_s, a contradiction.

Result 7: There is a total order T on the essential operations of the finite part of E_m such that any finite prefix of this total order is the prefix of an execution order of a sequentially consistent execution.

> From result 6, there is a total order on the essential operations of the finite part of E_m such that the order is consistent with $\xrightarrow{\text{po}}$ and $\xrightarrow{\text{co}}$ of E_m. Consider any finite prefix of this order. All reads in the prefix that are essential reads in the finite part of E_m return the value (in E_m) of the last conflicting write as ordered by the above total order (from result 4). Thus, this prefix represents the prefix of an execution order of an execution. This order can be continued to represent an execution where all subsequent operations appear in program order and all sub-operations of a specific operation appear together. An execution with such an execution order obeys Condition 7.12 since it obeys the finite speculation and termination assumptions, and any two operations ordered by an ordering path are present in the execution order in the same order as the ordering path. This proves the above result.

Result 8: Consider an essential operation W in the finite part of E_m such that $W(i)$ is not in E_m and there is a conflicting unessential read $R(i)$ in the finite part of E_m. Then there must be some exit read R_1 of the loop instance of R such that the stable write of R_1 is not defined in E_m.

> Suppose, for a contradiction, that there is a $R(i)$ in E_m and a W that violates the above. Let R be such a read that is the first such read of its processor. By result 1, there cannot be any other essential write that conflicts with W in E_m. Also, all exit reads of R's loop must have a stable write in E_m.
>
> Consider the prefix of the total order T (T is defined in result 7) that contains W and all essential operations of R's processor before R and the stable writes of all exit reads of R's loop. By result 7, there is a sequentially consistent execution E_s where the above prefix of T is the prefix of the execution order of E_s. There must also be an E_s such that the operations immediately after the above prefix in E_s are the exit reads of R's loop until R. These must all be essential since they all have stable writes. Further, by the termination part of the control requirement, there should be a valid path from W to R in E_s. But then one such path must be in E_m as well, and so by the valid path requirement, $W(i)$ is in E_m.

Result 9: There are no unessential operations in the finite part of E_m.

> Suppose there is an unessential read R in the finite part of E_m.
>
> Note that there is no stable write for at least one of the exit reads of R's loop. Otherwise, in E_m, by result 5, all the exit reads of R's loop should have returned values from the finite part of E_m. But then all these reads should have returned exit values in E_m and so R should not be unessential, a contradiction.
>
> We identify two types of unessential exit reads in E_s that do not return their exit values in E_m and for which there is no stable write in E_m. For a type 1 read, the number of writes that conflict with the read in the finite part of E_m is finite. For a type 2 read, the number of writes that conflict with the read in the finite part of E_m is infinite. Further, since there is no stable write for the type 2 read, note that the number of writes in the finite part of E_m that write non-exit values for the read must also be infinite.
>
> Consider the total ordering T defined in result 7. To this ordering add the unessential operations of the finite part of E_m as follows. Consider the above unessential operations of a specific processor in

program order below. If the operation O is an exit read of type 1, then place it anywhere in T after the last unessential operation placed for this processor and after the last conflicting essential write W; make O return the value of write W. If the operation O is an exit read of type 2, then place it anywhere in T after the last unessential operation O' placed for this processor and after the first write W that conflicts with O and is after O' and writes a non-exit value of O (recall there are infinite such writes); make O return the value of W. If the operation O is any other operation, then insert it immediately after the previous operation of its processor by program order in the new total order; if it is a read, make it return the value of the last conflicting write before it in T; if it is a write from an exit read-modify-write, then make it write the value its read returns.

The total order generated above is an execution order of a sequentially consistent execution. In this execution, an exit read of type 1 or type 2 always returns a non-exit value and so the loop cannot terminate. However, a synchronization loop must terminate in all sequentially consistent executions, a contradiction.

Thus, there are no unessential operations in the finite part of E_m.

Result 10: E_m appears sequentially consistent.

Consider the total order T defined in result 7. Then this order is an execution order of some execution. This execution obeys the finite speculation and termination assumptions of Condition 7.12. Further, two operations ordered by an ordering path are similarly ordered by the execution order of this execution. Thus, this execution is sequentially consistent. Call it E_s. All reads from the finite part of E_m are in E_s and return the same value in E_s and E_m. Therefore, any loops in the finite part of E_m that do not terminate in E_m must also not terminate in E_s. It follows (from the finite speculation part of the control requirement) that there cannot be any instructions that access the output interface in the infinite part of E_m. Thus, E_s and E_m have the same result. Therefore, E_m appears sequentially consistent.

Lemma E.3.2: There must be a finite prefix, *Prefix*, of the execution order of E_m such that for every E_s, there is at least one essential sub-operation O in *Prefix* that is not ok for E_s but is semi-ok for E_s. Further, all control+ reads of all writes before and including O in E_m are in *Prefix*.

Proof:

From the previous lemma, there must be some finite prefix, *Prefix′*, of the execution order of E_m such that for every E_s, there is at least one essential sub-operation O_{ES} in the prefix that is not ok for E_s. Further, if O_{ES} is a read, then it is in the finite part of E_m. Consider the prefix of the execution order of E_m that contains *Prefix′*. Further, for every E_s, the prefix should also contain all the control+ reads of the corresponding O_{ES} in E_m, and all the control+ reads of a write W that is before O_{ES} or before a control+ read of O_{ES} in E_m. The number of control+ reads for any write is finite (by the finite speculation assumption); the number of control+ reads for any read in the finite part of E_m must also be finite (by definition of the finite part). Thus, a prefix of the execution order of E_m that contains all of the above sub-operations must be finite. We show below that this prefix qualifies for *Prefix* defined in the lemma.

Consider any E_s and the corresponding O_{ES}. If O_{ES} is semi-ok for E_s, then it qualifies as O for the lemma. If O_{ES} is not semi-ok for E_s, then by Lemma E.2.4, there must be a control+ read R of O_{ES} in E_m that is semi-ok for E_s but not ok for E_s. The read R is in the above prefix. If O_{ES} is a read, then R is clearly essential since O_{ES} must be in the finite part and is essential. If O_{ES} is a write, then again R must be essential; otherwise, by property (g) of the control relation, the finite speculation part of the control requirement is violated. Thus, R qualifies as O for the lemma. □

Below, we consider the prefix called *Prefix* defined by the above lemma, but ignore all unessential sub-operations in *Prefix*. Henceforth, when considering E_m, we also ignore all the unessential sub-operations of E_m in *Prefix*.

Step E.4: Construction of E_1 and existence of a non-ok $K(i)$ with special properties.

Consider an E_s, called E_1, such that the number of sub-operations in *Prefix* that are ok for E_1 is the maximum for any E_s. We later show that there must be another E_s such that all sub-operations in *Prefix* that are ok for E_1 are ok for E_s. Further, one more sub-operation from *Prefix* is ok for E_s. This is a contradiction. This step picks a candidate for the additional ok sub-operation. We use the following definition and lemmas.

Definition : *Good valid+ paths ($\xrightarrow{gvp}$) for E_1:*

Consider an operation Y that is in E_m and E_1. We say that $X \xrightarrow{gvp} Y$ in E_1 if there is a critical+ path from X to Y in E_1 that obeys the following.

(1) For a $\xrightarrow{co}$ arc on the path, the operations on the arc are in E_m and are similarly related by $\xrightarrow{co}$ of E_m.

(2) For a $\xrightarrow{po}$ arc on the path, the operations on the arc are in E_m.

(3) All operations on the path are semi-ok for E_1.

(4) Suppose $W \xrightarrow{co} R$ arcs exist on the path such that W needs to be the last essential conflicting write before R or R needs to return the value of another processor's write, then the above properties are obeyed by R in E as well.

(5) If X is a read that returns the value of its own processor's write in E_m, then $X(i) \xrightarrow{xo} Y(i)$ for all i in E_m.

Lemma E.4.1: If $X \xrightarrow{gvp} Y$ for E_1, then $X(i) \xrightarrow{xo} Y(i)$ in E_m.

Proof:

Properties (1)-(4) of a good valid+ path and the assumptions for a valid path (condition 7.18) imply that the path is a valid+ path in E_m also. From the valid path requirement and from property (5) above, it follows that $X(i) \xrightarrow{xo} Y(i)$ for all i in E_m. □

Lemma E.4.2: There must be at least one non-ok sub-operation, $Y(i)$, in *Prefix* such that

(a) $Y(i)$ is semi-ok for E_1,

(b) there is a critical+ path from X to Y in E_1, but there is no $\xrightarrow{gvp}$ path from X to Y in E_1, and

(c) if $Y(i)$ is a read, then the last conflicting write sub-operation before it in E_m is ok for E_1.

Proof:

The proof considers three exhaustive cases below. Note that a sub-operation in *Prefix* that violates property (3) for reasonable sub-operations obeys property (b) of the current lemma.

Case 1: There is a write in *Prefix* that is semi-ok but not ok for E_1.

Consider a write $W_1(i)$ in *Prefix* that is semi-ok but not ok for E_1 and is the first such write in E_1. Let the last conflicting write before $W_1(i)$ in E_1 be $W_2(i)$. There is a critical+ path from W_2 to W_1 in E_1. If this is not a $\xrightarrow{gvp}$ path, then W_1 qualifies for $Y(i)$. If it is a $\xrightarrow{gvp}$ path, then W_2 is semi-ok for E_1. But then by the choice of W_1, W_2 must be ok for E_1. It follows that W_1 must also be ok for E_1, a contradiction.

Case 2: There is a semi-ok non-ok read in *Prefix* that is a control+ read for some sub-operation in *Prefix* and returns the value of its own processor's write in E_m.

Consider the first non-ok, semi-ok read $R(j)$ in *Prefix* that is a control+ read and returns the value of its own processor's write in E_m. Then either the last conflicting write before $R(j)$ in E_m that conflicts with $R(j)$ is not ok for E_1, or $R(j)$ violates property (3) for reasonable sub-operations, or the critical+ write before $R(j)$ in E_1 is not ok for E_1.

Suppose the last conflicting write $W(j)$ before $R(j)$ in E_m is not ok. $W(j)$ and $R(j)$ are from the same processor. $W(j)$ must be semi-ok by lemma E.2.2, and it must be in *Prefix*. Thus, by Case 1 above,

$W(j)$ qualifies as $Y(i)$ in E_m. Below, we can assume that the last conflicting write before $R(j)$ in E_m is ok.

Suppose $R(j)$ violates property (3) of reasonable sub-operations. Then $R(j)$ qualifies for $Y(i)$.

So assume that the last critical+ write $W'(j)$ before $R(j)$ in E_1 is not ok. Then there is either a $\xrightarrow{\text{gvp}}$ path from W' to R in E_1 or there is not. In the former case W' is a semi-ok, non-ok write and in *Prefix* and so Case 1 implies a $Y(i)$. In the latter case, $R(j)$ qualifies for $Y(i)$.

Case 3: There is no semi-ok non-ok read in *Prefix* that is a control+ read for some sub-operation in *Prefix* and returns the value of its own processor's write in E_m.

By construction, there is some sub-operation O in *Prefix* that is semi-ok but not ok for E_1. If O is a write, then by Case 1, there is a $Y(i)$ in E_m, proving the lemma. So assume that O is a read. Consider the first read $R(j)$ in *Prefix* that is semi-ok but not ok for E_1.

Note that a conflicting write sub-operation before $R(j)$ in E_m, say $W(j)$ must be semi-ok because of the following. If $W(j)$ is not semi-ok, then by lemma E.2.4, there is a control+ read of $W(j)$ in E_m that is semi-ok and non-ok for E_1. By construction of *Prefix*, this read must be in *Prefix*; it is also essential in E_m. Therefore, by the statement of this case, if this read is a control+ read, then it returns the value of another processor's write in E_m. Therefore, by construction of E_m, this read must be before $W(j)$ and so before $R(j)$ in E_m. But then this read cannot be non-ok semi-ok because $R(j)$ is the first such read in *Prefix*, a contradiction.

Either the last write sub-operation before $R(j)$ in E_m that conflicts with $R(j)$ is not ok for E_1, or $R(j)$ violates property (3) for reasonable sub-operations, or the critical+ write before $R(j)$ in E_1 is not ok for E_1.

If the last write sub-operation before $R(j)$ in E_m that conflicts with $R(j)$ is not ok for E_1, then by the above note, it is semi-ok for E_1, and so by Case 1, there is a $Y(i)$. Therefore, below we can assume that the last conflicting write sub-operation before $R(j)$ in E_m is ok for E_1.

If $R(j)$ violates property (3) for reasonable sub-operations, then $R(j)$ is a $Y(i)$ and we are done.

So it must be that the critical+ write $W_s(j)$ before $R(j)$ in E_1 is not ok for E_1. Then there is either a $\xrightarrow{\text{gvp}}$ path from W_s to R in E_1 or there is not. In the former case W_s is a semi-ok, non-ok write in *Prefix* and so Case 1 implies a $Y(i)$. In the latter case, $R(j)$ qualifies for $Y(i)$. □

Call the first sub-operation in E_1 that qualifies as a $Y(i)$ of the above lemma as $K(i)$. Next we give a transformation on the execution order of E_1 that produces an execution order of an E_s such that $K(i)$ is ok for E_s, and all sub-operations in *Prefix* that are ok for E_1 are also ok for the above E_s, a contradiction.

Step E.5: Transformation on E$_1$.

This step consists of four parts. The first part consists of the transformation itself. The second consists of showing that the transformation produces an execution that preserves ok sub-operations. The third consists of showing that the transformation terminates. The fourth shows that the transformation terminates in an execution with the required properties.

Step E.5.1: The Transformation.

The transformation given below as $Trans(A_1, A_2)$ is a recursive procedure call that transforms the execution order of E_1. The initial A_1, A_2 for the call are chosen as follows.

Let A_2 be the first operation Z in E_1 such that Z is semi-ok for E_1 and there is a critical+ path from some Z' to Z in E_1 but no $\xrightarrow{\text{gvp}}$ path from Z' to Z in E_1. Further, if Z is an exit read of a synchronization loop, then the last conflicting write before Z in E_m is ok for E_1. (Z must exist since K itself qualifies as Z.) For A_1, we choose an operation that qualifies for Z' above as follows. If some operation qualifying as Z' for A_2 is such that $Z' \xrightarrow{\text{co}} A_2$ ends a critical+ path in E_1, then let A_1 be the last such operation in E_1. Otherwise, let A_1 be the last operation in E_1 that qualifies as Z' for A_2.

Before we proceed with the transformation, we introduce some terminology below.

We call the execution order of E_1 as the *original order*. At any point in *Trans*, the *current order* is the result of applying *Trans* until that point to the original order. We use A_{1j} and A_{2j} to represent A_1 and A_2 for call *j*. We show later that at the beginning of any call *j*, the current order until A_{2j} (with possibly a different value returned by A_{2j}) is the prefix of an execution order of some sequentially consistent execution of *Prog*. We use E_j to denote such an execution. In the specification of the procedure below, we assume we are in call *i*.

We say an operation *O* is a *race* if *O* could be in some sequentially consistent execution and form a race in that execution.

Call an operation *O* as *semi-ok+* for E_1 if (a) *O* is semi-ok for E_1 and (b) either *O* has a sub-operation in *Prefix*, or there is at least one operation after *O* in E_1 that is semi-ok for E_1 and has a sub-operation in *Prefix*.

Call an operation *O* as *ok+* for E_1 if (a) *O* is ok for E_1 and (b) either *O* has a sub-operation in *Prefix*, or there is at least one operation after *O* in E_1 that is semi-ok for E_1 and has a sub-operation in *Prefix*.

Call an operation *O* as *semi-ok++* for E_1 if (a) *O* is semi-ok+ for E_1 and (b) if *O* is an exit read of a self-ordered loop or a race exit read from a synchronization loop, then the last conflicting write before *O* in E_m is ok for E_1.

We extend the notion of a good valid+ path for the various E_j's produced by *Trans*. The definition below differs from the corresponding definition for E_1 in point (3).

Definition : *Good valid+ paths (* $\xrightarrow{gvp}$*) for* E_j*:*

Consider an operation *Y* that is in E_m and E_j. We say that $X \xrightarrow{gvp} Y$ in E_j if there is a critical+ path from *X* to *Y* in E_j that obeys the following.

(1) For a $\xrightarrow{co}$ arc on the path, the operations on the arc are in E_m and are similarly related by $\xrightarrow{co}$ of E_m.

(2) For a $\xrightarrow{po}$ arc on the path, the operations on the arc are in E_m.

(3) All operations on the path are semi-ok for E_j and semi-ok+ for E_1.

(4) Suppose $W \xrightarrow{co} R$ arcs exist on the path such that *W* needs to be the last essential conflicting write before *R* or *R* needs to return the value of another processor's write, then the above properties are obeyed by *R* in *E* as well.

(5) If *X* is a read that returns the value of its own processor's write in E_m, then $X(i) \xrightarrow{xo} Y(i)$ for all *i* in E_m.

As for lemma E.4.1, it follows that if $X \xrightarrow{gvp} Y$ for E_j, then $X(i) \xrightarrow{xo} Y(i)$ in E_m.

Call an arc of a critical+ path in E_j as *impure* if it violates (1) or (2) in the definition of a $\xrightarrow{gvp}$ path for E_j or either of its components is not semi-ok+ for E_1. Call the other arcs as *pure* arcs.

Let the set *Ok_sub−ops* denote $K(i)$ and the sub-operations in *Prefix* that are ok for E_1.

The transformation follows next.

Trans(A_1, A_2)*:*

If this is the first call to *Trans*, then let Q_2 be the first sub-operation in the execution order of E_1 and jump to Step 3.

If A_1 = * or ** or ***, return.

Step 1 - Consider the critical+ path from A_1 to A_2 in E_i. If there is no impure arc on this path, then call *Trans* (*,A_2) and return. If there is an impure arc, then call the last such arc on the path as B_1,B_2.

Step 2 - Consider all operations after B_1 that are ordered before B_2 by ($\xrightarrow{po} \cup \xrightarrow{co}$)+ in E_i. Move B_2, B_2's mate write (if any), and all above operations except B_1 to just above B_1 (and its mate read if any) in the current order, maintaining the relative order of the moved operations. Let Q_2 be the first moved operation in the new order.

Step 3 - (In the first call to *Trans*, this step tries to ensure that there are no bad read-modify-writes. For subsequent calls, this step tries to ensure that the ordering until Q_2 is the prefix of a sequentially consistent execution. For the first call, only the if-statements of (iv) and (vi) below are relevant and could be true.)

(i) If Q_2 is after all sub-operations in *Ok_sub−ops* in the current order, then call $Trans(**, Q_2)$ and return.

(ii) If the processor of Q_2 does not have any sub-operations that are semi-ok+ for E_1 following and including Q_2, then delete Q_2, mark the processor and go to Step 3(ix).

(iii) If the instruction instance of Q_2 cannot be in any sequentially consistent execution which has the current order until before Q_2 as a prefix of its execution order, then delete Q_2 from the ordering and go to Step 3(ix).

(iv) If for any sequentially consistent execution that has the current order until before Q_2 as a prefix of its execution order, there must always be other operations before Q_2 (or before Q_2 with a modified address and value), then do the following.

> If a bounded number of operations need to be inserted before Q_2, then insert the first operation just before Q_2, redefine Q_2 to be that operation and repeat Step 3. Also unmark the processor of Q_2.
>
> If an unbounded number of operations need to be inserted before Q_2 and some of these are from a loop that may not terminate in some sequentially consistent execution of *Prog* that has operations until before Q_2 as prefix of its $\xrightarrow{xo}$, then delete Q_2 and all the other operations following Q_2 from its processor and mark the processor of Q_2. Go to Step 3(ix).
>
> If an unbounded number of operations need to be inserted and all of them are from loops that terminate in every sequentially consistent execution that has operations until before Q_2 as prefix of its $\xrightarrow{xo}$, then go to Step 3(x).

(v) Consider a sequentially consistent execution that has the current order until before Q_2 as a prefix of its execution order, and has the instruction instance of Q_2. Make Q_2 access the same address and write the same value (if Q_2 is a write) as the corresponding operation in the above execution.

(vi) Suppose Q_2 is semi-ok++ for E_1. If Q_2 is an exit read of a synchronization loop and the last conflicting write before Q_2 writes a non-exit value for Q_2, then consider a sequentially consistent execution that has the current order until before Q_2 as a prefix of its execution order, it has Q_2 as essential, and it has Q_2 immediately after the first conflicting write after Q_2's current position that writes Q_2's exit value. Otherwise, consider a sequentially consistent execution that has the current order until before Q_2 followed by Q_2 (possibly returning a different value) as prefix of its execution order. Suppose there is a Q_1 that is before Q_2 in the current order such that there is a critical+ path from Q_1 to Q_2 in the considered sequentially consistent execution, but no $\xrightarrow{gvp}$ path from Q_1 to Q_2 for that execution. If there is such a Q_1 such that $Q_1 \xrightarrow{co} Q_2$ ends a valid+ path, then pick the last such Q_1 in the current order; otherwise, pick the last Q_1 in the current order. Call $Trans(Q_1, Q_2)$ and return.

(vii) If Q_2 is a read, make it return the value of the conflicting write operation last before it in the current order. If Q_2 is an exit read of a synchronization loop and the value returned is not an exit value for Q_2, then go to Step 3(x). Otherwise, unmark the processor of Q_2.

(viii) Consider any sequentially consistent execution that has the current order until Q_2 as a prefix of its execution order. If Q_2 could be a bad read-modify-write for a special read in any such execution, Q_2 is not ok for the execution and either the special read is ok+ for E_1, or it is semi-ok+ for E_1 and a race, then go to Step 3(x).

(ix) If there is an operation after Q_2 in the current order, then redefine Q_2 as the next operation in the current order and repeat Step 3. Otherwise, assume ω is a hypothetical last operation of the execution and call $Trans(**, \omega)$ and return.

(x) Mark the processor of Q_2. Consider the first operation O of all processors after and including Q_2 that satisfy the following.

First, there is no operation O' between and including Q_2 and O such that (a) O' conflicts with O or its mate, and (b) O' is either ok+ for E_1, or O' is semi-ok++ for E_1 and races with O (or its mate) in some E_s.

Second, if the processor of O is marked, then the following should be true. (a) there should be no preceding operations that need to be inserted before O, (b) O is semi-ok++ for E_1, (c) O conflicts with O' that is before Q_2, (d) in any sequentially consistent execution that has the current order until before Q_2 as a prefix of its execution order and where O is essential, there is a critical+ path from O' to O but no $\xrightarrow{\text{gvp}}$ path from O' to O for that execution.

Now move the first of the above considered operations and its mate to before Q_2. Redefine Q_2 as the first moved operation and repeat Step 3.

Suppose there is no operation of the type O. If inserting a ''missing'' operation of a processor before Q_2 (in a manner that ensures the resulting order until the inserted operation is a prefix of a sequentially consistent execution) could result in terminating the first loop of any processor after Q_2 (assuming the loop does not terminate with the state of memory represented just before Q_2), then insert such an operation, call it Q_2, and repeat Step 3. Otherwise, call Trans(***,Q_2) and return.

Step E.5.2: Properties of the Transformation.

Below, B_{1j} and B_{2j} represent B_1,B_2 chosen in Step 2 of the j^{th} call to *Trans*.

Lemma E.5.1: The following results hold for the j^{th} call to *Trans*, $j \geq 1$. Below, Q is an operation in E_j that is before A_{2j}.

(a) B_{1i},B_{2i} must race with each other in E_i and B_{2i} is semi-ok+ for E_1, where call i is before call j.

(b) The current order at the beginning of call j until A_{2j} is the prefix of an execution order of a sequentially consistent execution of *Prog*.

(c) If Q is semi-ok++ for E_1 and there is a critical+ path from Q' to Q in E_j, then there is a $\xrightarrow{\text{gvp}}$ path from Q' to Q in E_j.

(d) (i) An operation that is semi-ok+ for E_1 is in the current order at the beginning of call j.

(ii) The value returned by a read that is ok+ for E_1 does not change until the j^{th} call. An exception is a read that is special for E_1; if the value of this read was not the same in E_1 and E_m, then it changes to become the same as that of E_m.

(iii) If Q or A_{2j} is semi-ok+ for E_1, then it is semi-ok for E_j.

(iv) If Q is semi-ok+ for E_1 and is a write, then Q is ok for E_j.

(v) If Q is semi-ok++ for E_1, Q is not self-ordered in E_j, then all operations before Q in E_j that conflict with Q must be semi-ok+ for E_1 and semi-ok for E_j, and are before Q in E_m.

(vi) If Q is ok+ for E_1, then Q is ok for E_j.

(vii) Consider two conflicting operations O' and O in E_m such that both are either ok+ for E_1, or both are semi-ok++ for E_1 and form a race in some E_s. Then once any conflicting suboperations of these operations are in the same order as in E_m, *Trans* does not change the ordering of these sub-operations until call j.

Proof:

We proceed by induction.

Base case: (a), (b), (c), and (d) clearly hold for E_1.

Induction: Suppose the above results hold for the n^{th} call to *Trans*. We next prove that they hold for the $n+1^{st}$ call to *Trans*.

Result (a) -

Suppose the result is not true. Then either B_{1n},B_{2n} do not form a race in E_n or B_{2n} is not semi-ok+ for E_1. Consider the critical+ path from A_{1n} to A_{2n} of E_n for which B_{1n},B_{2n} have been chosen as the last impure arc. Either there is a pure arc after B_{2n} on the above path, or B_{2n} is the last operation on the path. In either case, B_{2n} must be semi-ok+ for E_1. By induction hypothesis, B_{2n} is semi-ok for E_n. Thus, it must be that B_{1n},B_{2n} do not form a race in E_n. Only the following cases are possible.

Case (a1): $B_{1n} \xrightarrow{\text{po}} B_{2n}$ in E_n.

> By lemma E.2.3, B_{1n} is in E_m and is semi-ok for E_n. By lemma E.2.2, B_{1n} is semi-ok+ for E_1. But then B_1, B_2 is not impure, a contradiction.

Case (a2): $B_{1n} \xrightarrow{\text{co}} B_{2n}$ in E_n and B_{2n} is not a self-ordered read or a race exit read from a synchronization loop.

> Suppose B_{2n} is before A_{2n} in E_n. Then by part d(v) of this lemma and the induction hypothesis, B_{1n} must be semi-ok+ for E_1 and semi-ok for E_n, and $B_{1n} \xrightarrow{\text{co}} B_{2n}$ in E_m. It follows that B_{1n},B_{2n} is not impure, a contradiction.
>
> Suppose B_{2n} is A_{2n}. Then B_{1n} cannot be the same as A_{1n} (since B_{1n},B_{2n} do not race and are on a $\xrightarrow{\text{gvp}}$ path from A_{1n} to A_{2n} in E_n). If B_{1n} and B_{2n} are consecutive conflicting operations, then let X be B_{1n}; otherwise, let X be the last conflicting write between B_{1n} and B_{2n} in E_n. There must be a critical+ path from X to B_{2n} in E_n. Therefore, there must be a $\xrightarrow{\text{gvp}}$ path from X to B_{2n} in E_n (otherwise X, B_{2n} would be chosen as A_{1n}, A_{2n}). Therefore, X is semi-ok+ for E_1 and $X \xrightarrow{\text{co}} B_{2n}$ in E_m. By result d(iii) and the induction hypothesis, X is semi-ok for E_n as well. It follows from result d(v) that B_{1n} is semi-ok+ for E_1 and semi-ok for E_n, and $B_{1n} \xrightarrow{\text{co}}$ B_{2n} in E_m. Thus, B_{1n},B_{2n} is not an impure arc, a contradiction.

Case (a3): $B_{1n} \xrightarrow{\text{co}} B_{2n}$ in E_n, B_{2n} is a self-ordered read or a race exit read from a synchronization loop, and $B_{2n} \xrightarrow{\text{po}} B_3$ is on the critical+ path from A_{1n} to A_{2n} in E_n.

> Then B_3 is semi-ok+ for E_1 and semi-ok for E_n. By property (e) of the control relation, B_{2n} controls B_3 in E_m. Therefore, B_{2n} is ok for E_1 and E_n. Suppose B_{2n} is not a special read in E_n or it is a special read such that the reads from its bad read-modify-writes in E_n are ok for E_n. Then by lemma E.2.1, the last conflicting write operation W before B_{2n} is the same in E_m and in E_n and is ok+ for E_1 and E_n. From result d(v), B_{1n} must be semi-ok+ for E_1 and by result d(iv) B_{1n} must be ok+ for E_1. Further, $B_{1n} \xrightarrow{\text{co}} B_{2n}$ in E_m. By result d(vi), B_{1n} is ok+ for E_n. It follows that B_{1n},B_{2n} is not impure, a contradiction. Thus, B_{2n} must be a special read in E_n and the read from its bad read-modify-write must not be ok for E_n. But this is not possible because of step 3(viii) and 3(x) of *Trans* which ensures that such a bad read-modify-write cannot be before A_{2n}; further, if it is A_{2n}, then *Trans* terminates for that call.

Case (a4): $B_{1n} \xrightarrow{\text{co}} B_{2n}$ in E_n, B_{2n} is a self-ordered read or a race exit read from a synchronization loop, and $B_{2n} \xrightarrow{\text{co}} B_3$ is on the critical+ path from A_{1n} to A_{2n} in E_n.

> This case implies a $W_1 \xrightarrow{\text{co}} R \xrightarrow{\text{co}} W_2$ chain on a valid path. This contradicts our assumptions for a valid path (condition 7.18).

Case (a5): $B_{1n} \xrightarrow{\text{co}} B_{2n}$ in E_n, B_{2n} is a self-ordered read or a race exit read from a synchronization loop, and B_{2n} is the last operation on the critical+ path from A_{1n} to A_{2n} in E_n.

> If A_{1n} is not the same as B_{1n}, then the path from A_{1n} to A_{2n} is not a critical path because A_{1n} and A_{2n} are not consecutive conflicting operations in E_n, and a critical path that is not between consecutive conflicting operations must end in a $\xrightarrow{\text{po}}$ arc. The path is also not a critical+ path because A_{1n} and A_{2n} do not race, a contradiction.
>
> If A_{1n} is the same as B_{1n}, then the only way the path from A_{1n} to A_{2n} can be a critical+ path is if it forms a race, a contradiction.

Result (b) -

By the induction hypothesis, at call n, the order until A_2 of call n is a prefix of the execution order of a sequentially consistent execution. Since then, the only changes to the order since call n upto the beginning of call $n+1$ are due to Steps 2 and 3 in call n. For step 2, since B_1,B_2 form a race, the change preserves program order and atomicity. The only changes made by Step 3 are to mimic the order of a sequentially consistent execution until the A_2 of call $n+1$.

Result (c) -

We know that result (c) is true for E_n until before A_2 of E_n and therefore until before B_1 of E_n. The ordering until before B_1 of E_n stays the same in E_{n+1}. So if Q was before B_1 in E_n, result (c) cannot be violated in E_{n+1}. If Q was after or same as B_1 in E_n and is before A_2 in E_{n+1}, then Q should have been selected as A_2 in Step 3 of *Trans*, a contradiction. Thus, result (c) cannot be violated.

Result (d) -

Suppose result (d) is not true for E_{n+1}. We know that result (d) is true for E_n. In going from E_n to E_{n+1}, changes are made to the order in Steps 2 and 3 of *Trans*. The only operations that can result in a violation of result (d) are those examined by Step 3 of *Trans* before the A_2 of call $n+1$. Below, we say an operation is successfully examined in Step 3 if its examination in some iteration of Step 3 does not result in going to Step 3(x) of Trans, or if it is selected as A_2. Then if some operation violates result (d), there must be one such operation that is successfully examined by Step 3 of *Trans* at call $n+1$ and is before or the same as A_2 of call $n+1$. Let O be the first such operation. We examine each part of result (d) as a separate case below, and show that either O cannot violate that part or must also violate a later part.

Case (d1): O violates d(i).

Then one of the control+ reads for O in E_m is not ok for E_{n+1}. This was ok for E_1 and so existed in E_n and so must have been successfully examined by Step 3 before O. Therefore, this read should have been chosen as O, a contradiction.

Case (d2): O violates d(ii).

If O is not ok for E_{n+1}, then a later case applies, so assume that O is ok for E_{n+1}. By construction of *Trans*, if O is a special read for E_{n+1}, then its bad read-modify-writes must be ok for E_{n+1}. So by lemma E.2.1, O must return the same value in E_{n+1} and in E_m. So the result is true when O is special. If O is not special, then O should have returned the same value in E_n as well. By induction hypothesis, this is the same value until call $n+1$.

Case (d3): O violates d(iii).

One of the control+ reads for O in E_m must not be ok for E_{n+1}. This read was ok for E_1 and so existed in E_n and so must have been successfully examined before O, a contradiction.

Case (d4): O violates d(iv).

By d(iii), O must be semi-ok for E_{n+1}. By result (c), O obeys property (3) for reasonable sub-operations. Therefore, the only way that O is not ok for E_{n+1} is if the last conflicting write O' before O in E_{n+1} is not ok for E_{n+1}. There is a critical+ path from O' to O in E_{n+1}, and so a $\xrightarrow{gvp}$ path from O' to O in E_{n+1}. It follows that O' is semi-ok+ for E_1. Further, O' must have been successfully examined before O. So O' should have been selected as O, a contradiction.

Case (d5): O violates d(v).

Let O' be the last conflicting operation before O in E_{n+1}. Then there is a critical+ path from O' to O in E_{n+1}, and so a $\xrightarrow{gvp}$ path from O' to O in E_{n+1}. If follows that O' is semi-ok+ for E_1 and $O' \xrightarrow{co} O$ in E_m. Further, O' must be semi-ok for E_{n+1}; otherwise, it should have been selected as O, a contradiction.

Case (d6): O violates d(vi).

O cannot violate property (3) of reasonable sub-operations; otherwise, O would be A_2 for E_{n+1}. Also all control+ reads of O in E_m must be ok for E_{n+1}; otherwise, that read would be O. By result d(iv), O cannot be a write.

Thus, O must be a read and either the last critical+ write operation before O in E_{n+1} is not ok for E_{n+1}, or the last conflicting write operation before O in E_m is not ok for E_{n+1}.

Suppose the last conflicting write, O', before O in E_m is not ok for E_{n+1}. O' must be ok for E_1 and before O in E_1. If O' is not before O in E_{n+1}, then O violates d(vii) and is discussed in the next case. If O' is before O in E_{n+1}, then it must be ok, or should have been selected as O, a contradiction.

Suppose the last critical+ write O' before O in E_{n+1} is not ok for E_{n+1}. Suppose O' is semi-ok+ for E_1. Then O' must be ok for E_{n+1} or it would be chosen as O, a contradiction in both cases. Suppose O' is not semi-ok+ for E_1. Then O would be A_2, a contradiction.

Case (d7): O violates d(vii).

There must be an O' such that O and O' satisfy the requirements of d(vii), $O' \xrightarrow{\text{co}} O$ in E_m and E_n, but $O \xrightarrow{\text{co}} O'$ in E_{n+1}. By inspection, Step 3 could not change the order of O and O'. Thus, it must have been changed by Step 2. In Step 2, the only change is that B_2 and some operations between B_1 and B_2 (and B_2's mate write) are moved to before B_1. Suppose B_1 and one of the moved operations (X) satisfy the criteria for O and O'.

Suppose X is B_2. We know that $B_1 \xrightarrow{\text{co}} X$ in E_m. But then B_1,B_2 is pure, a contradiction.

Suppose X is B_2's mate write. Let B_2's processor be P_i. We know that $B_2(i) \xrightarrow{\text{xo}} X(i) \xrightarrow{\text{xo}} B_1(i)$ in E_m (otherwise B_1,B_2 is not impure), and $B_1(j) \xrightarrow{\text{xo}} X(j)$ for some j in E_m. Thus, it follows that B_1 and X are not coherent in E_m. Therefore, by the loop coherence part of the control requirement, neither B_1 nor X can be from a synchronization loop. Further, since both B_1 and X must form a race in some sequentially consistent execution, B_1 cannot be the mate write of a read-modify-write. It follows that there is some E_s such that B_1 and X form a race in that E_s and both are essential. It follows then that B_1 and X must be coherent in E_m, a contradiction.

Suppose X is before B_2. We know that $B_1 \xrightarrow{\text{co}} X$ in E_m. If X, B_2 conflict, then $X \xrightarrow{\text{co}} B_2$ must be in E_m; otherwise, some other operations should have been A_1, A_2. So $B_1 \xrightarrow{\text{co}} B_2$ in E_m. But then B_1,B_2 is not impure, a contradiction. If X, B_2 do not conflict (i.e., both are reads), then there must be a conflicting write W between them (because B_1,B_2 race). $W \xrightarrow{\text{co}} B_2$ must be in E_m and W must be semi-ok+ for E_1 (else W, B_2 would be A_1, A_2). So $X \xrightarrow{\text{co}} W$ must be in E_m. So $B_1 \xrightarrow{\text{co}} B_2$ must be in E_m, a contradiction. □

Lemma E.5.2: For every call to *Trans* (A_1,A_2) after the first call, Step 2 of *Trans* finds a B_1, B_2, or A_1 = **, or A_1 = ***.

Proof:

For every call to *Trans* after the first call, either the call is the last call, or there is a B_1,B_2 for the call, or the call makes the last call. If it is the last call, then either A_1 = ** or A_1 = *** or Step 2 of the previous call did not find a B_1,B_2. Thus, it is sufficient to show that for each call to *Trans* that is not the first or last call, Step 2 finds a B_1,B_2. For a contradiction, assume that call k of *Trans* does not have a B_1,B_2, and it is not the first or last call. In E_k, there is a critical+ path from the corresponding A_1 to the corresponding A_2, but there is no $\xrightarrow{\text{gvp}}$ path from A_1 to A_2 in E_k. Since there is no B_1, B_2, the critical+ path from A_1 to A_2 in E_k obeys parts (1), (2) and (3) for the definition of a $\xrightarrow{\text{gvp}}$ path for E_k. We show below that such a path also obeys parts (4) and (5) in the definition of a $\xrightarrow{\text{gvp}}$ path of E_k. In that case it must follow that the path is also a $\xrightarrow{\text{gvp}}$ path in E_k, a contradiction.

Case 1: There is a read on the critical+ path from A_1 to A_2 of E_k that does not obey part (4) in the definition of a $\xrightarrow{\text{gvp}}$ path for E_k.

Consider the last read R on the path in E_k that satisfies this case. Let W_m be the write whose value R returns in E_m. By assumption, R is semi-ok+ for E_1 and semi-ok for E_k. Further, $W \xrightarrow{\text{co}} R \xrightarrow{\text{po}} X$

is on the critical+ path in E_k, and either W is the last conflicting write before R in E_k but not in E_m, or R returns the value of another processor's write in E_k but not in E (and so not in E_m either). In the former case, by property (f) of the control relation and since X is semi-ok for E_1 and E_k, R must be ok for E_k; therefore W_m is before R in E_k. In the latter case, clearly W_m is before R in E_k. It follows that in all cases, W_m is before R in E_k. For a contradiction, it is sufficient to show that the last conflicting write before R in E_k is the same as W_m.

Consider the critical+ write W_s before R in E_k. There must be a $\xrightarrow{\text{gvp}}$ path from W_s to R in E_k (else R would be chosen as A_2 which is not possible since R cannot be the last operation on the path), and so $W_s \xrightarrow{\text{xo}} R$ in E_m, W_s is semi-ok+ for E_1, and so by lemma E.5.1, W_s is ok for E_k. There are two sub-cases.

Sub-case 1a: W_s and W_m are not the same.

Then W_m must be between W_s and R in E_k. So R is self-ordered in E_k. W_m writes an exit value of R in E_m and so by property (a) of the control relation, W_m must also write an exit value of R in E_k. Therefore, R must be a special read in E_k. Then by property (e) of the control relation, R must be ok+ for E_1 and (by lemma E.5.1) R must be ok for E_n. It follows that if $k \neq 2$, then R returns the value of the same write in E_k and E_m, a contradiction. If $k = 2$, then if the bad read-modify-write for R is ok for E_k, again R returns the value of the same write in E_m and E_k, a contradiction. If the bad read-modify-write for R is not ok for E_k, then R cannot be before A_2 by construction of *Trans*.

Sub-case 1b: W_s and W_m are the same.

Then W_m is ok for E_k. If W_s is the last conflicting write before R in E_k, it follows that R returns the value of the same write in E_m and E_k, a contradiction. If W_s is not the last conflicting write before R in E_k, then R must be self-ordered in E_k. As for sub-case 1a, R must be a special read in E_k since W_m writes an exit value of R in E_m and so in E_k. The rest of the argument is the same as for sub-case 1a; i.e., R must be ok for E_1 and so for E_k, implying that R must return the value of the same write in E_m and E_k, a contradiction.

Case 2: The critical+ path from A_1 to A_2 in E_k does not obey part (5) in the definition of a $\xrightarrow{\text{gvp}}$ path for E_k.

In this case, the critical+ path from A_1 to A_2 in E_k is from a read R to a write W where R returns the value of its own processor's write W_1 in E_m and $R(i)$ is after $W(i)$ in E_m. Since R is semi-ok+ for E_1 and semi-ok for E_k, (by lemma E.2.2) W_1 must be in E_1 and E_k and must be semi-ok+ for E_1 and semi-ok for E_k. By lemma E.5.1, W_1 must be ok for E_k. If W_1 and W are consecutive conflicting sub-operations in E_k, then let W_1 be Z below; otherwise, let Z be the last conflicting write before W in E_k. Then there must be a critical+ path from Z to W in E_k. We show first that all arcs on this path must be pure.

Assume for a contradiction that there is some impure arc on the above path. Let the last such arc on the path be from X to Y. There are three cases as follows.

Sub-case 2a: $X \xrightarrow{\text{po}} Y$ in E_k.

Y must be semi-ok+ for E_1 and semi-ok for E_k; therefore, X must be semi-ok+ for E_1 and semi-ok for E_k; therefore, the arc is pure, a contradiction.

Sub-case 2b: $X \xrightarrow{\text{co}} Y$ in E_k and Y is not W.

Y is semi-ok+ for E_1 and semi-ok for E_k. By lemma E.5.1, $X \xrightarrow{\text{co}} Y$ cannot be impure unless Y is self-ordered or a race exit read of a synchronization loop. In either case, Y must be ok for E_1 (by property (e) of the control relation). Further, Y must be ok+ for E_1 and so ok for E_k. But then $X \xrightarrow{\text{co}} Y$ in E_m and further X is semi-ok+ for E_1 and ok for E_k. Thus, $X \xrightarrow{\text{co}} Y$ is not impure, a contradiction.

Sub-case 2c: $X \xrightarrow{\text{co}} Y$ in E_k and Y is W.

X and W must be consecutive conflicting operations in E_k because there cannot be a conflicting write between X and W. Then there is a critical+ path from X to W. But there cannot be a $\xrightarrow{gvp}$ path, since otherwise $X \xrightarrow{co} W$ is not impure. But then since $X \xrightarrow{co} W$ ends the critical+ path from Z to W, $X \xrightarrow{co} W$ should have been chosen as A_1, A_2, a contradiction. (Note that $R \xrightarrow{co} W$ cannot end a critical+ path since we choose the longest possible paths to be critical+ paths. By the assumptions for valid paths (Condition 7.18), the valid path from R to W can always be substituted for $R \xrightarrow{co} W$ in a critical+ path.)

Thus, the path from Z to W obeys (1), (2), and (3) in the definition of $\xrightarrow{gvp}$ paths for E_k. By the arguments of Case 1 above, it follows that all operations on the path also obey (4) in the definition of $\xrightarrow{gvp}$ paths for E_k. Thus, the path from Z to W is a $\xrightarrow{gvp}$ path. Therefore, $Z \xrightarrow{co} W$ in E_m and Z is semi-ok+ for E_1 and semi-ok for E_k. By lemma E.5.1 it follows that $W_1(i) \xrightarrow{xo} W(i)$ for all i in E_m. But then it follows that $R \xrightarrow{co} W$ in E_m, a contradiction.

Lemma E.5.3: For every call to *Trans* (A_1,A_2) that finds a B_1,B_2 in Step 2, A_1, $A_2 = B_1$, B_2.

Proof:

Suppose the lemma is not true for call k. Then by lemma E.5.1(a), B_1,B_2 form a race in E_k. We also know that B_2 is semi-ok+ for E_1 and therefore semi-ok for E_k.

If B_2 is not A_2, then by lemma E.5.1(d(v)), B_1 must be semi-ok+ for E_1 and semi-ok for E_k and $B_1 \xrightarrow{co} B_2$ in E_m, or B_2 is self-ordered, or B_2 is a race exit read from a synchronization loop. In the first case, B_1,B_2 is not impure, a contradiction. In the second and third cases, B_2 is ok+ for E_1 and so ok for E_n. Therefore $B_1 \xrightarrow{co} B_2$ in E_m. Further, B_1 is semi-ok+ for E_1. Therefore, B_1, B_2 is not impure, a contradiction.

If B_2 is A_2, then B_1 is not A_1. Since B_1, B_2 race in E_k, there is a critical+ path from B_1 to B_2 in E_k. It follows that there must be a $\xrightarrow{gvp}$ path from B_1 to B_2 in E_k (otherwise, B_1 should be A_1). But then B_1 is semi-ok+ for E_1 and so semi-ok for E_k, and $B_1 \xrightarrow{co} B_2$ in E_m. But then B_1,B_2 is not impure, a contradiction.

Step E.5.3: The Transformation Always Terminates.

Lemma E.5.4: Trans always terminates.

Proof:

Suppose *Trans* does not terminate. Then by the previous lemma, every call after the first call to *Trans* has a B_1,B_2. This is the same as A_1, A_2, so A_1, A_2 get swapped in every call. Further, note that B_2 is semi-ok++ for E_1. Also, note that the same operation can be chosen as A_2 for only a finite number of successive calls and then some other operation must get chosen as A_2.

Consider the k^{th} call of *Trans* and its corresponding B_{1k},B_{2k} such that the k-1st call had a different B_2. Consider the sequence of operations in E_k that are semi-ok+ for E_1, are before B_{1k}, and were a B_{2j} for some call j before call k. We prove below that for every call j before call k, the above sequence for E_k either (a) is different from the corresponding sequence for E_j and is also not the prefix of the corresponding sequence for E_j, or (b) is the same as the corresponding sequence for E_j and A_2 for all E_l since and including E_j up to E_k is the same as for E_k.

Suppose the sequence for E_k does not obey the above properties. Then B_{2k} of E_k must have been a B_{2j} for a call j before call $k-1$ such that at the end of call j, B_{1k} was not before B_{2k}. Thus, the only way that B_{1k} can come before B_{2k} at the end of call $k-1$ is if in between, some other B_{1p},B_{2p} were swapped such that B_{1p} was before B_{1k} or before a control read of B_{1k} in E_p, and B_{2p} was not same as B_{2k}. Consider the last call q before call k such that B_{1k} was before B_{2k}, B_{1q} was before B_{1k} or before a control read of B_{1k} in E_q, and B_{2q} was not B_{2k}. Then the sequence at call k is the sequence for call q followed by B_{2k} which must be a unique sequence.

Since *Prefix* is finite, it follows that the number of operations that are semi-ok+ for E_1 is finite, and so the number of unique sequences involving such operations is finite and so the number of calls to *Trans* must be finite. Thus, we need to show that each call to *Trans* terminates. The call is tail recursive so we only need to show that the last call to *Trans* terminates. Consider the last call to *Trans*. Then either the call sequentially examines all operations (in which case it must cover all operations in *Ok_sub-ops* and so terminate), or it is interrupted because there are marked operations and other out of sequence operations are pulled up in the current order or *Trans* terminates. When out of sequence operations are pulled up, either more marked processors are generated, or from then on, *Trans* resumes its sequential examination. Eventually, *Trans* will cover all operations that are semi-ok+ for E_1 or will not pull up any operations and terminate.

Step E.5.4: The Transformation Terminates with the Required Execution.

We first analyze the case when *Trans* terminates with ***. Let the final call be f and its corresponding sequentially consistent execution be E_f.

Call all operations after and including A_{2f} as pending operations. Then for every processor P_i and its first pending operation O_i, one of the following must be true. (We say O_i is waiting for a pending operation if there is a pending operation before O_i in the final order that conflicts with O_i (or its mate) and that is either ok+ for E_1, or it is semi-ok++ for E_1 and forms a race with O_i (or its mate) in some E_s.)

(1) O_i is an exit read from a synchronization loop, the last write before A_{2f} that conflicts with O_i writes a non-exit value of O_i, and O_i is not waiting for a pending operation.

(2) There should be a loop before O_i that does not terminate at A_{2f}, and placing any more operations of this loop in the execution will not make any such or above loops terminate. Further, the loop is guaranteed to terminate in every sequentially consistent execution with operations until before A_{2f} as prefix of the execution order.

(3) O_i is from a read-modify-write synchronization loop, O_i is not ok+ for an E_f that has O_i immediately after all the operations before A_{2f} in the current order (and so O_i is not ok for E_1), O_i is a bad read for another loop read R after it where R is ok+ for E_1 or R is a race that is semi-ok+ for E_1, and O_i is not waiting for a pending operation.

(4) O_i is waiting for a pending operation, O_i is an exit read from a synchronization loop, the last write before A_{2f} that conflicts with O_i in the final order writes a non-exit value of O_i, and O_i is not ok+ for E_1.

(5) O_i is waiting for a pending operation and does not satisfy case 4 above.

(6) The pending operations of P_i do not consist of any operations that have sub-operations in Ok_sub-op, or operations that are semi-ok+ for E_1, or operations that might terminate the above loops of other processors.

Call the loops in (1), (2), (3), (4) above as pending loops. We next prove some properties about the pending loops.

Lemma E.5.5: Consider a pending operation O whose processor has a pending loop. (i) If O is the first pending exit read of a pending synchronization loop, then it cannot be ok+ for E_1. (ii) If O is not the first pending exit read of a pending synchronization loop, then O cannot be semi-ok+ for E_1 and be a race or an exit read of a self-ordered loop.

Proof:

Let L be the pending loop of O's processor. There are four cases depending on the type of loop L is.

Case 1: L is a loop of type (1).

Then the first exit read R of L cannot be ok+ since its value must have changed in some call of *Trans* and is different from its value in E_m. This proves the first part of the lemma. For the second part, note that R must be a race operation. Therefore, if O is a race or an exit read of a self-ordered loop, then R controls O by property (e) of the control relation. But then O cannot be semi-ok+ for E_1, proving the second part of the lemma.

Case 2: L is a loop of type (2).

Then the first part of the lemma is trivially true. For the second part, since the loop is guaranteed to terminate for every sequentially consistent execution (with operations until before A_{2f} as the prefix of the execution order), but cannot terminate before A_{2f}, it follows that the termination of the loop must depend on some shared-memory reads that form a race and that are not present in the current order. These reads cannot be semi-ok+ for E_1 because they were either not present in E_1 or were deleted from E_1. However, some reads that control these reads in E_m are in E_1 and are not ok+ for E_1. Therefore, if *O* is a race or an exit read of a self-ordered loop, then these reads control *O* (by property (d)). But then *O* cannot be semi-ok+ for E_1, proving the lemma.

Case 3: L is a loop of type (3).

We know that the exit read of L is not ok+ for E_1. This proves the first part of the lemma. We also know that the exit read of L must be a race or an exit read of a self-ordered loop. If *O* is a race or an exit read of a self-ordered loop, then by property (e) of the control relation, the exit read must control *O*. But then *O* is not semi-ok+ for E_1.

Case 4: L is a loop of type (4).

We know that the exit read of L is not ok+ for E_1. This proves the first part of the lemma. The exit read of L must also be a race operation. If *O* is a race or self-ordered operation, then by property (e) of the control relation, the exit read must control *O*. But then *O* is not semi-ok+ for E_1. □

Lemma E.5.6: If *Trans* terminates with ***, then there must be some processor whose first pending operation is of type (5).

Proof:

For a contradiction, assume that *Trans* terminates with *** and all processors' first pending operations are of type (1)-(4) or (6).

Suppose one of the processors P_i is of type (3); i.e., its first pending operation, O_i, is a read of a bad read-modify-write. Consider the last such O_i. Then there must be some other read *R* from a self-ordered loop where *R* is ok+ for E_1 or *R* is semi-ok for E_1 and is from a race operation, and *R* is after O_i. The last conflicting write before O_i is not pending and writes an exit value for O_i. Thus, this write also writes an exit value for *R*. Therefore, *R* cannot be an exit read of a pending loop. But it also cannot follow a pending loop by the previous lemma and since *R* must be from a self-ordered loop. Thus, *R* must be from a processor whose first pending operation is of type (5), a contradiction.

Suppose all processors are of type (6). Then *Trans* must terminate with **, a contradiction.

Thus, at least one processor must be of type (1), (2), or (3). The pending loops of type (1), (2), and (3) are guaranteed to terminate in every sequentially consistent execution (with operations before A_{2f} as a prefix of the execution order). However, they do not terminate before A_{2f} and therefore cannot terminate for any sequentially consistent execution that has the operations until before A_{2f} as a prefix of its execution order, a contradiction. □

Consider the first pending operation *J* in the final order that is from a processor of type (5). Then *J* is waiting for a pending operation *I*. If both *J* and *I* are reads for every *I*, then *J* must be a read from a read-modify-write. In that case, call the write of *J*'s read-modify-write the waiter. Otherwise, call *J* the waiter. If the waiter is a read, then consider the last write that qualifies for *I*. If *J* is a write, then consider the first operation that qualifies for *I*.

Lemma E.5.7: I must be ok+ for E_1, *I* must be from a processor that has a pending loop, *I* is not the first exit read of its processor's pending loop, *I* is not a race operation or an exit read from a self-ordered loop.

Proof:

First note that *I* must be from a processor with a pending loop; otherwise, *I* is from a processor of type (5) and so should have been chosen as *J*, a contradiction.

Suppose *I* is the first exit read of its processor's pending loop. Then *I* is not ok+ for E_1 (by lemma E.5.5). The last conflicting write W_m before *I* in E_m is ok+ for E_1 and is before *I* in the final order. If W_m is a pending operation, then W_m should have been selected as *I*, a contradiction. So W_m is before A_{2f}. So W_m is ok for E_f. If W_m is the last write to its location before A_{2f} in E_f, then since it writes an exit value for *I*, *I* should be a bad read-modify-write. But then I must be ok for ‘‘an E_f with I just before A_{2f}’’, a contradiction. Therefore, the last conflicting write before I and A_{2f} in E_f must be a write that is not W_m. This can only be if I is a special read and a read of its bad read-modify-write is not ok for E_f (by construction of *Trans*). But this is not possible with *Trans*. Thus, I is not the first exit read of its processor’s pending loop.

Since I is from a processor with a pending loop, from lemma E.5.5 it follows that I must be ok+ for E_1 and must not be a race or self-ordered operation. This proves the above lemma. □

Lemma E.5.8: There is a sequentially consistent execution, E_g, without unessential operations, that is the same as E_f until just before A_{2f}, and has the following property. If I is in E_g, then either there exists a conflicting operation before I in E_g that was not before A_{2f} in E_f, or there exists a write before I in E_g such that it is not semi-ok+ for E_1 and it is before A_{2f} in E_f.

Only the following cases are possible.

Case 1: Either J is not an exit read of a synchronization loop, or the last write conflicting with J before A_{2f} writes an exit value for J.

Consider a sequentially consistent execution whose execution order is the same as the current order until before A_{2f}, then has J, and then proceeds in any way. If J is from a synchronization loop with a single exit read, then clearly J is essential. If J is from a synchronization loop with multiple exit reads, we can still consider J as the essential exit read. If I is in this execution, then I is after J. Thus, this execution qualifies for E_g.

Case 2: J is an exit read from a synchronization loop, the last write conflicting with J before A_{2f} does not write an exit value for J, and I is a write.

In this case, J must be ok+ for E_1; otherwise, it would qualify as being from a pending loop, and so its processor is not of type (5). Recall that I is the last write before J in E_f that is ok+ for E_1. From the definition of *Trans*, it follows that I writes the exit value of J. Consider a sequentially consistent execution whose execution order is the same as the current order until before A_{2f}, and then proceeds in any way. Consider only essential operations in this execution. If I is in this execution, then J must come before I because otherwise it is possible to have an execution where J comes just after I, making I a race operation, a contradiction.

Case 3: J is an exit read from a synchronization loop, the last write conflicting with J before A_{2f} does not write an exit value for J, and I is a read.

In this case, J must be part of a read-modify-write. As in case 2, J must be ok+ for E_1. Therefore, the last conflicting write W before J and A_{2f} must not be semi-ok+ for E_1. Consider any E_g which has the operations before A_{2f} as a prefix of its execution order. If I is in E_g, then W is before I in E_g.

We next show that *Trans* cannot terminate with A_1 = ***.

Lemma E.5.9: Trans cannot terminate with A_1 = ***.

Proof:

Suppose *Trans* terminates with A_1 = ***. Consider the E_g described in the previous lemma. We show that the existence of E_g is a contradiction. There are two cases.

Case 1: I is in E_g and all control+ reads of I in E_m are in E_g and return the same value as in E_1.

Let W be the last conflicting write before I in E_g. Since I cannot be self-ordered, there must be a critical+ path to I from W. The path is also a valid+ path. Further, either W is not before A_{2f} or W is before A_{2f} and is not semi-ok+ for E_1. If any $X \xrightarrow{co} Y$ arc on the path is such that there is another valid+ path from X to Y, then consider a path where such a $\xrightarrow{co}$ arc is replaced with the corresponding valid+ path. All $\xrightarrow{co}$ arcs on the new path are either race paths or are from a write to a self-ordered read. Further, all $\xrightarrow{po}$ arcs on such a path are $\xrightarrow{vpo}$ arcs. There are two sub-cases.

Sub-case 1a: W is not before A_{2f}.

It follows that there must be an operation O in I’s processor P_i that (a) is the same as or follows the first pending operation of P_i in E_g, (b) O is a race or a self-ordered read in E_g, and (c) O is either I or is before I by ($\xrightarrow{\text{vpo}}$)+ in E_g. Further, since I is ok+ for E_1 and all control reads of I return the same value in E_g and E_m (and therefore in E_g and E_1), it follows (from (c) above) that O must be in E_1. So O must also be semi-ok+ for E_1. But then from lemma E.5.5 and (b) above, it follows that O must be the first exit read of its processor's pending loop. Further, O must form a race in E_g, and so O must control I in E_m (by property (e) of the control relation). It follows that I cannot be ok+ for E_1, a contradiction.

Sub-case 1b: W is before A_{2f}.

It follows from arguments similar to the above sub-case that all operations on the above path other than I must be from before A_{2f} in the final order. Further, all operations on the path must be semi-ok+ for E_1. Therefore, W is semi-ok+ for E_1, a contradiction.

Case 2: I is not in E_g or some control+ reads of I in E_m are not in E_g or do not return the same value in E_g and E_1.

Some read in E_m that follows a pending loop and was ok+ for E_1 executes in E_g but returns a different value in E_g and E_1. Consider the first such read R as ordered by the final order. Note that by lemma E.5.5, the read cannot be a self-ordered read. Only the following cases are possible. Below, W is the last write before R in E_g.

Sub-case 2a: W is not in E_1 and is not before A_{2f} in the final order.

There must be a valid+ path from W to R in E_g. This is a contradiction by an argument identical to that used for Case 1a above and substituting R for I.

Sub-case 2b: W is not in E_1 and is before A_{2f} in the final order.

There must be a valid+ path from W to R in E_g. As in Case 1 above, replace this path with a longer path so that all $\xrightarrow{\text{co}}$ arcs on the path are either race or self-ordered paths and all $\xrightarrow{\text{po}}$ arcs are $\xrightarrow{\text{vpo}}$ arcs. It follows from arguments similar to Case 1 above that operations from all processors other than that of R must be from before A_{2f} in the final order. Further, all operations on the path must be semi-ok+ for E_1. Therefore, W is in E_1, a contradiction.

Sub-case 2c: W is in E_1 but is not the last conflicting write before R in E_1.

Let the last conflicting write before R in E_1 be W_1. Then W_1 must be ok+ for E_1 and W_1 must be between W and R at the end of the last call to *Trans*. Thus, some control read of W_1 must return different values in E_1 and E_g. Then if W_1 is not from a processor with a pending loop, then a sub-operation of the first pending operation of W_1's processor should have been chosen as J, a contradiction. If W_1 is from a processor with a pending loop, then some other read should have been chosen as R, a contradiction. □

Let the last call to *Trans* be f. Consider E_f.

Lemma E.5.10: All sub-operations in *Prefix* that were ok for E_1 are ok for E_f and $K(i)$ is ok for E_f.

Proof:

By the previous lemmas, *Trans* terminates with **. Thus, all operations with a sub-operation in *Prefix* that are ok for E_1 and $K(i)$ are before A_2 for E_f. Therefore, from the previous lemmas, all operations with sub-operations in *Prefix* that are ok for E_1 are ok for E_f. $K(i)$ is semi-ok+ for E_1. Therefore, by the previous lemmas, if $K(i)$ is a write, then $K(i)$ is ok for E_f, proving the lemma. If $K(i)$ is a read, then the last conflicting write before $K(i)$ in E_m is ok+ for E_1 and is in *Prefix*; therefore, it is ok for E_f. Thus, if $K(i)$ is not ok for E_f, then either the critical+ write W before it in E_f is not ok for E_f or there is no valid+ path from W to K in E_m. But by the previous lemma, there must be a $\xrightarrow{\text{gvp}}$ path from W to K in E_f; therefore, there is a valid+ path from W to K in E_m. Further, W is semi-ok+ for E_1 and so ok+ for E_f. Thus, $K(i)$ is ok for E_f, proving the lemma. □

The proof of theorem E.1 follows from lemma E.5.10. □

Appendix F: Correctness of Low-Level System-Centric Specification of Data-Race-Free-0

This appendix proves the correctness of the low-level system-centric specifications of the data-race-free-0 model (Conditions 5.11 - 5.16), based on the framework developed in Chapter 7. The proof shows that the conditions for data-race-free-0 meet the generic valid path and control requirements of a generic model in Chapter 7 for all data-race-free-0 programs.

In any sequentially consistent execution of a data-race-free-0 program, all conflicting pairs of synchronization operations are ordered by $\xrightarrow{hb0}$ paths, while all conflicting pairs of operations where at least one of the pair is data are ordered by $\xrightarrow{hb0+}$ paths. Thus, $\xrightarrow{hb0}$ paths between conflicting synchronization operations, and $\xrightarrow{hb0+}$ paths between other conflicting operations form a set of critical paths for the execution. It follows that the above paths can be considered to be the valid paths of data-race-free-0.[45]

With the above valid paths, the data and synchronization requirements of data-race-free-0 (Conditions 5.11 - 5.15) obey the generic valid path requirement of a generic model in Condition 7.16. We show that the control requirement of data-race-free-0 (Condition 5.16) obeys the generic control requirement for a generic model in Condition 7.21. There are five parts to the generic requirement: critical set, finite speculation, write termination, and loop coherence. Loop coherence is not relevant to data-race-free-0 since it does not exploit synchronization or self-ordered loops. The following three paragraphs show that the data-race-free-0 control requirement obeys the other three parts of the generic requirement.

First consider the critical set part. Parts (a), (b), (c), and (d) of Condition 5.16 implicitly specify a relation between a read and some following operations. This relation directly satisfies properties (a), (b), (c), and (d) of the control relation (Definition 7.19) specified for the generic control requirement. (Note that part (c) and the condition for synchronization loops in parts (b) and (d) of Definition 7.19 are not relevant to data-race-free-0 since data-race-free-0 does not exploit synchronization loops.) Condition 5.16 ensures that if R is ordered before X by the above relation, then $R(i) \xrightarrow{xo} X(j)$ for all i, j. This ensures that all control paths are executed safely, thereby obeying the critical set part of the generic control requirement.

Next consider the finite speculation part. Part (a) requires that the number of operations ordered before any write by $\{ \xrightarrow{ctl} \}+$ be finite. Since $R \xrightarrow{ctl} X$ implies that $R(i) \xrightarrow{xo} X(j)$ for all i, j and since there can only be finite sub-operations before any sub-operation by $\xrightarrow{xo}$, it follows that part (a) of the finite speculation requirement is trivially satisfied by Condition 5.16. Part (b) of finite speculation is directly satisfied by part (e) of Condition 5.16.

The write termination part of the generic requirement requires that some sub-operations of certain writes in the execution should be in the execution, and has a requirement for self-ordered loops. The latter requirement is not relevant to data-race-free-0. The former requirement is imposed on writes that can form a race in some sequentially consistent execution of the program. In data-race-free-0 programs, such writes are distinguished as synchronization. Therefore, part (f) of the data-race-free-0 condition ensures that the generic write termination requirement is also obeyed. This completes the proof.

[45] This appendix assumes that data-race-free-0 does not ignore unessential operations. If, however, unessential operations are ignored with data-race-free-0, then the assumptions for valid paths (Condition 7.18) require that system designers assume a modified version of the $\xrightarrow{so0+}$ relation as follows. $W \xrightarrow{so0+} R$ iff W and R are both synchronization and $W(i)$ is the last essential conflicting sub-operation before $R(i)$ by the execution order (assuming R is issued by processor P_i). In contrast, the definition in Chapter 4 requires $W(i)$ to be the last conflicting sub-operation before $R(i)$. This modification does not affect programmers since programmers need only consider sequentially consistent executions without unessential operations.

Appendix G: System-Centric Specifications of PLpc1 and PLpc2

This appendix discusses system-centric specifications for PLpc1 and PLpc2 that allow the optimizations discussed in Chapter 6. Based on the results of Chapter 7, the valid paths of a model completely define system-centric specifications of the model. Sections G.1 and G.2 derive a set of valid paths for PLpc1 and PLpc2 respectively, and use the results of Chapter 7 to show that the system-centric specifications corresponding to these valid paths allow the optimizations discussed in Chapter 6. We also indicate which reads can be self-ordered since that information can be exploited by aggressive implementations of the control requirement. The specifications and proofs are similar to those for the PLpc model [AGG93].

G.1 The PLpc1 Model

Theorem G.2 below gives a set of valid paths for the PLpc1 model. The theorem uses several types of *valid conflict order* relations ($\xrightarrow{\text{vco1}}$, $\xrightarrow{\text{vco2}}$, $\xrightarrow{\text{vco3}}$) to capture the possible conflict order arcs on the valid paths (analogous to the $\xrightarrow{\text{vpo}}$ relation for the program order arcs). Definition G.1 first gives these relations. Following the statement of the theorem, we show how it allows the PLpc1 optimizations discussed in Chapter 6, and then prove the theorem is correct.

In the following, we say an operation is of a particular category if it is distinguished as that category. We use W to denote any write, SW to denote a synchronization write, NLUW to denote a non-loop or unpaired write, LW to denote a loop write, and DW to denote a data write. We use R, SR, NLUR, LR, and DR correspondingly for reads. We use S, NLU, L, D to denote either a read or a write of the corresponding category. We use numerical suffixes to the above if we need to refer to two operations of the same category. X, Y, Z, A, B denote any memory operations.

When using the notion of paired synchronization after ignoring unessentials, the assumptions for valid paths (Condition 7.18) require that system designers assume a modified version of the notion of *paired* as follows. A write *W* should be considered paired with a read *R* (issued by processor P_i) if *W* and *R* are distinguished as pairable with each other and if $W(i)$ is the last essential conflicting sub-operation before $R(i)$ by execution order. In contrast, the definition in Chapter 6 requires $W(i)$ to be the last conflicting sub-operation before $R(i)$. This modification does not affect programmers (or our analysis for the proof) since programmers and our analysis need only consider sequentially consistent executions without unessential operations.

Definition G.1: *The valid conflict order ($\xrightarrow{vco}$) relations for an execution E and the PLpc1 model:*

X and Y below are memory operations from different processors in *E*.

X $\xrightarrow{\text{vco1}}$ Y iff X and Y are respectively the first and last operations in one of

NLUR $\xrightarrow{\text{co}}$ NLUW

SW $\xrightarrow{\text{co}}$ SR, the last conflicting write before SR in *E* is from a different processor than SR

NLUW $\xrightarrow{\text{co}}$ NLUW

NLUR $\xrightarrow{\text{co}}$ NLUW $\xrightarrow{\text{vco1}}$ SR

X $\xrightarrow{\text{vco2}}$ Y iff X and Y are respectively the first and last operations in

SW $\xrightarrow{\text{co}}$ SR, the last conflicting write before SR in *E* is from a different processor than SR

X $\xrightarrow{\text{vco3}}$ Y iff

X $\xrightarrow{\text{vco2}}$ Y, X and Y are paired with each other.

Theorem G.2: *Valid paths of PLpc1:* The following constitute a set of valid paths for PLpc1.

(1) X ($\xrightarrow{\text{po}}$ ∪ $\xrightarrow{\text{vco1}}$)+ Y, X and Y are non-loops or unpaired synchronization.

(2) X ($\xrightarrow{\text{po}}$ ∪ $\xrightarrow{\text{vco2}}$)+ Y, both X, Y are synchronization, and at least one of X or Y is loop.

(3) X ($\xrightarrow{\text{po}}$ ∪ $\xrightarrow{\text{vco3}}$)+ Y, at least one of X or Y is data.

(4) X ($\xrightarrow{\text{po}}$ ∪ $\xrightarrow{\text{vco1}}$)+ Y, X is a write from a read-modify-write whose read is non-loop or unpaired synchronization, Y is a read from a read-modify-write whose write is non-loop or unpaired synchronization.

Further, in all of the above paths, no two consecutive edges are both $\xrightarrow{\text{po}}$ or both due to a $\xrightarrow{\text{vco}}$ relation.

Note that paths of type (4) described in theorem G.2 are executed safely if paths of type (1) are executed safely. For paths of type (1) - (3), by inspection it follows that SW $\xrightarrow{\text{po}}$ SR is a $\xrightarrow{\text{vpo}}$ arc only if both the write and read are non-loop or unpaired synchronization (or both are to the same location). Thus, a synchronization write followed by a synchronization read to a different location need be executed in that order only if both are distinguished as non-loop or unpaired. (The other $\xrightarrow{\text{vpo}}$ arcs are similar to data-race-free-1; i.e., acquire $\xrightarrow{\text{po}}$ data, data $\xrightarrow{\text{po}}$ release, SR $\xrightarrow{\text{po}}$ S, SW1 $\xrightarrow{\text{po}}$ SW2, and arcs between operations to the same location.) From the proof below, it also follows that only synchronization reads are exploited as self-ordered with respect to synchronization writes. This information can be used for aggressive implementations of the control requirement.

We say a read (or a write) is an *intrinsic loop read* (or *intrinsic loop write*) in a sequentially consistent execution E_s if it obeys definition 6.7 for a loop read (or definition 6.8 for a loop write) for E_s. A path of type (1), (2), (3), or (4) refers to paths of the type specified in theorem G.2.

Consider a PLpc1 program *Prog*. Let E_s denote a sequentially consistent execution of *Prog* without unessential operations. We prove that theorem G.2 is correct by proving that for any E_s, there is a critical set of paths for E_s that consists only of paths of the type (1), (2), (3), and (4). We use the following lemmas. Below, terms indicating order (e.g., last, after, first, between, etc.) implicitly refer to the execution order of the considered execution (unless otherwise specified). Further, since we will only be considering execution orders of sequentially consistent executions, we assume execution orders are on operations rather than sub-operations.

Lemma G.3: If $X \xrightarrow{\text{co}} Y$ in E_s and X and Y form a race[46] in E_s, then $X \xrightarrow{\text{vco1}} Y$ in E_s.

Proof:

X and Y must be synchronization operations because they form a race in E_s.

X cannot be an intrinsic loop read because of the following. Suppose X is an intrinsic loop read. Then by definition 6.7 of a loop read, X must be an exit read of a synchronization loop in E_s. But then it must return the value of the write it races with in E_s. Since $X \xrightarrow{\text{co}} Y$ in E_s, it follows that X cannot return the value of Y in E_s, a contradiction.

Y cannot be an intrinsic loop write. This is because by definition 6.8 of an intrinsic loop write, an intrinsic loop write can only race with an intrinsic loop read.

Thus, X is either NLUR or SW, and Y is either NLUW or SR. Further, if X is an intrinsic loop write, then Y must be a read because an intrinsic loop write can only form a race with a read. The lemma follows immediately. □

Lemma G.4: Define the relation (and graph) *rel* (on the memory operations of an execution) as the union of program order and arcs of the type $W \xrightarrow{\text{co}} R$ where W is the last conflicting write before R and is from a different processor than R. Let X and Y be synchronization operations in E_s. If there is a path in *rel* from X to Y in E_s that ends in a $\xrightarrow{\text{po}}$ arc, then there is a path of type (2) from X to Y that ends in a $\xrightarrow{\text{po}}$ arc.

[46] The proofs in this appendix apply to all definitions of a race.

Proof:

Consider a path in *rel* from X to Y in E_s that ends in a $\xrightarrow{po}$ arc and such that no two consecutive arcs on the path are $\xrightarrow{po}$ arcs and the path has fewest possible data operations. If no $\xrightarrow{co}$ arcs on the above path have data operations, then the above path is of type (2) and the lemma follows. So suppose that some $\xrightarrow{co}$ arc has data operations. Then there must be a $\xrightarrow{hb1}$ path between the two operations on the above $\xrightarrow{co}$ arc. Replace the above $\xrightarrow{co}$ arc with the $\xrightarrow{hb1}$ path. The resulting path also implies a path in *rel* from X to Y that satisfies all the criteria for the chosen path but has fewer data operations than the chosen path, a contradiction. □

If operation X is from a read-modify-write, then let X' be the other operation of the read-modify-write. Similarly, define Y' for Y.

Lemma G.5: Consider two conflicting operations X and Y in E_s such that there is no *rel* path from X to Y that ends in a $\xrightarrow{po}$ arc. Consider the operations O that are between X and Y and have a *rel* path to Y that ends in a $\xrightarrow{po}$ arc; do not include Y' in O. Then there is another sequentially consistent execution E_s' such that the execution order of E_s' is as follows. The $\xrightarrow{xo}$ first has all the operations of the execution order of E_s until just before X (in the same relative order) with possibly the exception of X' (if X is a write of a read-modify-write) and Y' (if Y is a write of a read-modify-write and Y' is before X), followed by all the operations in O (in the same relative order as in E_s), followed by X', X, Y' and Y in any order as long as a write is not placed between the read and write of a read-modify-write and program order between the read and write of a read-modify-write is preserved. Further, there are no unessential operations until before all of X, X', Y, and Y', and all operations until and including X, X', Y, and Y' are distinguished as in E_s.

Proof:

Consider a modification of the execution order of E_s where the operations in O are moved to just before X, retaining their original order, and then X' (if X is a write of a read-modify-write) is moved to just before X. Consider the resulting order until before X' and X. This resulting order is still consistent with program order, and the last conflicting write before any read is still the same as with the original execution order of E_s. Now consider the resulting order appended with X', X, Y' and Y (in any order that preserves the constraints specified by the lemma) and make the appended reads return the value of the last conflicting write before them. It follows that the resulting order is the prefix of an execution order of some sequentially consistent execution where the operations of the processors of X and Y preceding X and Y are the same as for E_s, and X and Y access the same addresses as before. Consider one such sequentially consistent execution E. E has the operations required of E_s'. Since all operations before X,X',Y,Y' return the same values as in E_s, none of them are unessential in E. From the assumptions of how operations are distinguished (Condition 7.18 and Section 5.13) and by the above observations, it follows that all operations until and including X', X, Y', and Y must be distinguished the same way in E and E_s. Thus, we have proved the existence of an execution that satisfies all the required properties of E_s'. □

When applying lemma G.5 in the appendix, we refer to X, X', Y, and Y' as used by the lemma as *appended operations.*

Lemma G.6: Consider two conflicting operations X and Y in E_s such that there is no other conflicting write between X and Y in E_s and there is a race path from X to Y in E_s (as in definition 7.4). Then $X \xrightarrow{vco1} Y$ in E_s.

Proof:

Consider E_s' as defined in the lemma G.5 with X before Y. Then since there are no writes between X and Y in E_s, it follows that the reads from the appended operations return the same values in E_s and E_s'. Therefore, all operations until Y are essential in E_s'. Since there is a race path from X to Y in E_s, it follows that if X is part of a read-modify-write, X must be the write. Further, if Y is part of a read-modify-write, then Y must be the read. Thus, X and Y form a race in E_s'. Therefore, from lemma G.3, it follows that $X \xrightarrow{vco1} Y$ in E_s' and therefore, $X \xrightarrow{vco1} Y$ in E_s, proving the lemma. □

Lemma G.7: Consider two conflicting operations X and Y in E_s such that there is a race path from X to Y in E_s (as in definition 7.4). Then $X \xrightarrow{vco1} Y$ in E_s.

Proof:

If there is no other conflicting write between X and Y in the execution order of E_s, then the lemma follows from lemma G.6. Therefore, assume that there are conflicting writes between X and Y in the execution order of E_s. For any such write, there must be a race path from X to the write and from the write to Y in E_s. Let W_1 be the first conflicting write ordered after X by the execution order of E_s. Let W_2 be the first conflicting write ordered before Y by the execution order of E_s. (W_1 may or may not be the same as W_2.) From lemma G.6, $X \xrightarrow{vco1} W_1$ and $W_2 \xrightarrow{vco1} Y$ in E_s. It follows that $X \xrightarrow{vco1} Y$ in E_s. □

We next prove theorem G.2 is correct. We do this by considering every pair of consecutive conflicting operations, X and Y, in E_s and showing that one of the candidates for a critical path for every pair (when a critical path is necessary) is a path of type (1), (2), (3), or (4). If X and Y are from the same processor, then the $\xrightarrow{po}$ arc is a critical path that is of type (1), (2), or (3). Therefore, assume that X and Y are from different processors. Without loss of generality, assume $X \xrightarrow{co} Y$. Then there are the following cases.

Case 1: At least one of X or Y is data.

In this case, there must be a $\xrightarrow{hb1}$ path from X to Y. There is always an $\xrightarrow{hb1}$ path where all the $\xrightarrow{so1}$ arcs are between operations of different processors and no two adjacent arcs are $\xrightarrow{po}$ arcs. Then on this path, the $\xrightarrow{so1}$ arcs are $\xrightarrow{vco3}$ arcs. Therefore, the path is of type (3).

Case 2: Both X and Y are synchronization, Y is not from a synchronization loop.

There are two sub-cases.

Sub-case 2a: X and Y are both NLU.

A path corresponding to X and Y needs to be critical only if there is an ordering path from X to Y. So suppose there is an ordering path from X to Y. There must be an ordering path where the $\xrightarrow{co}$ arcs are race paths and no two adjacent arcs are both $\xrightarrow{po}$. The $\xrightarrow{co}$ arcs on this path are $\xrightarrow{vco1}$ arcs by lemma G.7. Therefore, this path is of type (1).

Sub-case 2b: At least one of X or Y is a loop operation.

If there is a path in *rel* from X to Y that ends in a $\xrightarrow{po}$ arc, then by lemma G.4, there is an ordering path of type (2) from X to Y that can be chosen as critical. Therefore, assume that there is no path in *rel* from X to Y that ends in a $\xrightarrow{po}$ arc. There cannot be any other write W between X and Y in E_s such that W conflicts with X and Y and has a path in *rel* to Y that ends in a $\xrightarrow{po}$ arc (otherwise, X and Y are not consecutive conflicting operations). Consider E_s' defined in lemma G.5 with Y after X. Then X' and X (if either is a read) return the same value as E_s and so are essential. Since Y is not from a synchronization loop, Y is also essential. Further, note that if Y is a write from a read-modify-write and X is a write, then the read of the read-modify-write cannot be between X and Y in E_s because then there is a *rel* path from X to Y that ends in $\xrightarrow{po}$ arc. (Also X cannot be a read from a read-modify-write because then X and Y are not consecutive conflicting operations in E_s.) Therefore, we can have an E_s' where X and Y are adjacent in the execution order and so form a race. From lemma G.3, $X \xrightarrow{vco1} Y$ in E_s'. Therefore, Y cannot be LW. Therefore, X must be a loop operation. The only possibility is for X to be LW and Y to be NLUR. But X must be an intrinsic loop operation and Y is not an intrinsic loop operation. Therefore, X cannot race with Y, a contradiction.

Case 3: Both X and Y are synchronization, Y is an exit write of a synchronization loop.

Let Y' be the read of Y's read-modify-write. Note that there cannot be another write between X and Y in E_s that conflicts with Y since then X and Y are not consecutive conflicting operations. There are two sub-cases.

Sub-case 3a: X is a write.

Either Y' is data or synchronization. Suppose first that Y' is a data operation. Then there is a $\xrightarrow{\text{hb1}}$ path from X to Y' in E_s and therefore a $\xrightarrow{\text{hb1}}$ path from X to Y in E_s. This implies a path of type (2) from X to Y in E_s.

Next suppose that Y' is a synchronization operation. Y' must return the value of X in E_s. It follows that $X \xrightarrow{\text{co}} Y' \xrightarrow{\text{po}} Y$ is a path of type (2) in E_s.

Sub-case 3b: X is a read.

If X is part of a read-modify-write, then the corresponding write must be before Y'. But then X and Y are not consecutive conflicting operations. Therefore, X is not part of a read-modify-write. Consider $E_s{}'$ of lemma G.5 with Y after X and Y' before X. X, Y' and Y must be essential and X and Y form a race in $E_s{}'$. By lemma G.3, $X \xrightarrow{\text{vco1}} Y$ in $E_s{}'$. Then both X and Y must be NLU. A path corresponding to X and Y need be critical in E_s only if there is an ordering path from X to Y in E_s. So suppose there is an ordering path from X to Y in E_s. There must be some path where the $\xrightarrow{\text{co}}$ arcs are race paths and no two adjacent arcs are both $\xrightarrow{\text{po}}$. Then the $\xrightarrow{\text{co}}$ arcs are $\xrightarrow{\text{vco1}}$ arcs by lemma G.7. Therefore, the path is of type (1).

Case 4: Both X and Y are synchronization, Y is an exit read of a synchronization loop.

Y is from a self-ordered loop. Let W_s be the write in the definition of critical paths such that a path after a write after W_s to Y is a candidate for a critical path. (We use the extended definition in Appendix C.) If there needs to be a critical path, there must be such a W_s and there must be an ordering path from a write after W_s to Y that ends in a $\xrightarrow{\text{po}}$ arc. If there is a *rel* path that ends in a program order arc from some write after W_s to Y, then by lemma G.4, the theorem follows for this case. So for a contradiction, assume that there is no *rel* path that ends in a program order arc from any write after W_s to Y. Then let W_u be the first write after W_s. Consider $E_s{}'$ assuming W_u to be X. There are three sub-cases.

Sub-case 4a: Y is not part of a read-modify-write.

Consider $E_s{}'$ with Y just before W_u. Y must be essential since W_s writes an exit value for Y. W_u must also be essential since even if it is part of a synchronization loop, its read returns the same value as in E_s. But Y forms a race with W_u and W_u is not necessary to make Y essential. Therefore, Y must be NLU and W_u must be NLU also. In E_s, there is an ordering path from W_u to Y that ends in a program order arc. One such path must be such that the $\xrightarrow{\text{co}}$ arcs are race paths and no two adjacent arcs are both $\xrightarrow{\text{po}}$. Then the $\xrightarrow{\text{co}}$ arcs are $\xrightarrow{\text{vco1}}$ arcs by lemma G.7. Therefore, the path is of type (1) and can be selected as the critical path corresponding to X and Y.

Sub-case 4b: Y is part of a read-modify-write, W_u is not part of a read-modify-write.

Consider $E_s{}'$ with Y before W_u. Again, Y must be essential. W_u is always essential. Then W_u forms a race with Y's exit write and does not terminate Y. So Y must be NLU and W_u must be NLU. As for the previous sub-case, in E_s, there is an ordering path from W_u to Y that ends in a program order arc and is of type (1). This path can be selected as the critical path corresponding to X and Y.

Sub-case 4c: Y is part of a read-modify-write, W_u is part of a read-modify-write.

Consider $E_s{}'$ with Y before W_u. Again, Y must be essential. There are two cases depending on whether W_u is essential in $E_s{}'$.

Sub-case 4c1: W_u is essential in $E_s{}'$.

The read of W_u's read-modify-write forms a race in $E_s{}'$. If this read is not from a synchronization loop, then this read is NLU. If this read is from a synchronization loop, then since both W_s and the write of Y's read-modify-write make W_u's loop terminate in $E_s{}'$, the read of W_u's read-modify-write must be NLU. Thus, in all cases, the read of W_u's read-modify-write is NLU. Since the write of Y's read-modify-write must form a race with the read of W_u's read-modify-write, it follows that the write of Y's read-modify-write must be NLU. In E_s, there is an ordering path from W_u to Y that ends in a program order arc. One such path must be such that the $\xrightarrow{\text{co}}$ arcs are race paths and no two adjacent arcs are both $\xrightarrow{\text{po}}$. Then the $\xrightarrow{\text{co}}$ arcs are

$\xrightarrow{\text{vco1}}$ arcs by lemma G.7. Therefore, the path is of type (4) and can be selected as the critical path corresponding to *X* and *Y*.

Sub-case 4c2: W_u is unessential in E_s'.

W_u must be from a synchronization loop and the exit read of the loop must form a race in some sequentially consistent execution. Therefore, W_u must write an exit value for *Y* (from the extended definition of critical sets in appendix C).

Consider E_s' with W_u before *Y*. Both W_u and *Y* must be essential in E_s'. Both W_s and W_u write exit values for *Y* in E_s'; therefore, *Y* must be NLU. W_u forms a race with *Y* in E_s'. Therefore, W_u must be NLU. Again, from arguments used in previous sub-cases, there exists an ordering path from W_u to *Y* of type (1) and that ends in a program order arc in E_s. This path can be chosen as critical.

This completes the proof. □

G.2 The PLpc2 Model

The following theorem gives a set of valid paths for PLpc2. These are the same as the PLpc1 model when the operations at the end-points are both atomic or NLU. If the operations are non-atomic, then a more constrained type of paths can be specified as follows. (The notation used is the same as for PLpc1 in Section G.1; the modification to the notion of paired discussed in Section G.1 is required for PLpc2 also.)

Theorem G.8: *Valid paths of PLpc2:* The following constitute a set of valid paths for PLpc2.

(1) X ($\xrightarrow{\text{po}} \cup \xrightarrow{\text{vco1}}$)+ Y, X and Y are non-loops or unpaired synchronization.

(2) X ($\xrightarrow{\text{po}} \cup \xrightarrow{\text{vco2}}$)+ Y, both *X*, *Y* are synchronization and atomic, at least one is loop.

(3) X ($\xrightarrow{\text{po}} \cup \xrightarrow{\text{vco3}}$)+ Y, at least one of X or Y is data, both are atomic.

(4) X $\xrightarrow{\text{po}}$ A ($\xrightarrow{\text{po}} \cup \xrightarrow{\text{vco2}}$)+ B $\xrightarrow{\text{po}}$ Y, both *X*, *Y* are synchronization, at least one is non-atomic.

(5) X $\xrightarrow{\text{po}}$ A ($\xrightarrow{\text{po}} \cup \xrightarrow{\text{vco3}}$)+ B $\xrightarrow{\text{po}}$ Y, at least one of X or Y is data, at least one is non-atomic.

(6) SW $\xrightarrow{\text{vco2}}$ SR $\xrightarrow{\text{po}}$ S, at least one of SW or S is non-atomic.

(7) SW $\xrightarrow{\text{vco3}}$ SR $\xrightarrow{\text{po}}$ D, at least one of SW or D is non-atomic.

(8) X ($\xrightarrow{\text{po}} \cup \xrightarrow{\text{vco1}}$)+ Y, X is a write from a read-modify-write whose read is non-loop or unpaired synchronization, and Y is a read from a read-modify-write whose write is non-loop or unpaired synchronization.

(9) X $\xrightarrow{\text{po}}$ Y.

Further, in all of the paths, no two consecutive edges are both $\xrightarrow{\text{po}}$ or $\xrightarrow{\text{vco}}$.

By inspection, it follows that the $\xrightarrow{\text{vpo}}$ arcs for PLpc2 are the same as for PLpc1. Further, a write distinguished as non-atomic is never a receiver, and never starts a valid path with *W* $\xrightarrow{\text{co}}$ *R* unless it is a path of type (6), (7), or (8). For paths of type (6) and (7), cache coherence and preserving intra-processor dependences suffices; write atomicity (as discussed in Chapter 6) is not needed. Paths of type (8) are executed safely if paths of type (1) are executed safely. Thus, writes distinguished as non-atomic can be executed non-atomically, as long as they are coherent with respect to other writes and intra-processor dependences are maintained.

Further, as mentioned in Chapter 6, if all atomic, unpairable, and non-loop reads are converted into read-modify-writes, then all writes can be executed non-atomically (as long as all synchronization writes are coherent with respect to other conflicting writes). Appendix H uses this conversion to show how PLpc2 programs can be run correctly on release consistency (RCpc) and processor consistency systems; the proof in Appendix H can be easily adapted to show the result is true in general. Essentially, the proof consists of a simple case analysis of the valid paths of type (1) - (3) (which require write atomicity) and shows that the above conversion allows replacing some of the paths of type (1)-(3) by paths that do not require write atomicity, and the conversion ensures that the remaining paths of type (1)-(3) are executed safely without write atomicity but because of the properties of a read-modify-write.

Finally, from the proof below, it follows that as for PLpc1, synchronization reads from synchronization loops can be self-ordered with respect to synchronization writes. For PLpc2, additionally, data reads can also be self-ordered with respect to synchronization writes where at least one of the read or write is non-atomic.[47]

We next prove that theorem G.8 is correct. We do this by considering a PLpc2 program *Prog* and a sequentially consistent execution E_s (without unessentials) of program *Prog*. We consider every pair of consecutive conflicting operations, *X* and *Y*, in E_s and show that one of the candidates for a critical path for every pair (if a critical path is necessary for that pair) is a path from type (1)-(9) described above. Without loss of generality, assume $X \xrightarrow{co} Y$ in E_s. The PLpc1 analysis applies directly in the cases where (i) *X* and *Y* are from the same processor, or (ii) when both *X* and *Y* are atomic or NLU, and at least one is data, or (iii) when both *X* and *Y* are atomic or NLU, both are synchronization, and *Y* is not a self-ordered read. Therefore, below we consider only cases when *X* and *Y* are from different processors. Further, either at least one of *X* or *Y* is non-atomic (under PLpc2), or both *X* and *Y* are synchronization and *Y* is a self-ordered read.

Case 1: At least one of *X* or *Y* is data, and *X* and *Y* do not form a partial race.

> Then there is a $\xrightarrow{hb1}$ path from *X* to *Y* such that it begins and ends with a $\xrightarrow{po}$ arc or consists only of operations to the same location. The former case implies a path of type (5). In the latter case, the path must be of the type $X = \text{SW} \xrightarrow{vco3} \text{SR} \xrightarrow{po} Y$ (since *X* and *Y* are consecutive conflicting operations). Thus, the path is of type (7).

Case 2: At least one of *X* or *Y* is data, and *X* and *Y* form a partial race.

> Since at least one of *X* or *Y* is non-atomic and they form a partial race, *Y* must be an exit read from a synchronization loop and must be an intrinsic non-atomic read. This loop is also self-ordered. Consider the last write *W* before *X* that conflicts with *X* in E_s (including the hypothetical initial write). Then by definition of a non-atomic read, *W* must write a non-exit value for *Y*. If *W* is the hypothetical initial write, then there is no need for a critical path corresponding to *X* and *Y*, proving the theorem for this case. Therefore, assume that *W* is not the hypothetical initial write. By definition of a non-atomic read, *W* and *Y* cannot form a partial race in E_s. There are three sub-cases.
>
> *Sub-case 2a:* At least one of *W* or *Y* is data.
>
> > There must be a $\xrightarrow{hb1}$ path from *W* to *Y* in E_s that satisfies the conditions of definition 6.11 of not forming a partial race. Since at least one of *X* or *Y* is data, this path must end in a program order arc. Thus, the path is either of type (5) or type (7) and can be considered as the critical path corresponding to *X* and *Y*.
>
> *Sub-case 2b:* Both *W* and *Y* are synchronization and there is a path in *rel* from *W* to *Y* that begins and ends in a $\xrightarrow{po}$ arc.
>
> > By argument similar to that for lemma G.4, the above *rel* path implies an ordering path from *W* to *Y* of type (4) and can be considered as the critical path corresponding to *X* and *Y*.
>
> *Sub-case 2c:* Both *W* and *Y* are synchronization and there is no path in *rel* from *W* to *Y* that begins and ends in a $\xrightarrow{po}$ arc.
>
> > Let *O* be the set of operations that are after *W* and have a *rel* path to *Y* in E_s such that the path ends in $\xrightarrow{po}$. *X* must be in *O* because *X* must have a $\xrightarrow{hb1}$ path to *Y*. Consider a modification of the execu-

[47] This is in contrast to the PLpc model where all self-ordered reads are considered competing [AGG93]. (Recall that competing operations in PLpc are otherwise analogous to synchronization operations in PLpc2.) This affects aggressive implementations of the control requirement since now all reads in synchronization loops need to be assumed to be self-ordered. This suggests further distinguishing data operations that are involved in a partial race from those that are not; only the former are self-ordered.

tion order of E_s where the operations in *O* and *Y* are moved to just after *W*, retaining their relative order. The resulting order until and including *Y* is still consistent with program order, and the last conflicting write before any read until and including *Y* is still the same as with the original execution order. It follows that the resulting order until *Y* is the prefix of an execution order of some sequentially consistent execution where the operations until *Y* are distinguished as in E_s and are all essential. *Y* must form a partial race with *X* in the new execution as well. Since at least one of *X* or *Y* is distinguished as non-atomic and *X* and *Y* form a partial race, *Y* must be intrinsic non-atomic (by definition 6.12 of non-atomic reads) for the new execution. Therefore, *Y* cannot form a partial race with *W* in the new execution. There cannot be a path in the program/semi-causal-conflict graph from *W* to *Y* that begins with a $\xrightarrow{\text{po}}$ arc. Therefore, there must be a path in the program/semi-causal-conflict graph from *W* to *Y* that consists of operations to the same location and is of the type *W* = SW $\xrightarrow{\text{vco2}}$ SR $\xrightarrow{\text{po}}$ *Y*. This path is also present in E_s. It is of type (6) and can be considered as the critical path corresponding to *X* and *Y*.

Case 3: Both *X* and *Y* are synchronization and there is a path in *rel* from *X* to *Y* that begins and ends in a $\xrightarrow{\text{po}}$ arc.

By argument similar to that for lemma G.4, the *rel* path described above implies an ordering path from *X* to *Y* of type (4). This path can be considered critical.

Case 4: Both *X* and *Y* are synchronization, there is no path in *rel* from *X* to *Y* that begins and ends in a $\xrightarrow{\text{po}}$ arc, and *Y* is not an exit read from a synchronization loop.

Note that *X* cannot be a read from a read-modify-write since then *X* and *Y* are not consecutive, and there cannot be any other conflicting write between *X* and *Y*. Let *O* be the set of operations that are after *X* and have a *rel* path to *Y* in E_s such that the path ends in $\xrightarrow{\text{po}}$. Consider a modification of the execution order of E_s where the operations in *O* and *Y* are moved to just after *X*, retaining their relative order. The resulting order until *Y* is still consistent with program order, and the last conflicting write before any read until *Y* is still the same as with the original execution order. It follows that the resulting order until *Y* is the prefix of an execution order of some sequentially consistent execution where the operations until *Y* are distinguished as in E_s and are all essential. Call the new execution E_s'. There are two sub-cases.

Sub-case 4a: *X* and *Y* do not form a partial race in E_s'.

There cannot be a path in the program/semi-causal-conflict graph from *X* to *Y* that begins with a $\xrightarrow{\text{po}}$ arc. Therefore, there must be a path in the program/semi-causal-conflict graph from *X* to *Y* that consists of operations to the same location and is also present in E_s, and is of the type *X* = SW $\xrightarrow{\text{vco2}}$ *SR* $\xrightarrow{\text{po}}$ *Y*. This is a path of type (6), and can be considered critical.

Sub-case 4b: *X* and *Y* form a partial race in E_s'.

Since either one of *X* or *Y* is non-atomic or *Y* is a self-ordered read, it follows that *Y* must be an exit read from a synchronization loop, a contradiction.

Case 5: Both *X* and *Y* are synchronization, there is no path in *rel* from *X* to *Y* that begins and ends in a $\xrightarrow{\text{po}}$ arc, *Y* is an exit read from a synchronization loop, and *X* and *Y* form a partial race in E_s.

If there is no critical path required for *X* and *Y*, the theorem is proved for this case. So assume that a critical path is required for *X* and *Y*.

Let W_c be the first conflicting write in E_s from which there can be a critical path to *Y* corresponding to *X* and *Y* (we use the extended definition of a critical set in Appendix C). Then the last conflicting write W_s (including the hypothetical initial write) before W_c writes an exit value for *Y*. Let W_u be any conflicting write in E_s between W_c and *X* and including W_c and *X*. Let *Y′* be the first operation before *Y* by $\xrightarrow{\text{po}}$.

Sub-case 5a: There does not exist a *rel* path from any W_u to *Y′*.

Since the program is a PLpc1 program as well and a critical path is required for *X* and *Y*, it follows that the path corresponding to PLpc1 is of type (1) or (8). That path can be considered critical for PLpc2 as well.

Sub-case 5b: There exists a *rel* path from some W_u to Y'.

Consider the last such W_u. Let W_u' be the first operation after W_u by program order in E_s. There cannot be a *rel* path from W_u' to Y' in E_s because this implies a path of type (4) from W_u to Y in E_s (by arguments similar to lemma G.4), a contradiction. Let O be the set of operations that are after W_u and have a *rel* path to Y' in E_s. O cannot contain any write conflicting with Y. Consider a modification of the execution order of E_s where the operations in O and Y' are moved to just after W_u, retaining their relative order. The resulting order until Y' is still consistent with program order, and the last conflicting write before any read is still the same as with the original execution order. Thus, it represents the prefix of the execution order of a sequentially consistent execution where all operations until Y' and Y are distinguished similar to E_s and all operations until Y' are essential. Now extend the resulting order to represent a sequentially consistent execution where Y is also essential, Y returns the value of the first write after W_u that writes its exit value, and Y occurs immediately after this write. Note that the *rel* path from W_u to Y' also exists in the new execution. The following sub-cases are possible for the new execution.

Sub-case 5b1: At least one of W_u or Y is intrinsically non-atomic (by definitions 6.12 and 6.13) in the new execution.

Since there is a *rel* path from W_u to Y that ends in a $\xrightarrow{\text{po}}$ arc, by definitions 6.12 and 6.13, there must be a path from W_u to Y of the type described in the definition of a partial race and such that it ends in a $\xrightarrow{\text{po}}$ arc. Such a path must be of type (6) and qualifies as the required critical path.

Sub-case 5b2: Both W_u and Y are intrinsically atomic (by definitions 6.12 and 6.13) in the new execution.

The *rel* path from W_u to Y is of type (1), (2), or (3) and qualifies as the required critical path.

Case 6: Both X and Y are synchronization, there is no path in *rel* from X to Y that begins and ends in a $\xrightarrow{\text{po}}$ arc, Y is an exit read from a synchronization loop, and X and Y do not form a partial race in E_s.

There cannot be a conflicting write between X and Y; otherwise, X and Y are not consecutive. Let O be the set of operations that are after X and have a *rel* path to Y in E_s that ends in a $\xrightarrow{\text{po}}$ arc. Consider a modification of the execution order of E_s where the operations in O and Y are moved to just after X, retaining their relative order. Consider the resulting order until Y. This resulting order is still consistent with program order, and the last conflicting write before any read is still the same as with the original execution order. It follows that the resulting order until Y is the prefix of an execution order of some sequentially consistent execution where the operations until Y are distinguished similar to E_s and are essential.

Sub-case 6a: X and Y do not form a partial race in the new execution.

There must be a path of the type $X = \text{SW} \xrightarrow{\text{vco2}} \text{SR} \xrightarrow{\text{po}} Y$ in the new execution. This path is also present in the old execution. It is of type (6) and can be considered the critical path for X and Y.

Sub-case 6b: X and Y form a partial race in the new execution.

Then applying analysis similar to Case 5, there must be a write W_u before X such that a path from W_u to Y is of type (1) - (9) and can be chosen as the critical path for the new execution. This path must also exist in the old execution.

This completes the proof. □

Appendix H: Porting PLpc1 and PLpc2 Programs to Hardware-Centric Models

This appendix proves the correctness of the mappings discussed in Chapter 6 for porting PLpc1 and PLpc2 programs to systems based on hardware-centric models.

For the hardware-centric models, we use the specifications developed in [GAG93]. These specifications use the extension of Collier's abstraction discussed in Section 7.6.5, and have been shown to be equivalent to the original specifications [GAG93]. Section H.1 discusses the extended abstraction and gives the specifications using this abstraction. It then shows how to derive specifications in Collier's original abstraction that are more aggressive than the above specifications (i.e., the derived specifications allow a superset of the executions than the corresponding hardware-centric models). Section H.2 shows that the derived aggressive specifications obey the valid path and control requirements for PLpc1 with the mappings of Chapter 6, thereby proving that the hardware-centric models obey the necessary requirements. Section H.3 repeats the above for PLpc2. The material in Sections H.2 and H.3 is very similar to that for the corresponding proofs for the PLpc model [AGG93].

H.1 Specifications of Hardware-Centric Models

The extension to Collier's abstraction [GAG93] explicitly models the equivalent of a write buffer in a processor. A write operation now involves an additional initiation sub-operation called Winit that can be viewed as the write being placed in the write buffer of its processor. Informally, a read returns the value of the last conflicting write placed in its processor's write buffer if such a write exists; if there is no such write, then the read returns the value in its processor's memory copy. More formally, as mentioned in [GAG93], ‘‘a read sub-operation R(i) by processor 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 Winit(i) $\xrightarrow{xo}$ R(i) and R(i) $\xrightarrow{xo}$ W(i), then R(i) returns the value of the last such Winit(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 $\xrightarrow{xo}$.’’

The specifications of the various models we are concerned with are given in figures H.1 - H.4, taken directly from [GAG93]. The notation is similar to that used in the rest of this thesis. Some differences are: RW is used to indicate a read or a write, AR and AW indicate a read and write respectively from a read-modify-write, RMW indicates a read-modify-write, Wc, Rc, RWc indicate competing operations, Rc_acq and Wc_rel indicate an acquire and a release respectively. The $\xrightarrow{rch}$ relation for RCpc will be discussed in Section H.3. All specifications require certain ordering paths to be executed safely,[48] a condition for read-modify-write, a coherence condition for certain writes, and a termination condition for certain sub-operations of certain writes (which requires the specific sub-operations to be in the execution). In addition, they also require an initiation condition which essentially ensures that initiation sub-operations of a processor appear in program order with respect to other conflicting sub-operations of the same processor. More formally, as mentioned in [GAG93], the condition requires: ‘‘If R $\xrightarrow{po}$ W, then R(i) $\xrightarrow{xo}$ Winit(i). If W $\xrightarrow{po}$ R, then Winit(i) $\xrightarrow{xo}$ R(i). If W1 $\xrightarrow{po}$ W2, then W1init(i) $\xrightarrow{xo}$ W2init(i).’’ The specifications also implicitly assume the low-level finite speculation requirement similar to ours (Condition 7.21(2)).

> **Theorem H.1:** Consider a specification given in the extended Collier's abstraction which only requires safe execution of certain ordering paths, safe execution of certain $\xrightarrow{co}$ paths, termination of certain writes, the finite speculation requirement for some appropriate control relation, and the initiation condition. Consider a corresponding specification given in Collier's abstraction which differs from the above specification only in requiring that (1) in the ordering paths that need to be executed safely, W $\xrightarrow{co}$ R arcs should involve only reads that return the value of another processor's write, (2) a path that begins with a read that returns its own processor's write is not required to be executed

[48] An ordering path from operation X to operation Y is executed safely if $X(i) \xrightarrow{xo} Y(i)$ for all i.

Define $\xrightarrow{spo}$ as follows: X $\xrightarrow{spo}$ Y if X,Y are the first and last operations in one of

R $\xrightarrow{po}$ RW

W $\xrightarrow{po}$ W

AW (in RMW) $\xrightarrow{po}$ R

W $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R

Define $\xrightarrow{sco}$ as follows: X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

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

Define $\xrightarrow{sxo}$ as follows: X $\xrightarrow{sxo}$ 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 $\xrightarrow{spo}$ R

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

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

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then

either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Termination condition holds for all sub-operations.

Figure H.1. Specification of Total Store Ordering

safely (unless the path is between operations by the same processor), and (3) if $W \xrightarrow{po} R$, then $W(i) \xrightarrow{xo} R(i)$ for all i. Then the result of an execution that obeys the specification in the extended abstraction is the same as the result of an execution that obeys a specification in Collier's abstraction.

Proof:

Let E be an execution of the specification in the extended abstraction. We construct an execution E' below that has the same result as E but obeys the corresponding specification in Collier's abstraction described above.

For every W $\xrightarrow{po}$ R such that R(i) $\xrightarrow{co}$ W(i) in E and W(i) is the last such write in E (by $\xrightarrow{xo}$), move R(i) to just below W(i) in the execution order of E.

Call the result of the above conversion as the new order, and the $\xrightarrow{xo}$ of E as the old order.

The only difference in the $\xrightarrow{co}$ of the old and new orders is that a R $\xrightarrow{co}$ W1 in the old order can be W1 $\xrightarrow{co}$ R in the new order, if R returns the value of its own processor's write, W, in E and either R $\xrightarrow{co}$ W1

Define $\xrightarrow{spo}$ as follows: 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

AW (in RMW) $\xrightarrow{po}$ RW

W $\xrightarrow{po}$ STBAR $\xrightarrow{po}$ RMW $\xrightarrow{po}$ R

Define $\xrightarrow{sco}$ as follows: X $\xrightarrow{sco}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{co}$ Y

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

Define $\xrightarrow{sxo}$ as follows: X $\xrightarrow{sxo}$ 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 $\xrightarrow{spo}$ R

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

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

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then

either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Termination condition holds for all sub-operations.

Figure H.2. Specification of Partial Store Ordering

$\xrightarrow{co}$ W in E or W1 = W.

Now in the new order, every read returns the value of the last conflicting write before it by the execution order and this value is the same as for E. Thus, the new order is an execution order using Collier's abstraction with the same result as E. Call the corresponding execution E'.

Further, all ordering paths in E' that have to be executed safely by the specification with Collier's abstraction also had to be executed safely by the specification with the extended abstraction. These paths are executed safely in E' as well unless it is required to safely execute a path from a read R to write W2 where R and W2 are from different processors and R returned the value of its own processor's write. However, by assumption, paths of the above type are not required to be executed safely. Thus, all the required ordering paths are executed safely in E'. All the other aspects of the specification in Collier's abstraction are trivially obeyed by E', proving the theorem. □

Define $\xrightarrow{spo}$ as follows: X $\xrightarrow{spo}$ Y if X,Y are the first and last operations in one of

R $\xrightarrow{po}$ RW

W $\xrightarrow{po}$ W

Define $\xrightarrow{sco}$ as follows: X $\xrightarrow{sco}$ Y if X $\xrightarrow{co}$ Y

Define $\xrightarrow{sxo}$ as follows: X $\xrightarrow{sxo}$ 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 $\xrightarrow{spo}$ R

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

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then

either W $\xrightarrow{sxo}$ AR or AW $\xrightarrow{sxo}$ W

Conditions on $\xrightarrow{xo}$:

Initiation condition holds.

$\xrightarrow{sxo}$ condition: if X $\xrightarrow{sxo}$ Y, then X(i) $\xrightarrow{xo}$ Y(i) for all i.

Termination condition holds for all sub-operations.

Figure H.3. Specification of Processor Consistency.

H.2 Porting PLpc1 Programs to SPARC V8 Systems

Below, we give the correctness proof for the mappings for total store ordering (TSO). The proof for partial store ordering is almost identical.

By inspection, it follows that TSO executes the following ordering paths safely. Recall again that an ordering path from operation *X* to operation *Y* is executed safely if $X(i) \xrightarrow{xo} Y(i)$ for all *i*. (Below, $\xrightarrow{spo}$ and $\xrightarrow{sco}$ are the corresponding relations defined for TSO.)

(1) X ($\xrightarrow{spo}$ ∪ $\xrightarrow{sco}$)+ Y where no two adjacent arcs are both $\xrightarrow{sco}$ or both $\xrightarrow{spo}$, and for any *W* $\xrightarrow{sco}$ *R* arc, *R* returns the value of another processor's write.

(2) $X \xrightarrow{po} Y$.

We refer to the above paths as the safe ordering paths of TSO. Note that a path of type (4) given in Theorem G.2 is executed safely as long as a path of type (1) of theorem G.2 is executed safely; we therefore need not consider paths of type (4) when considering the valid path requirement below.

We first consider the mapping of converting a NLUW into a read-modify-write when NLUW $\xrightarrow{po}$ NLUR. By inspection, this results in the $\xrightarrow{spo}$ of TSO being a superset of the $\xrightarrow{vpo}$ of PLpc1 (with the appropriate mapping). Further, $\xrightarrow{sco}$ of TSO is a superset of the $\xrightarrow{vco}$ relations of PLpc1. It follows that the safe ordering paths of TSO are a superset of the valid paths of PLpc1. Thus, TSO obeys the valid path requirement of PLpc1. It trivially obeys the control requirement because of the following. The critical set requirement is trivially obeyed because the control paths are safe ordering paths of TSO. The finite speculation, write termination, and loop coherence requirements are obeyed directly. Thus, with the above mapping, a TSO system guarantees sequential consistency to PLpc1 programs.

Define $\xrightarrow{\text{spo}}$ as follows: X $\xrightarrow{\text{spo}}$ Y if X,Y are the first and last operations in one of

Rc $\xrightarrow{\text{po}}$ RWc

Wc $\xrightarrow{\text{po}}$ Wc

RW $\xrightarrow{\text{po}}$ Wc_rel

Rc_acq $\xrightarrow{\text{po}}$ RW

X $\xrightarrow{\text{spo}}$ Y if X { $\xrightarrow{\text{rch}}$ $\cup$ $\xrightarrow{\text{spo}}$ }+ Y

Define $\xrightarrow{\text{sco}}$ as follows: X $\xrightarrow{\text{sco}}$ Y if X,Y are the first and last operations in one of

X $\xrightarrow{\text{co}}$ Y

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

Define $\xrightarrow{\text{sxo}}$ as follows: X $\xrightarrow{\text{sxo}}$ Y if X and Y conflict and X,Y are the first and last operations in one of

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

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

multiprocessor dependence chain: one of

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

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

atomic read-modify-write (AR,AW): if W conflicts with AR,AW, then

either W $\xrightarrow{\text{sxo}}$ AR or AW $\xrightarrow{\text{sxo}}$ W

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

Initiation condition holds.

$\xrightarrow{\text{sxo}}$ condition: if X $\xrightarrow{\text{sxo}}$ Y, then X(i) $\xrightarrow{\text{xo}}$ Y(i) for all i.

Termination condition holds for all sub-operations.

Figure H.4. Specification of Release Consistency (RCpc).

We next consider the mapping of converting a NLUR into a read-modify-write when NLUW $\xrightarrow{\text{po}}$ NLUR. The only missing $\xrightarrow{\text{vpo}}$ of PLpc1 from TSO's $\xrightarrow{\text{spo}}$ is that of NLUW $\xrightarrow{\text{po}}$ NLUR. Consider a valid path of PLpc1 that uses such a $\xrightarrow{\text{vpo}}$. If the above type of $\xrightarrow{\text{vpo}}$ arc is not the last arc on the path, then the next arc is of the type NLUR $\xrightarrow{\text{co}}$ W1. By replacing NLUR by a read-modify-write (RMW), we can replace the NLUW $\xrightarrow{\text{po}}$ NLUR $\xrightarrow{\text{co}}$ W1 sequence of the original path by a NLUW $\xrightarrow{\text{po}}$ W $\xrightarrow{\text{co}}$ W1. Such programs do not require valid paths with NLUW $\xrightarrow{\text{po}}$ NLUR arcs except at the end of the path. Consider a path that does have the above arc at the end. Then the path must begin with a write W1. Since the NLUR is replaced by a read-modify-write, it follows that there is a safe ordering path from W1 to the write W2 of this read-modify-write. Thus, W1(i) $\xrightarrow{\text{xo}}$ W2(i) for all i. But we also know that for the read R of the read-modify-write, R(i) $\xrightarrow{\text{xo}}$ W2(i) for all i. Thus, it follows that W1(i) $\xrightarrow{\text{xo}}$ R(i) for all i. Thus, the above path is executed safely and can be considered as a safe ordering path of TSO. Thus, TSO obeys the valid path requirement for PLpc1 with the above mapping. The argument for the control requirement is the same as for the earlier mapping. Thus, the above mapping is correct.

H.3 Porting PLpc2 Programs to Processor Consistency and RCpc Systems

Below, we give the correctness proof for the mapping for RCpc. The arguments for the mapping for processor consistency are a subset of the arguments for RCpc.

The mapping requires converting all operations distinguished as data to ordinary, all operations distinguished as unpairable to nsyncs, all operations distinguished as pairable to syncs, and all reads distinguished as unpairable, non-loop, or atomic reads to read-modify-writes. For the conversion to read-modify-write, the write of the read-modify-write for synchronization reads should be either sync or nsync (the read should be as discussed above).

By inspection, the above mapping ensures that RCpc executes valid paths of types (4), (5), (6), (7), and (9) in theorem G.8 safely. Further, paths of type (8) are executed safely as long as paths of type (1) are executed safely. Thus, we need consider only paths of types (1)-(3) when considering the valid path requirement below.

There are two cases for paths of types (1)-(3).

Case 1: The path is from types (1)-(3), and begins and ends with a $\xrightarrow{po}$ arc.

Paths of type (2) and (3) that begin and end with a $\xrightarrow{po}$ arc are executed safely by RCpc. Paths of type (1) that do not have arcs of the type NLUW $\xrightarrow{po}$ NLUR and NLUR $\xrightarrow{co}$ NLUW $\xrightarrow{vco1}$ SR are also executed safely by RCpc. So consider paths of type (1) with the above arcs. Converting NLURs into read-modify-writes allows replacing the latter arcs by SW $\xrightarrow{vco2}$ SR arcs. As for TSO, the former arcs get replaced by NLUW $\xrightarrow{po}$ NLUW arcs (except for the last one on the path). Thus, if the last arc is not NLUW $\xrightarrow{po}$ NLUR, then the paths get converted into paths that are executed safely by RCpc. If the last arc is NLUW $\xrightarrow{po}$ NLUR, then arguments similar to those for TSO are applicable and ensure that the path is executed safely.

Case 2: The path is from types (1)-(3), and does not begin and end with $\xrightarrow{po}$ arcs.

Consider the longest sub-path of such a path that starts and ends with $\xrightarrow{po}$. If the end points of the above sub-path are conflicting, then the above sub-path is executed safely by RCpc, as argued above. Therefore, the entire path can be replaced by a path with $\xrightarrow{co}$ arcs. This path is executed safely on RCpc.

So assume the above sub-path is between non-conflicting operations; i.e., between reads, say R_1 and R_2. Suppose R_2 is the last operation on this path. Then R_2 is distinguished as either atomic, non-loop, or unpaired and so is replaced by a read-modify-write. Then the path from R_1 to the write of the above read-modify-write is executed safely and so it follows that the entire path is executed safely. Suppose R_2 is not the last operation on this path. Then the next arc must be a $\xrightarrow{co}$ arc and so R_2 must be NLU. So again R_2 is replaced by a read-modify-write. Again, as above, the path must be safe.

Thus, RCpc obeys the valid path requirement with the above mapping. For the control requirement, the $\xrightarrow{rch}$ relation of RCpc is similar to the $\xrightarrow{rch}$ relation described in Appendix D for the valid paths discussed in Appendix G for PLpc2, and for the mapping to RCpc discussed above. However, one difference arises because the $\xrightarrow{rch}$ relation for RCpc was developed to correspond to the necessary requirement for PLpc. As mentioned in appendix G, a difference between PLpc and PLpc2 is that PLpc2 allows some data reads to be self-ordered. Thus, the $\xrightarrow{rch}$ relation for RCpc given in [GAG93] needs to be slightly altered (based on the generic relation in Appendix D) to incorporate such self-ordered reads as well. Recall, however, that this is only pertinent for systems that allow writes to be executed before it is known whether preceding loops will terminate. With the above modification, RCpc executes control paths safely. RCpc obeys the other parts of the control requirement directly.

Appendix I: Correctness of Theorem 8.5 for Detecting Data Races

This appendix shows that theorem 8.5 is correct. The theorem and Condition 8.4 mentioned in the theorem are repeated below for reference. Recall also that the valid paths for data-race-free-0 are $\xrightarrow{hb0}$ paths between conflicting synchronization operations and $\xrightarrow{hb0+}$ paths between other conflicting operations; the valid paths for data-race-free-1 are the same except that $\xrightarrow{hb0+}$ is replaced by $\xrightarrow{hb1}$.

> **Theorem 8.5:** Condition 8.4 for dynamic detection of data races is obeyed by all executions that obey the generic high-level valid path requirement (Condition 7.16) assuming valid paths for data-race-free models as described above, and the generic low-level control requirement (Conditions 7.21) with the following additional restrictions: (1) the $\xrightarrow{vpo}$ relation in properties (b) and (c) for the control relation (Definition 7.19) should be replaced by $\xrightarrow{po}$, (2) if $R \xrightarrow{po} O$ in E where R is a pairable read, then $R \xrightarrow{ctl} O$ in E, and (3) if $R \{ \xrightarrow{ctl} \}+ W$ in E, then $R(i) \xrightarrow{xo} W(j)$ for all i, j in E.

> **Condition 8.4:** For any execution E of a program *Prog*,
>
> (1) if there are no data races in E, then E appears sequentially consistent, and
>
> (2) there exists an SCP of E such that a data race in E either occurs in the SCP, or is affected by another data race that occurs in the SCP.

The proof in this appendix uses notation corresponding to data-race-free-1. The identical proof holds for data-race-free-0 if $\xrightarrow{hb1}$ is replaced by $\xrightarrow{hb0+}$ and all synchronization operations are assumed to be pairable with each other. (Thus, a release and an acquire are paired synchronization write and paired synchronization read respectively.)

We use the term ‘‘data-race-free requirements of theorem 8.5’’ to refer to the high-level valid path and low-level control requirements of Chapter 7 (assuming the valid paths mentioned above), and the additional constraint on the control requirement mentioned in theorem 8.5. Consider an execution, E, of program, *Prog*, that obeys the data-race-free requirements of theorem 8.5. The following lemma first shows that E must obey Condition 8.4(1).

Lemma I.1: E obeys Condition 8.4(1).

Proof:

Condition 8.4(1) requires that if E does not have data races, then E must appear sequentially consistent. Assume that E does not have data races. We show another execution E' that has the same result as E and obeys Condition 7.12 for sequential consistency. Below, we first construct an execution E' that has the same result as E, obeys the data-race-free requirements of theorem 8.5, obeys the write termination assumption of Condition 7.12, and such that there is no instruction instance in E' that follows a loop instance of a loop that is not guaranteed to terminate in every sequentially consistent execution. We then show that E' obeys all the requirements of Condition 7.12 relevant to the data-race-free models. (The loop coherence assumption is the only one not relevant to the data-race-free models.)

Construction of E' that has the same result as E, obeys the data-race-free requirements of theorem 8.5, obeys the write termination assumption of Condition 7.12, and such that no instruction instance in E' follows a loop instance from a loop not guaranteed to terminate in every sequentially consistent execution.

Consider E' which is the same as E, except for the following. First, for every write W in E issued by processor P_i and such that a sub-operation of W is not in E, add the missing sub-operation to the set of sub-operations in E'. Second, from the set of instruction instances of E, delete all instances that follow a loop instance that does not terminate in E and is from a loop that is not guaranteed to terminate in every sequentially consistent execution of the program. Third, delete all memory operations and sub-operations corresponding to the deleted instruction instances from the set of operations and sub-operations of E' (by the finite speculation part of the generic low-level control requirement, this does not include any write operations). Fourth, consider the following modification of the execution order of E as a candidate for the execution order of E': place the sub-operations of a write W by processor P_i that are added to E' just before $W(i)$ in the original execution order of E, and delete all sub-operations of the operations deleted from E'. The following paragraph shows that E' is an execution of *Prog* because its instruction set obeys the uniprocessor correctness condition, and the execution order of E with the above modification obeys the execution order condition for E'.

The instruction set of E' obeys the uniprocessor correctness condition because it is the same as that of E except for instances that follow non-terminating loop instances of E. The execution order of E with the above modification obeys the execution order condition for E' because of the following. For every pair of conflicting operations, O_1 and O_2, in E, there is a valid path between O_1 and O_2 in E (otherwise O_1 and O_2 form a data race). Therefore, by the valid path requirement, all conflicting sub-operations of O_1 and O_2 are in E. Therefore, for a write W in E that is issued by processor P_i, a sub-operation $W(j)$ is not in E only if P_i is not the same as P_j, if there is no read by P_j that conflicts with W, and if there is no other write that conflicts with W. Thus, none of the newly added sub-operations in E' conflict with any other sub-operations of E'. Further, by the finite speculation part of the low-level generic control requirement, none of the deleted sub-operations are writes. Therefore, the modification to the execution order of E obeys the execution order condition for E'.

E' has the same result as E because of the following. The only difference between the instruction instances of E and E' is that some instances following a non-terminating loop instance L in E are not in E', where L is not guaranteed to terminate in every sequentially consistent execution. By the finite speculation part of the low-level generic control requirement, the deleted instruction instances do not write any output interface. Further, all reads that are in E and E' return the same value in E and E'. Therefore, all the instruction instances that write to an output interface are the same in E and E', and write the same value in E and E'. Thus, E and E' have the same result.

E' clearly obeys the data-race-free requirements of Theorem 8.5 and the write termination assumption of Condition 7.12. E' also clearly obeys the requirement that there be no instruction instance that follows a loop instance from a loop not guaranteed to terminate in every sequentially consistent execution. Thus, E' is the required execution.

We next show that E' obeys Condition 7.12.

There is a critical set of E' such that if there is a critical path from X to Y in E', then $X(i) \xrightarrow{xo} Y(i)$ for all i in E'; i.e., the critical paths are executed safely.

E' does not have data races. Therefore, one critical set of E' consists of the above-mentioned valid paths of the corresponding data-race-free model. The valid path requirement ensures that these paths are executed safely.

E' obeys the write termination assumption of Condition 7.12.

This follows trivially from the construction of E'.

E' obeys the finite speculation assumption of Condition 7.12.

The finite speculation assumption requires that an instruction instance in E' should be preceded (by program order) by only a finite number of other instruction instances. As in Appendix E, let the *finite part* of E' be the operations that do not follow infinite operations in E' (recall that *follow* refers to program order). Let the rest of the operations be the infinite part of E'. We first show below that there cannot be a write in the infinite part of E' whose value is read in E' by a read in the finite part of E'.

Suppose there is a write in the infinite part of E' such that its value is read in E' by a read in the finite part of E'. Let W be such a write such that there is no such write ordered before W by the program/conflict graph of E'. Since E' does not have data races and obeys the valid path requirement, E' has an acyclic program/conflict graph; therefore, the above mentioned W exists. Let R be the read in the finite part of E' that returns the value of W. Let the processor of W be P_i. Then there must be a non-terminating loop instance L issued by P_i in the finite part. By construction of E', L must be guaranteed to terminate in every sequentially consistent execution. Further, because of the choice of W, there is no read by P_i in the finite part of E' such that the read returns the value of a write from the infinite part in E'. Consider the set S of operations of E' such that for every operation O in S, (a) O is from the finite part of E', (b) neither O nor any read program ordered before O returns the value of a write from the infinite part in E', and (c) all operations before O by the program/conflict graph of E' are in S. S must include all operations of P_i in the finite part. Then since the program/conflict graph of E' is acyclic, there is a total order of the operations in S such that the order is consistent with the program/conflict order of E' and such that the last write before every read in this order is the same as for the execution order of E'. It follows that any finite prefix of such a total order represents the initial operations of the execution order of some sequentially consistent execution E_s. All operations of the loop instance L are in the total order. Thus, it follows that for any number of iterations of L, there is an E_s where those iterations are in E_s. It follows that for an infinite number of such E_s, the termination of L must depend on reads that are in E' and E_s and form a race in E_s, and L must consist of infinite such reads in E' (the reasoning for this is similar to result 3 in Step E.3 of Appendix E). By property (d) of the control relation and the finite speculation part of the generic low-level control requirement, it follows that there cannot be any synchronization operation following L in E'. However, since W and R cannot form a data race in E', it follows that there must be some synchronization operation following L in E', a contradiction. Thus, no read in the finite part of E' can return the value of a write in the infinite part.

From the above result it follows that there is a total order of all operations in the finite part of E' such that the order is consistent with the program/conflict order of E' and such that the last write before every read in this order is the same as for the execution order of E'. It then follows that (i) this total ordering represents the initial operations of the execution order of some sequentially consistent execution E_s of program *Prog*, and (ii) if there are operations in O from loop instances that do not terminate in E', then these loop instances do not terminate in E_s either. But by construction of E', there is no instruction instance in E' that follows a loop instance that is not guaranteed to terminate in every sequentially consistent execution. Thus, it follows that there is no instruction instance of E' that follows a loop that does not terminate in E'. Thus, E' obeys the finite speculation assumption. □

We next show that E obeys Condition 8.4(2). For a contradiction, assume that E does not obey Condition 8.4(2). The proof uses the following definitions.

Definition : For a prefix (as in definition 8.1), S, of E, call the first operation of a processor (by $\xrightarrow{\text{po}}$) that is executed in E and that is not in S as the S *terminator* of the processor. Denote the S terminator of processor P_i by $t_{i,s}$. ($t_{i,s}$ does not exist if all operations of processor P_i are in S.)

Definition : A read in a prefix of E is a *significant* read for the prefix if it is a pairable read in E or if it controls an operation O in E where O is in the prefix.

Definition : A prefix, S, of E is a *proper prefix* corresponding to a sequentially consistent execution E_s if it obeys the following three properties.

(P1) If $op_1 \xrightarrow{\text{co}} op_2$ in E_s and op_2 is in S, then op_1 is in S.

(P2) If op_1 and op_2 are in S, then either $op_1 \xrightarrow{\text{co}} op_2$ in E_s and E, or $op_2 \xrightarrow{\text{co}} op_1$ in E_s and E.

(P3) A significant read in S returns the value of a write in S in E.

Definition : For a proper prefix S of E, let $S+$ be the set of all operations in S, and all S terminators, $t_{i,s}$, such that the following is true for $t_{i,s}$. $S \cup t_{i,s}$ does not violate property (P3) above for a proper prefix, and if $t_{i,s}$ is a pairable read such that the last write in S to the same location as the read (as ordered by the $\xrightarrow{\text{co}}$ of E) is pairable, then the read returns the value of that pairable write in E.

The proof proceeds in the following steps.

Step 1: Lemma I.2 shows that a proper prefix corresponding to E_s is also an SCP corresponding to E_s.

Step 2: We use Lemma I.2 to choose a particular (‘‘maximal’’) proper prefix of E called *prefix*.

Step 3: Lemma I.3 uses Lemma I.2 to show that any *prefix* terminator is affected by a data race in *prefix*+ in E.

Step 4: Lemma I.4 shows that *prefix*+ is an SCP of E.

Step 5: Lemma I.5 uses Lemmas I.3 and I.4 to show that E obeys Condition 8.4(2).

Below, we say an SCP, S, of E *corresponds* to a sequentially consistent execution, E_s, if E_s satisfies the conditions of Definition 8.2 that make S an SCP.

Step 1: A proper prefix corresponding to E_s is also an SCP corresponding to E_s.

We use the following results to prove the required lemma.

Result 1: Let S be a proper prefix of E corresponding to E_s. A significant read in S returns the value of the same write (and therefore the same value) in E_s and E.

Proof:

A read in an execution returns the value of the conflicting write sub-operation that is ordered last before it by the execution order. By properties (P1) and (P3), this last write for a significant read must be in S for both E_s and E. By property (P2), this last write must be the same for E_s and E.

Result 2: Let S be a proper prefix of E corresponding to E_s and let y be an operation in S. If $x \xrightarrow{\text{po}} y$ in E_s, then x is in S.

Proof:

If x is in E, then x is in S (by definition of a prefix of E). Otherwise, some reads that control y in E return different values in E and E_s. These reads are significant reads in S, a contradiction to Result 1. Thus, x is in S.

Lemma I.2: Let S be a proper prefix of E corresponding to E_s. Then S is an SCP corresponding to E_s.

Proof:

For a contradiction, assume that S is not an SCP corresponding to E_s. Thus, there must be an operation *op* in S where at least one of the following must be true.

(1) *op* is not in E_s.

(2) There exists a memory operation x in E_s such that $x \xrightarrow{\text{hb1}} op$ in E_s but x is not in S.

(3) There exists a memory operation x in S such that x and *op* form a data race in E but not in E_s.

The following shows that each of the above cases leads to a contradiction. Below, consider *op* such that if it obeys (2), then it is the first such *op* (by the $\xrightarrow{\text{xo}}$ of E_s).

Case 1: op is not in E_s.

Then some read that controls *op* in E and is in S does not return the same value in E and E_s. This contradicts Result 1.

Case 2: There exists a memory operation x in E_s such that $x \xrightarrow{\text{hb1}} op$ in E_s but x is not in S.

There are three sub-cases as follows.

Sub-Case 2a: $x \xrightarrow{\text{po}} op$ in E_s.

Then by Result 2, x is in S, a contradiction.

Sub-Case 2b: $x \xrightarrow{\text{so1}} op$ in E_s.

Then by property (P1), x is in S, a contradiction.

Sub-Case 2c: There exists a y such that $x \xrightarrow{\text{hb1}} y \xrightarrow{\text{po}} op$ in E_s or $x \xrightarrow{\text{hb1}} y \xrightarrow{\text{so1}} op$ in E_s.

By previous two sub-cases, y is in S. By the choice of op, x must be in S too, a contradiction.

Case 3: There exists a memory operation x in S such that x and op form a data race in E but not in E_s.

From Case 1, x must be in E_s. Therefore, either $x \xrightarrow{\text{hb1}} op$ or $op \xrightarrow{\text{hb1}} x$ in E_s. The analysis for both cases is identical, and so we consider only the first case below.

Operations x and y cannot be from the same processor since they form a data race in E. Therefore, in E_s, there must be operations rel_i and acq_i such that $x \xrightarrow{\text{po}} rel_1$ or x is the same as rel_1 and $rel_1 \xrightarrow{\text{so1}} acq_1 \xrightarrow{\text{po}} rel_2 \xrightarrow{\text{so1}} acq_2 \cdots \xrightarrow{\text{so1}} acq_n$ and $acq_n \xrightarrow{\text{po}} op$ or acq_n is the same as op and $n \geq 1$. Since $acq_n \xrightarrow{\text{po}} op$ or acq_n is the same as op, it follows (using Result 2) that acq_n must be in S. By Result 1, acq_n returns the value of the same write in E_s and E; therefore, $rel_n \xrightarrow{\text{so1}} acq_n$ in E also. By property (P1), rel_n must be in S; therefore, by Result 2, acq_{n-1} is in S also. Continuing the same argument yields that rel_i and acq_i are in S and $rel_i \xrightarrow{\text{so1}} acq_i$ in E for all $i \leq n$. It follows that $x \xrightarrow{\text{hb1}} op$ in E as well, a contradiction. □

Step 2: Choosing *prefix*.

Consider a proper prefix, *prefix*, and a corresponding *prefix*+ of E such that there is no other proper prefix, S and a corresponding $S+$, of E for which either (i) the number of processors that do not have S terminators or those whose S terminators are affected by a data race in $S+$ is greater than the corresponding number for *prefix*, or (ii) the set of operations in S is a superset of the operations in *prefix*. *prefix* exists because of the following. The null set is a proper prefix of E. Therefore, *prefix* exists unless for every proper prefix, there is always another proper prefix S where S satisfies (i) or (ii). Since the number of processors is finite, it follows that there must be at least one proper prefix of E such that there is no other prefix that satisfies (i). If for every proper prefix, there is always another proper prefix that satisfies (ii), then it follows that all operations of E form a proper prefix. By Lemma I.2, all operations of E form an SCP. Therefore, E obeys Condition 8.4(2), a contradiction.

Henceforth, we will usually consider the *prefix* terminators of E and will therefore drop the reference to *prefix* when discussing terminators of *prefix*. Thus, t_i implicitly denotes the *prefix* terminator of P_i. Further, we will use E_s to implicitly denote a sequentially consistent execution for which *prefix* is a corresponding proper prefix.

Step 3: Any *prefix* terminator is affected by a data race in *prefix*+ in E.

We use the following additional results.

Result 3: If $y \xrightarrow{\text{po}} t_i$ in E_s, then y is in *prefix* or *prefix* $\cup\ t_{i,s}$ violates (P3).

Proof:

If y is in E, then y must be in *prefix* since t_i is the first operation of its processor (by $\xrightarrow{\text{po}}$) that is not in *prefix*. If y is not in E, then some reads that control t_i in E return different values in E and E_s. These reads are in *prefix*. Thus, either t_i is in *prefix* or *prefix* $\cup\ t_i$ violates (P3).

Result 4: If $x \xrightarrow{\text{hb1}} op$ in E and op is in *prefix*, then $x \xrightarrow{\text{hb1}} op$ in E_s.

Proof:

There are two cases as follows.

Case 1: x and op are from the same processor; i.e., $x \xrightarrow{\text{po}} op$ in E.

By Lemma I.2, *prefix* is an SCP for E_s. Therefore, $x \xrightarrow{\text{po}} op$ in E_s, proving the result.

Case 2: x and op are not from the same processor.

There must be operations rel_i and acq_i in E such that $x \xrightarrow{\text{po}} rel_1$ or x is the same as rel_1 and $rel_1 \xrightarrow{\text{so1}} acq_1 \xrightarrow{\text{po}} rel_2 \xrightarrow{\text{so1}} acq_2 \cdots \xrightarrow{\text{so1}} acq_n$ and $acq_n \xrightarrow{\text{po}} t_i$ or acq_n is the same as t_i and $n \geq 1$. Since $acq_n \xrightarrow{\text{po}} op$ or acq_n is the same as op, it follows that acq_n must be in *prefix* and in E_s (using Result 2 and definition of a prefix of E). By Result 1, acq_n returns the value of the same write in E_s and E, therefore, $rel_n \xrightarrow{\text{so1}} acq_n$ in E_s also. By definition of a prefix of E, rel_n must be in *prefix*;

therefore, acq_{n-1} is in *prefix* and in E_s also. Continuing the argument yields that rel_i and acq_i are in *prefix* and E_s and $rel_i \xrightarrow{\text{so1}} acq_i$ in E_s for all $i \le n$. It follows that $x \xrightarrow{\text{hb1}} op$ in E_s as well.

Result 5: Let *op* be an operation in *prefix* and t_i be a *prefix* terminator such that *prefix* $\cup\ t_i$ does not violate (P3), there is no data race in *prefix*+ that affects t_i in *E*, $op \xrightarrow{\text{co}} t_i$ in E_s, and $t_i \xrightarrow{\text{co}} op$ in *E*. Then

(i) t_i forms a race with *op* in *E*.

(ii) if t_i is not in *prefix*+, then t_i is a pairable read,

(iii) t_i is a synchronization operation,

(iv) if t_i is in *prefix*+, then *op* is a synchronization operation.

Proof:

$op \xrightarrow{\text{hb1}} t_i$ in *E* implies $op \xrightarrow{\text{co}} t_i$ in *E* (by the valid path condition), a contradiction. $t_i \xrightarrow{\text{hb1}} op$ in *E* implies t_i is in *prefix*, a contradiction. Therefore, *op* and t_i must form a race in *E*, proving (i).

The proof for (ii) follows directly from the definition of *prefix*+ and since *prefix* $\cup\ t_i$ does not violate (P3).

Suppose t_i is a data operation. By the proof of (ii), t_i is in *prefix*+. But then by (i), there is a data race in *prefix*+ that affects t_i in *E*, a contradiction.

If *op* is a data operation and t_i is in *prefix*+, then again by (i), there is a data race that affects t_i in *E*, a contradiction.

Lemma I.3: A *prefix* terminator is affected by a data race in *prefix*+ in *E*.

Proof:

Consider a *prefix* terminator t_i. In the following, consider an E_s such that *prefix* is a proper prefix corresponding to E_s and such that the $\xrightarrow{\text{xo}}$ of E_s orders the operations in *prefix* before any operations that are not in *prefix*, and orders t_i before any other operations that are not in *prefix*. Such an E_s is possible by (P1) and Result 3. *prefix* $\cup\ t_i$ cannot be a proper prefix since there is no superset of *prefix* that is a proper prefix. Therefore, *prefix* $\cup\ t_i$ must violate one of properties (P1)-(P3) for E_s. We divide our proof into five exhaustive cases based on the above observation. Cases 1 and 5 assume *prefix* $\cup\ t_i$ violates (P1) and (P3) respectively. Cases 2, 3, and 4 assume *prefix* $\cup\ t_i$ violates (P2), and examine three exhaustive ways in which this violation can occur. Specifically, *prefix* $\cup\ t_i$ violates (P2) if there is an operation *op* in *prefix* such that either *op* and t_i are not related by $\xrightarrow{\text{co}}$ in *E*, or $op \xrightarrow{\text{co}} t_i$ in *E* and $t_i \xrightarrow{\text{co}} op$ in E_s, or $t_i \xrightarrow{\text{co}} op$ in *E* and $op \xrightarrow{\text{co}} t_i$ in E_s. Cases 2, 3, and 4 examine each of the above separately. We show that for each of the five cases below, either there is a contradiction, or a later case is valid, or a data race in *prefix*+ affects t_i in *E*. We proceed by contradiction. Suppose for each of the following cases, no later case applies and no data race in *prefix*+ affects t_i in *E*.

Case 1: prefix $\cup\ t_i$ violates property (P1); i.e., there must be an operation *x* in E_s such that $x \xrightarrow{\text{co}} t_i$ in E_s and *x* is not in *prefix*.

By our choice of E_s, all operations ordered before t_i by $\xrightarrow{\text{co}}$ in E_s are in *prefix*, a contradiction.

Case 2: prefix $\cup\ t_i$ violates property (P2) such that there is an operation *op* in *prefix* such that *op* and t_i are not related by $\xrightarrow{\text{co}}$ in *E*.

By the valid path requirement, *op* and t_i must form a race in *E*. We also know that t_i is in E_s (by Result 3 and since *prefix* $\cup\ t_i$ does not violate (P3)). Further, we know that $op \xrightarrow{\text{co}} t_i$ in E_s (since *prefix* obeys (P1)). There are three sub-cases discussed below.

Sub-Case 2a: At least one of *op* or t_i is a data operation in *E*.

op and t_i form a data race in *E* and this data race affects t_i. Therefore, t_i cannot be in *prefix*+. Since *prefix* $\cup\ t_i$ does not violate (P3), from the definition of *prefix*+, it follows that t_i must be a pairable read, and the last write in *prefix* that conflicts with t_i (by the $\xrightarrow{\text{xo}}$ of *E*) is a pairable write *w*. Further, this write *w* is different from *op* since *op* and t_i form a data race. Also since t_i is a read, *op* must be a write. By (P2), $op \xrightarrow{\text{co}} w$ in *E*. If $op \xrightarrow{\text{hb1}} w$ in *E*, then by the valid path requirement, all sub-operations of *op* are in *E*. It follows that *op* and t_i are related by $\xrightarrow{co}$ in *E*, a contradiction. Therefore, *op* and *w* must form a data race in *E*. Since *op* and t_i form a race in *E*, the data race between *op* and *w* affects t_i in *E*, a contradiction.

Sub-Case 2b: op and t_i are both synchronization operations in *E* and there is no path from *op* to t_i in $\xrightarrow{po} \cup \xrightarrow{co}$ of E_s that ends in a $\xrightarrow{po}$ arc in E_s.

Modify the execution order of E_s as follows. Consider all operations between *op* and t_i that have a path to t_i in $\xrightarrow{po} \cup \xrightarrow{co}$ of E_s that ends in a $\xrightarrow{po}$ arc. Move all the above operations to just before *op*. Next, if *op* is the write of a read-modify-write, then move the corresponding read to just before *op*. The above order preserves the $\xrightarrow{po}$ and $\xrightarrow{co}$ relation of E_s. Next derive two orders from the above order as follows. In the first order, move t_i (and the write of its read-modify-write if t_i is a read from a read-modify-write) to just before *op* (and before the read of *op*'s read-modify-write if *op* is a write of a read-modify-write). In the second order, move t_i to just after *op* (and after the write of *op*'s read-modify-write if *op* is a read of a read-modify-write). If any of t_i, *op*, and the other operations from their read-modify-writes (if any) are reads, then let those reads return the value of the last conflicting write before them in the new orders. The modified orders until *op*, t_i and the other operations of their read-modify-writes (if any) are the initial part of some sequentially consistent executions of *Prog* where there is a race path from t_i to *op* in at least one of these executions. Therefore, from the write termination part of the low-level generic control requirement, it follows that *op* and t_i must be related by $\xrightarrow{co}$ in *E*, a contradiction.

Sub-Case 2c: op and t_i are both synchronization operations in *E* and there is a path from *op* to t_i in $\xrightarrow{po} \cup \xrightarrow{co}$ of E_s that ends in a $\xrightarrow{po}$ arc in E_s.

Consider the longest path from *op* to t_i in $\xrightarrow{po} \cup \xrightarrow{co}$ of E_s such that the path ends in a $\xrightarrow{po}$ arc and does not have two consecutive $\xrightarrow{po}$ arcs. Then all operations on this path must be synchronization operations for the following reason. Suppose there is a data operation op_1 on the path. Then this operation must be on a $\xrightarrow{co}$ arc on the path. Let the other operation on this arc be op_2. Then op_1 and op_2 must form a data race in E_s (otherwise, the path chosen is not the longest path possible). By Result 4, op_1 and op_2 also form a data race in *E*. By (P2), the above path also exists in *E*. It follows that the above data race affects t_i in *E*, a contradiction. Thus, all operations on the chosen path are synchronization operations and therefore the path is a $\xrightarrow{hb0}$ path in E_s. By (P2), this $\xrightarrow{hb0}$ path also exists in *E*. It follows from the valid path requirement that *op* and t_i must be related by $\xrightarrow{co}$ in *E*, a contradiction.

Case 3: prefix $\cup$ t_i violates property (P2) such that there is an operation *op* in *prefix* such that *op* $\xrightarrow{co}$ t_i in *E* and t_i $\xrightarrow{co}$ *op* in E_s.

Since *op* is in *prefix* and t_i is not, this implies that *prefix* violates (P1), a contradiction.

Case 4: prefix $\cup$ t_i violates property (P2) such that there is an operation *op* in *prefix* such that t_i $\xrightarrow{co}$ *op* in *E* and *op* $\xrightarrow{co}$ t_i in E_s.

Denote the set of operations *op* in *prefix* that satisfy this case by *O*. Denote the last operation of P_i (by $\xrightarrow{po}$) that is in *prefix* by l_i. Since *prefix* $\cup$ t_i does not violate (P3), Result 3 implies that l_i is the last operation of P_i preceding t_i in E_s. There are two cases depending on whether there is a path in $\xrightarrow{po} \cup \xrightarrow{co}$ of E_s from any of the operations in *O* to l_i. Each case is separately handled below. (Below we abbreviate $\xrightarrow{po} \cup \xrightarrow{co}$ by $\xrightarrow{pc}$.)

Sub-case 4a: There is no path in $\xrightarrow{pc}$ of E_s from any operation in *O* to l_i.

Modify the $\xrightarrow{xo}$ of E_s as follows. Move all the operations in *O*, and all the operations ordered after any operation in *O* by the $\xrightarrow{pc}$ of E_s (and that are before t_i) to just before t_i. The moved operations cannot include l_i because no operation in O has a path to l_i in the $\xrightarrow{pc}$ of E_s. Further, if t_i is a write of a read-modify-write, the moved operations cannot include a conflicting write. Therefore, the new order is consistent with $\xrightarrow{po}$ and $\xrightarrow{co}$ of E_s, and forms an execution order of a sequentially consistent execution of *Prog*. Next move t_i just before all the operations moved in the above step. (If t_i is a read from a read-modify-write, then move the write of the read-modify-write to just before any conflicting writes between t_i and the write of t_i's read-modify-write.) The new order still preserves $\xrightarrow{po}$. Make all reads until $\mathit{prefix} \cup t_i$ in the new order return the value of the last conflicting write before them by the new order. We prove in the next paragraph that all operations that are moved that conflict with t_i are in O. Therefore, the $\xrightarrow{co}$ relation on the operations in $\mathit{prefix} \cup t_i \cup$ {the write of t_i's read-modify-write (if t_i is a read from a read-modify-write)} is the same for the new order and E. Therefore, the last write before all reads in *prefix* that control an operation in $\mathit{prefix} \cup t_i$ in E is the same in E and in the new order and by Result 1, this is the same in E_s. Therefore, all the above reads return the same value in the new order and in E and in E_s. Therefore, the new order until and including all the moved operations forms the initial operations of an execution order of some sequentially consistent execution, $E_s{'}$, of *Prog*. Further, $\mathit{prefix} \cup t_i$ obeys (P1)-(P3) for $E_s{'}$. Thus, we now have a proper prefix that is a superset of *prefix*, a contradiction.

We still, however, need to give the proof mentioned above that all operations that are moved that conflict with t_i are in O.

For a contradiction, suppose there is an operation that is moved that conflicts with t_i and is not in O. Let op_2 be the first such operation (by the $\xrightarrow{xo}$ of E_s). Let op_1 be the first operation in O before op_2 (by the $\xrightarrow{xo}$ of E_s) such that there is a path in $\xrightarrow{pc}$ of E_s from op_1 to op_2. (There has to be such an op_1 or op_2 cannot have been moved.)

Suppose op_1 and op_2 conflict. Then in E, $op_1 \xrightarrow{co} op_2$, $op_2 \xrightarrow{co} t_i$, $t_i \xrightarrow{co} op_1$. By the valid path requirement, this is possible only if at least two of op_1, op_2 and t_i form a data race in E. This data race must affect t_i in E. This is a contradiction unless the only data race between the above operations involves t_i and t_i is not in *prefix*+. From Result 5, t_i must be a (pairable) read. Therefore, op_1 and op_2 are both writes; further, they do not form a data race. Therefore, $op_1(k) \xrightarrow{xo} op_2(k)$ for all k in E. It therefore follows that $op_1(i) \xrightarrow{xo} t_i(i)$ in E, a contradiction.

Suppose op_1 and op_2 do not conflict. Then they are both reads, and therefore t_i is a write. From Result 5, it follows that t_i is in *prefix*+, and t_i and all operations in O are synchronization operations. If there is a path from op_2 to l_i in $\xrightarrow{pc}$ of E_s, then there is a path from op_1 to l_i in the $\xrightarrow{pc}$ of E_s, a contradiction. Therefore, there is no path from op_2 to l_i in the $\xrightarrow{pc}$ of E_s. Therefore, there is no $\xrightarrow{hb1}$ path from op_2 to l_i in E_s. From Result 4, there is no $\xrightarrow{hb1}$ path from op_2 to l_i in E. Therefore, there is no $\xrightarrow{hb1}$ path from op_2 to t_i in E (since t_i is a write). There cannot be a $\xrightarrow{hb1}$ path from t_i to op_2 in E; otherwise, t_i would be in *prefix*. This implies that op_2 and t_i form a race in E. Suppose op_2 is a data operation. Then we have a data race in *prefix*+ that affects t_i in E, a contradiction. Therefore, op_2 must be a synchronization operation. Perform the following transformation on the path from op_1 to op_2 in the $\xrightarrow{pc}$ of E_s. First, if any $\xrightarrow{co}$ edge can be replaced with a longer path in the $\xrightarrow{pc}$ of E_s, then replace it. Now replace any consecutive $\xrightarrow{po}$ edges with one $\xrightarrow{po}$ edge. The above path has the property that if there is a data operation on it with a $\xrightarrow{co}$ edge into or out of it, then the data operation forms a data race in E_s (and therefore in E by Result 4 and property (P2)) with the operation on the other side of the $\xrightarrow{co}$ edge. (If this not true, then the concerned $\xrightarrow{co}$ edge is a $\xrightarrow{hb1}$ edge which can be replaced with a longer path in $\xrightarrow{pc}$ of E_s, a contradiction.) We will need the above transformation and property in a later case also and will refer to them as the *data race* *transformation and property* for convenience.

Suppose all operations on the above transformed path are synchronization operations. Then by (P2), there exists a $\xrightarrow{\text{hb0}}$ path from op_1 to t_i in E. By the valid path condition, $op_1(k) \xrightarrow{\text{xo}} t_i(k)$ for all k in E, a contradiction.

Suppose at least one of the operations on the above transformed path is data. We know that op_1 and op_2 are both synchronization operations, therefore the data operation on the above path will be on a $\xrightarrow{\text{co}}$ edge. Therefore, the data operation forms a data race in E_s (and therefore in E by Result 4). By (P2), the above path also exists in $\xrightarrow{\text{pc}}$ of E. Therefore, the above data race affects t_i in E, a contradiction. Thus, any operation that is moved that conflicts with t_i must be in O.

Sub-case 4b: There is a path from some operation in O to l_i in the $\xrightarrow{\text{pc}}$ of E_s.

Let op_1 be the closest such operation to l_i as ordered by the $\xrightarrow{\text{xo}}$ of E_s. Perform the data race transformation discussed earlier on the path from op_1 to l_i. In addition, if the edge to l_i is a $\xrightarrow{\text{po}}$ edge, then remove l_i from the path. Thus, the last edge on the path is a $\xrightarrow{\text{co}}$ edge to an operation from P_i.

Suppose all operations on the transformed path are synchronization. Then by (P2), there exists a $\xrightarrow{\text{hb0}}$ path from op_1 to t_i in E; therefore, $op_1(k) \xrightarrow{\text{xo}} t_i(k)$ for all k in E, a contradiction.

Suppose at least one of the operations on the transformed path is data. If the data operation lies on a $\xrightarrow{\text{co}}$ edge, then by the data race property of the transformed path discussed earlier, this data operation forms a data race with the other operation on the $\xrightarrow{\text{co}}$ edge in E_s. This is also a data race in E (by Result 4) and affects l_i in E (because of the path to l_i), and therefore affects t_i in E, a contradiction.

The only remaining case is where the only data operations on the transformed path do not lie on $\xrightarrow{\text{co}}$ edges. op_1 is the only operation that can satisfy the above condition. Thus, the only case left is where op_1 is the only data operation on the transformed path and the edge out of op_1 is a $\xrightarrow{\text{po}}$ edge. By Result 5, it must be that t_i is not in *prefix*+, and so t_i is a (pairable) read operation.

Consider the set O' of writes ordered after op_1 by the $\xrightarrow{\text{co}}$ of E_s. Modify the $\xrightarrow{\text{xo}}$ of E_s as follows (similar to the modification of Sub-case 4a). Move all the operations in O', and all the operations ordered after any operation in O' by the $\xrightarrow{\text{pc}}$ of E_s to just before t_i. Next move t_i to before the operations moved in the first step above. (If t_i is a read from a read-modify-write, then move the write of the read-modify-write to just before any conflicting writes between t_i and the write of t_i's read-modify-write.) Note that the operations in O' must also be in O because of the following. Suppose there is some op_2 in O' that is not in O. Then we have $op_1 \xrightarrow{\text{co}} op_2$, $op_2 \xrightarrow{\text{co}} t_i$, and $t_i \xrightarrow{\text{co}} op_1$ in E, a contradiction. Let a read in the new order return the value of the last conflicting write before it in the new order. As for Sub-Case 4a, the new order until the operations of *prefix* $\cup\, t_i$ represents the initial operations of a sequentially consistent execution $E_s{}'$. Consider S = *prefix* - all the moved operations. S is a prefix of E and obeys (P1)-(P3). Further, the last write in S (by the $\xrightarrow{\text{xo}}$ of E) to t_i's location is a data write; therefore, t_i is in $S+$. Further, t_i is affected by a data race (between op_1 and t_i) in $S+$ in E. Also all the operations that were deleted from *prefix* to form S were ordered after op_1 by $\xrightarrow{\text{pc}}$ of E_s and therefore of E (by (P2)). Therefore, the data race above affects all the deleted operations too. Thus, for S, the number of processors without terminators or whose terminators are affected by a data race in $S+$ is greater than the corresponding number for *prefix*, a contradiction.

Case 5: prefix $\cup$ t_i violates property (P3), i.e., t_i is a pairable read that does not return the value of write in E that is in *prefix*, or some read in *prefix* controls t_i and does not return the value of a write in *prefix* $\cup$ t_i in E.

If t_i satisfies the first case above, then let $r_i = t_i$, else let the read that satisfies the second case above be r_i.

Let the write whose value r_i returns in E be called the *violator* of P_i, denoted by v_i. We have so far proved that for every processor, P_i, that has a terminator t_i, either there is a data race in *prefix+* that affects t_i in E or t_i satisfies the current case. Suppose P_i satisfies the current case and v_i is issued by a processor P_j, which does not satisfy the current case. Then a data race $\langle x,y \rangle$ in *prefix* affects t_j. The data race $\langle x,y \rangle$ also affects t_i and we have proved our proposition for P_i.

The only remaining case is when P_i satisfies the current case and v_i is issued by a processor that also satisfies the current case. Such processors form clusters, where a cluster labeled (without loss of generality) as P_1, P_2, ..., P_k is a set of processors such that for $j < k$, v_j is executed by Pj+1, and vk is executed by P_1.

Consider a cluster P_1, P_2, ..., P_k. The violator v_1 of P_1 is issued by P_2. By the data-race-free requirements of theorem 8.5, it follows that in E, $r_2(2) \xrightarrow{xo} v_1(j)$ for all j. We know that $v_1(1) \xrightarrow{xo} r_1(1)$ in E (because r_1 reads the value of v_1 in E). Therefore, $r_2(2) \xrightarrow{xo} r_1(1)$ in E. Similarly, $r_3(3) \xrightarrow{xo} r_2(2)$ in E and so on until $r_k(k) \xrightarrow{xo} r_{k-1}(k-1)$ and $r_1(1) \xrightarrow{xo} r_k(k)$ in E. Therefore, $r_1(1) \xrightarrow{xo} r_1(1)$ in E. This is not possible. Therefore, there cannot be any clusters, the final contradiction for Lemma I.3. □

Step 4: *prefix+* is an SCP of E.

Lemma I.4: prefix+ is an SCP of E.

Proof:

The proof proceeds by contradiction. Consider an E_s for which *prefix* is a corresponding proper prefix of E, and whose $\xrightarrow{xo}$ orders its operations as follows. The $\xrightarrow{xo}$ first orders operations of *prefix*, followed by remaining reads of *prefix+*, followed by remaining writes of the read-modify-writes of reads of *prefix+*, followed by the remaining writes of *prefix+*, followed by other operations. Such an E_s is possible by (P1), Result 3, and since the terminators in *prefix+* do not violate (P3).

Assume for a contradiction that *prefix+* is not an SCP corresponding to the above E_s. Then at least one of the following must be true for some *prefix* terminator t_i that is also in *prefix+*.

(1) t_i is not in E_s.

(2) There exists a memory operation x in E_s such that $x \xrightarrow{hb1} t_i$ in E_s but x is not in *prefix+*.

(3) There exists a memory operation x in E such that $x \xrightarrow{hb1} t_i$ in E but x is not in *prefix+*.

(4) There exists a memory operation x in *prefix* such that x and t_i form a data race in E but not in E_s.

The first and second cases are not possible by construction of E_s. The following shows that the third and fourth cases also lead to a contradiction.

Case 3: There exists a memory operation x executed in E such that $x \xrightarrow{hb1} t_i$ in E but x is not in *prefix+*.

Operations x and t_i must be from different processors. Either $x \xrightarrow{so1} t_i$ in E or there exists a y such that $x \xrightarrow{hb1} y \xrightarrow{po} t_i$ in E or $x \xrightarrow{hb1} y \xrightarrow{so1} t_i$ in E. Suppose $x \xrightarrow{so1} t_i$ in E. Then *prefix* $\cup$ t_i violates (P3), a contradiction. Suppose there is a y such that $x \xrightarrow{hb1} y \xrightarrow{po} t_i$ or $x \xrightarrow{hb1} y \xrightarrow{so1} t_i$ in E. In the first case, since t_i is the first operation of its processor (by $\xrightarrow{po}$) not in *prefix*, y is in *prefix* and so x is in *prefix*, a contradiction. In the second case, since *prefix* $\cup$ t_i does not violate (P3), y is in *prefix* and so x must be in *prefix* too, a contradiction.

Case 4: There exists a memory operation x in *prefix+* such that x and t_i form a data race in E but not in E_s.

Either $t_i \xrightarrow{hb1} x$ or $x \xrightarrow{hb1} t_i$ in E_s. Suppose $t_i \xrightarrow{hb1} x$ in E_s. Then t_i must be in *prefix*, a contradiction. Suppose $x \xrightarrow{hb1} t_i$ in E_s. Either $x \xrightarrow{so1} t_i$ in E_s or there is a y such that $x \xrightarrow{hb1} y \xrightarrow{po} t_i$ in E_s or $x \xrightarrow{hb1} y \xrightarrow{so1} t_i$ in E_s. Suppose $x \xrightarrow{so1} t_i$ in E_s. Then both x and t_i are synchronization operations, therefore they cannot form a data race in E, a contradiction. Suppose there is a y such that $x \xrightarrow{hb1} y \xrightarrow{po} t_i$ in E_s. By construction, y is in *prefix* and so $x \xrightarrow{hb1} y$ in E and so $x \xrightarrow{hb1} t_i$ in E, a contradiction. Suppose there is a y such that $x \xrightarrow{hb1} y \xrightarrow{so1} t_i$ in E_s. By construction, y is in *prefix*. So $x \xrightarrow{hb1} y$ in E. Since t_i is in *prefix*+, it follows that t_i returns the value of y in E, so $x \xrightarrow{hb1} t_i$ in E, a contradiction. □

Step 5: *E* obeys Condition 8.4(2).

Lemma I.5: E obeys Condition 8.4(2).

Proof:

E obeys Condition 8.4(2) if there is an SCP of *E* such that a data race in *E* is either in the SCP or is affected by a data race in the SCP. Lemma I.3 implies that for every processor, either all of its operations in *E* are in *prefix* or there is a data race in *prefix*+ that affects its terminator in *E*. The terminator is the first operation of its processor (by $\xrightarrow{po}$) that is executed in *E* that is not in *prefix*; therefore, a data race that affects a processor's terminator also affects all operations of the processor that are executed in *E* that are not in *prefix*. Thus, Lemma I.3 implies that for every operation in *E* that is not in *prefix*, there is a data race in *prefix*+ that affects the operation in *E*. Thus, it follows that every data race in *E* is either in *prefix*+ or is affected by a data race in *prefix*+. By Lemma I.4, *prefix*+ is an SCP. Therefore, *prefix*+ is the SCP required by Condition 8.4(2). Thus, *E* obeys Condition 8.4(2). □

Theorem 8.5 directly follows from Lemmas I.1 and I.5. □