TESTING MEMORY CONSISTENCY OF SHARED-MEMORY MULTIPROCESSORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF
ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Chaiyasit Manovit

June 2006

Abstract

Shared-memory multiprocessors are becoming the dominant architecture for single-chip and multi-chip microprocessor based systems. Shared memory architectures are difficult to design because they must correctly implement the complexity of cache coherence and a memory consistency model. Memory consistency is a contract between hardware and software that specifies how memory behaves with respect to read and write operations from multiple processors.

We address the challenge of correctly implementing a memory consistency model by developing a methodology for testing shared-memory multiprocessors which is composed of three steps: generating pseudo-random multithreaded programs, executing these programs on a system under test, and checking their compliance with the given memory consistency model. Although the last step is known to be an NP-complete problem, we develop a suite of novel algorithms that work efficiently in practice. Using these algorithms, our methodology has found hundreds of bugs during design and verification of several commercial-graded processors. Many of these bugs are subtle and could not have been detected otherwise.

We also successfully apply our methodology to transactional memory, an emerging architecture that can significantly improve programmability while preserving or even enhancing efficiency of the memory system.

Acknowledgments

First and foremost, I would like to thank all my dissertation committee members, Oyekunle Olukotun, Giovanni De Micheli, and Robert Cypher, all of whom are truly great advisers and mentors. Without the generous support from Giovanni De Micheli, my Ph.D. pursuit may not have even started. His optimism also encouraged me to welcome changes when my research interest began to shift, which resulted in my joining Sun Microsystems and switching to Olukotun's group. At Sun, I am grateful to Robert Cypher for his exceptional expertise and the inspiration which saw me navigate through the research in verifying memory consistency and related concepts. Oyekunle Olukotun helped connect my work to a trendy research topic, and it is with his vision and support that I was eventually able to reach this final milestone. I would also like to thank Bernard Widrow who graciously served as my orals committee chairman.

The team at Sun were also a great source of support. Sudheendra Hangal was practically my fourth adviser, with many interesting questions and ideas often bouncing between us. In particular, the following people have made Sun one of my best experiences: Durgam Vahia, Sridhar Narayanan, Gopal Reddy, Aleksandr Gert, and Juin-Yeu Joseph Lu.

With De Micheli's group, I received financial support from Stanford's Electrical Engineering Department, the Microelectronics Advanced Research Corporation (MARCO), and the National Science Foundation (NSF). I am thankful for the guidance from many of his former students, especially Jim Smith, Luca Benini, Tajana Simunic, and Yung-Hsiang Lu. I also enjoyed the friendships with other

group members and visitors, particularly Armita Peymandoust, Terry Tao Ye, Luc Semeria, Eui-Young Chung, Davide Bertozzi, and Srinivasan Murali.

The following people helped improve the quality of this thesis one way or another: Christoforos Kozyrakis, Hassan Chafi, Austen McDonald, John Davis, and David Lande. I am also appreciative for the administrative support from Kathleen DiTommaso, Evelyn Ubhoff, and Darlene Hadding.

Finally, I would like to thank my friends and family for fulfilling my life outside school and work, with special thanks to my parents for their constant enthusiasm in providing me as best an education as they could.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Although microprocessor performance has been growing at an exponential rate as suggested by Moore's law, there is always demand for large computing capacity beyond what can readily be provided by a single processor, even the most advanced one, hence a need for multiprocessor systems. Furthermore, striving to achieve ever higher performance, the industry has now turned more toward dual-core or multi-core designs as these are proving to be a better way to utilize resources on a chip. Therefore, even systems with a single processor chip are becoming multiprocessors.

1.1.1 Shared-Memory Multiprocessors

A popular architecture for multiprocessors is the shared-memory architecture where all processors share the same memory space. By sharing the same memory, processors can communicate to coordinate their execution by using only the basic memory read and write operations, as opposed to using an explicit mechanism such as that required by another architecture called the message-passing architecture. This convenience makes shared-memory systems relatively easier to program.

From a programmer's perspective, a simple shared-memory multiprocessor could be modeled as shown in Figure 1.1. All processors are connected to the main memory

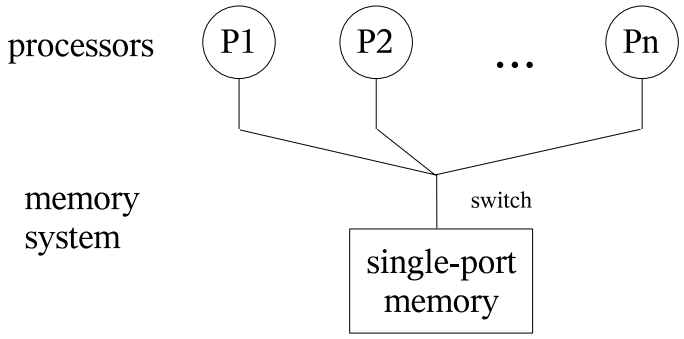


Figure 1.1: A programmer's model of a simple shared-memory multiprocessor.

through a switch which grants the access to only one arbitrarily selected processor at a time. The connected processor will then perform one memory operation, according to the order specified by its program, and disconnect itself from the memory. In reality, actual implementations will possess several details, making them much more complex than this abstraction.

1.1.2 Memory Consistency Models

One of the major attributes that describe a shared-memory multiprocessor is its *memory consistency model*, essentially a contract between hardware and software regarding the semantics of memory operations. The simple abstract model shown in Figure 1.1 is an example of a memory consistency model, called *Sequential Consistency* (*SC*), which was first defined by Lamport in 1979 [38]. Note that an actual system needs not strictly implement what is shown in the figure, as long as its behavior that *appears* to programmers conforms to the model. Nevertheless, the SC model is already restrictive enough that it precludes many optimizations used by most microprocessors today. Therefore, relaxations have been made to the SC model to allow for better performance. This attempt results in relaxed memory consistency models (e.g. [2, 15, 16, 23, 28, 64, 70]) which become less intuitive, however, making them difficult for both hardware designers and software writers to understand. As an example, while cache memory at each processor is functionally transparent to programmers in the SC model, it may not be so in some relaxed memory consistency models.

1.1.3 Verifying Shared-Memory Multiprocessors

To build a shared-memory multiprocessor system, it is insufficient and inappropriate to assume that putting functional processors together will create a functional system. These processors have to be explicitly designed to cooperate, and they must be carefully and thoroughly verified, starting from the specifications, at a high level of abstraction, down to every detail in the implementations.

As technology keeps advancing, design complexity always tends to increase, and so does verification effort which, unfortunately, seems to be growing at a faster rate than design effort. For example, verification was claimed to account for 50% of the total development effort spent on a chip design in 1999, and this contribution recently went up to 80% in 2004 [36]. Despite this significant amount of effort in verification, microprocessors are still shipped with many design defects. For example, processor errata available in the public domain usually report about 10 to 100 bugs in each microprocessor. This is not satisfactory, considering the fact that most, if not all, processors have already gone through a few silicon revisions before being released to the market. Additional revisions can incur significant cost. Sarangi further characterizes these processor errata and finds that majorities of critical bugs, roughly 44%, are in the memory subsystem [59].

It is not surprising that the memory subsystem is one of the most bug-prone because it is among the most complex parts of modern computer system designs based on shared memory multiprocessing. With the large gap between processor and memory speeds, and the trend towards multiple logical processors on a single chip – for example, by employing chip multiprocessing (CMP) or simultaneous multithreading (SMT) techniques – there is intense pressure on computer architects to design high performance memory systems to feed these processors. This leads to ever more complex designs involving shared caches, pipelined protocols, speculative memory operations and elaborate coherence mechanisms. Verifying that these designs work correctly, both in terms of protocol and implementation, is a challenging problem. This is not a problem for high-end workstation and server systems alone; even personal computers are beginning to incorporate multiprocessing capabilities.

There are several techniques involved in verifying shared-memory multiprocessors.

Formal verification is usually regarded as the most rigorous approach because it attempts to exhaustively check the design. However, a complete multiprocessor design is simply too large for this approach and, therefore, it is typically limited to verifying only part of the design or is employed at the protocol or the specification level where many details are abstracted away. This leaves the actual implementation of the design unchecked, which is, in fact, a significant source of complexity and errors in large designs.

Testing approach, on the other hand, can be applied at any level up to the complete system. In this approach, stimuli are manually or randomly created and applied to the module under test, and the results are observed and checked. For a complete multiprocessor system, these stimuli usually refer to test programs.

Industrial design teams have been using random code generators extensively for processor verification [6, 41, 46, 66]. Most code generators rely on a self-checking mechanism or an instruction-level simulator to check for correct execution of test programs. A part of the problem in this testing approach, however, is the difficulty of reasoning about the validity of results of a program which has *data races*. Since the results of such a program are timing-dependent, multiple legal outcomes may exist, and a simple architectural model of the processor cannot be used to cross-check results. In fact, it has been shown by Gibbons and Korach that verifying sequential consistency of shared-memory program executions is an NP-complete problem [26]. Consequently, current testing methodologies usually omit data races entirely or avoid the difficulty in result checking by constraining the test programs to produce only predictable results so that checking their correctness is relatively straight-forward. In pre-silicon verification, some additional checks may be added to the design as artificial modules that monitor the correctness in real time. These checks, however, are not very sophisticated and they are design dependent.

1.2 Thesis Contributions

Our research addresses the challenge of correctly implementing a memory consistency model for shared-memory multiprocessors, with the following primary contributions:

- We develop a methodology for testing shared-memory multiprocessor implementations by focusing on scenarios that allow data races [30]. The methodology is applicable to both pre- and post-silicon validation and is composed of three steps: generating pseudo-random multithreaded programs, executing these programs on a system under test, and checking the execution results for their compliance with the given memory consistency model.

- We develop a suite of novel algorithms for the checking step in the above methodology [30, 43, 44]. Despite the fact that this check is an NP-complete problem, which means there is no known polynomial time algorithm that can solve it completely, our algorithms work efficiently and effectively in practice, having found several subtle memory inconsistencies in actual processor designs.

- With *transactional memory* [31] emerging as active research for a new multiprocessor architecture which can significantly improve programmability while preserving or even enhancing efficiency of the memory system, we extend our work to support this architecture by first formally specifying its memory consistency model and then integrating this model into our algorithms [45].

- We successfully apply our methodology to test several shared-memory multiprocessor designs, uncovering hundreds of bugs during their design and verification.

1.3 Thesis Organization

This thesis is divided into 7 chapters.

Chapter 2 discusses memory consistency models in more detail. We also present their formal specifications using an axiomatic framework. This framework serves as a foundation of our work in later chapters.

Chapter 3 outlines our methodology, called *TSOtool*, for testing implementations of shared-memory multiprocessors. The methodology is composed of three steps: generating test programs, executing them, and checking the compliance of the execution results with a given memory consistency model.

Chapter 4 focuses on algorithms used in the checking step in our methodology. Some variants of the problem are considered. We also present some optimization techniques that improve performance of our algorithms dramatically.

Chapter 5 presents transactional memory and its formal specification as a memory consistency model. We also show how our algorithms presented in the previous chapter can be easily extended to transactional memory.

Chapter 6 presents the results of applying our methodology to several shared-memory multiprocessor systems designed and built at Sun Microsystems. We also successfully apply our methodology to a research prototype of transactional memory called *Transactional memory Coherence and Consistency* (*TCC*) which is being developed here at Stanford.

Chapter 7 concludes and discusses potential directions for future research.

Chapter 2

Memory Consistency Models

Although shared-memory multiprocessor systems can be simplistically described as systems with a single address space, they possess several subtle characteristics which require further clarification; for example, in order to write correct and optimal code for a particular multiprocessor system, programmers may wish to know whether or not two memory updates performed by a processor will be observed in the same order by all other processors. Therefore, a detailed specification which establishes a contract between the hardware and software regarding the behavior of accesses to the shared memory is usually required. Such a specification is called a *memory consistency model* or a *memory model* (both terms may be used interchangeably throughout this dissertation).

In this chapter, we examine a variety of memory consistency models ranging from the most intuitive model, which is also the most restrictive, to the more relaxed models which trade programmability for performance. For example, performance has been shown to improve substantially when each processor buffers its write operations in a separate queue which will be looked up by subsequent read operations (often called bypassing or read forwarding) and is drained in parallel with execution of other instructions. Performance evaluation of memory consistency models is beyond our scope, however, and we refer readers to Gharachorloo's dissertation for more detail [21]. This chapter also presents an axiomatic approach to establishing formal specifications of memory consistency models. These specifications serve as a

foundation of our work in later chapters.

2.1 Sequential Consistency

The most intuitive memory consistency model, *Sequential Consistency* (*SC*), was first defined by Lamport as he extended the definition of the term "sequential" for a uniprocessor to a multiprocessor system [38]. It is the model that most programmers would naturally assume.

A conceptual SC system can be modeled as shown in Figure 2.1 where processors are connected to the shared memory through a switch. The following describes the behavior that *appears* to programmers.

1. The switch connects the memory to only one processor at a time, and the memory services only one operation at a time, thus, making each memory operation to appear to execute atomically with respect to other memory operations. The order in which memory operations are serviced at the memory is called the *memory order*. All processors observe the same view of this memory order.

2. When granted the access to the memory, a processor executes its memory operations in the order specified by its program, called the *program order*.

3. A read operation returns the value from the most recent write operation (according to the memory order) to the same memory location.

4. Switch arbitration is fair so that memory operations from all processors are eventually serviced.

We emphasize the word "appear" here because actual systems need not strictly implement what is depicted in the conceptual model nor do they have to maintain the stated ordering requirements at all times, as long as any *execution result* produced by these systems can be explained as if it were produced by a hypothetical, strict SC implementation. (An execution result refers to the values returned by the read operations in the execution.) In other words, a system may aggressively perform

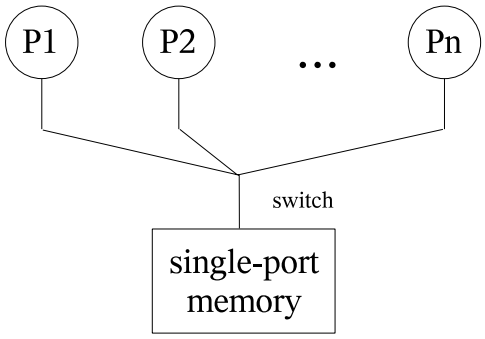
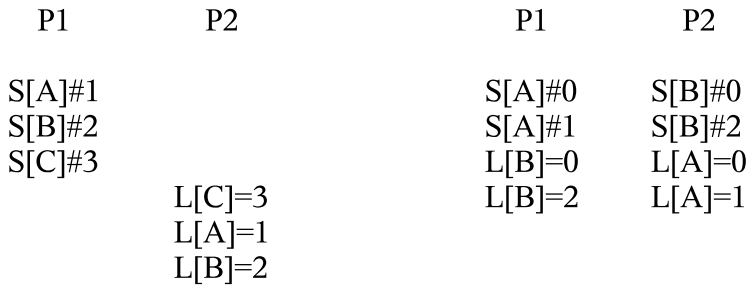


Figure 2.1: A conceptual model of Sequential Consistency (SC).

beyond what is allowed by the memory model provided that programmers will not notice it doing so. An example illustrating this point is shown in Figure 2.2(a) where a strict SC implementation can produce the shown execution result if all memory operations from processor P1 are performed before those from P2. However, this same execution result can still be produced even if S[A]#1 and S[B]#2 (as well as L[A] and L[B]) are executed out of order. Although such reordering would indeed violate the second condition of the SC model above regarding maintaining the program order, it does not lead to an execution result that is not producible by the strict SC implementation. Therefore, programmers can choose to believe that such reordering did not occur. Hence, the memory order that appears to programmers needs not correspond to the actual order.

Unfortunately, determining (both statically and dynamically) whether or when it is safe to deviate from the strict definition is difficult. This effectively rules out several hardware optimizations which are otherwise applicable to sequential uniprocessors such as overlapping accesses to different memory locations and using write buffers to hide write latency. Compiler optimizations affecting memory operations also become severely restricted; complex code analysis has to be performed to determine when such optimizations would be safe (e.g., other processors must not be able to observe the reordering of memory operations, if any). Oftentimes, such analysis has to remain conservative and produce code which is less efficient but it is guaranteed to be correct in all possibilities.



(a) SC (b) Not SC

Figure 2.2: Examples of execution results.
The notation used here is as follows:
S[A]#1 refers to a store operation writing value 1 to a memory location A.
L[A]=1 refers to a load operation reading value 1 from a memory location A.

Finally, note that caches are not included in the SC model since they should be transparent to software. A *cache coherence* protocol governs the propagation of each newly written value among caches such that write operations to the *same memory location* are visible to all processors in the same order. Cache coherence is necessary but not sufficient for Sequential Consistency because the SC model further requires that operations to *all memory locations* must appear to all processors in the same order. As an example, the execution result shown in Figure 2.2(b) is cache coherent but not sequentially consistent. It is cache coherent because load operations for each memory location observe the values changing in the same order as they are written by the store operations. However, there is no memory order that can produce such a result while remaining sequentially consistent. Consider, for example, L[B]=0 performed by processor P1. The fact that it reads value 0 means that it must be performed after P2's S[B]#0 has written the value 0 to the location, but before P2's S[B]#2 overwrites that with a new value. Because operations from P1 (as well as P2) must appear to happen in the program order, both S[A]'s preceding L[B]=0 must have already been performed before L[B]=0 and memory location A will hold the most recent value, 1. This disallows the first L[A] performed by P2, which happens later, to see the already overwritten value, 0.

2.2 Specification of Sequential Consistency

We follow the axiomatic approach introduced by Sindhu et al. [63]. The formal specification describes the semantics of memory operations and constrains the possible behaviors that can be observed for the SC model.

2.2.1 Memory Operations

We consider memory operations which are dynamically executed by each processor during the course of its program. A load operation reads the value from the memory location, and a store operation writes the value to the location. In addition to load and store operations, most processors also provide a swap operation which reads the value from the memory location and immediately writes a new value to it, without intervention from any other memory operations. A swap operation, which is usually helpful for implementing code that synchronizes the execution among processors, is treated as both a load and a store operation. The following summarizes the notation:

L_a^i	a load from location a by processor i.
S_a^i	a store to location a by processor i.
$[L_a^i; S_a^i]$	a swap to location a by processor i; [ ] represents atomicity.
$Val[L_a^i]$	the value read by L_a^i.
$Val[S_a^i]$	the value written by S_a^i.
Op_a^i	either a load or a store.

2.2.2 Orders

An order is defined as a binary relation $\rightarrow$ over a set of operands such that it is:

Irreflexive[1]: $\neg(x \rightarrow x)$

Transitive: $(x \rightarrow y) \wedge (y \rightarrow z) \Rightarrow x \rightarrow z$

Antisymmetric: $x \rightarrow y \Rightarrow \neg(y \rightarrow x)$

The order is total if it is also:

[1]A partial order is typically defined to be reflexive. However, we choose to exclude the case of equality to avoid possible confusion; we will only consider the order of distinct operations.

Trichotomous[2]: $(x \rightarrow y) \vee (x = y) \vee (y \rightarrow x), \forall x, y$

There are two types of orders defined over the memory operations:

1. The *program order* – ";": a per-processor total order that denotes the dynamic sequence of instructions executed by each processor.

2. The *memory order* – "<": a single total order that conforms to the order in which operations *appear* to be performed by memory.

Note that, due to the transitivity, $Op_1; Op_2$ does not necessarily mean that Op_1 and Op_2 are consecutive operations in the program order, that is, there may or may not be another Op_3 such that $Op_1; Op_3; Op_2$. Similar is true for the memory order.

2.2.3 Axioms

The following axioms formally capture the four conditions of the SC model as described earlier in Section 2.1, with an additional axiom to describe the behavior of a swap operation. Below, a and b may refer to the same memory location, while i and j may refer to the same processor.

Order: The memory order is a total order.

$(Op_a^i < Op_b^j) \vee (Op_b^j < Op_a^i) \quad ; Op_b^j \neq Op_a^i$

OpOp: The program order is maintained in the memory order.

$Op_a^i; Op_b^i \Rightarrow Op_a^i < Op_b^i \quad ; Op_b^i \neq Op_a^i$

Value: A load returns the value written by the most recent store to the same memory location according to the memory order.

$Val[L_a^i] = Val[Max_<\{S_a^j | S_a^j < L_a^i\}]$

Termination: All stores eventually terminate. If one processor does a store and another processor repeatedly does loads to the same location, there will eventually be a load that succeeds the store in the memory order.

[2] $x = y$ means x and y are identical, not that they are distinct but equal in rank – this cannot happen when the order is total.

$$S_a^i \wedge (L_a^j;)^\infty \Rightarrow \exists L_a^j \in (L_a^j;)^\infty \text{ such that } S_a^i < L_a^j$$

Atomicity: No memory operations can intervene between the load and store parts of a swap.

$$[L_a^i; S_a^i] \Rightarrow (L_a^i < S_a^i) \wedge (Op_b^j < L_a^i \vee S_a^i < Op_b^j), \forall Op_b^j \quad ; Op_b^j \neq L_a^i \wedge Op_b^j \neq S_a^i$$

Again, it is important to note that these axioms describe the behavior of SC that *appears* to programmers, but *do not* describe or suggest how it should be implemented. This concept is similar to the fact that a uniprocessor can be simply described as being "sequential" regardless of how much instruction level parallelism it actually exploits. As an example in our context, consider the following two alternatives of the Atomicity axiom.

Atomicity-A[3]: *Stores* cannot intervene between the load and store parts of a swap.

$$[L_a^i; S_a^i] \Rightarrow (L_a^i < S_a^i) \wedge (S_b^j < L_a^i \ \vee \ S_a^i < S_b^j), \forall S_b^j \qquad ; S_b^j \neq S_a^i$$

Atomicity-B: Operations from *any processor to the same location* or operations from *the same processor to any location* cannot intervene between the load and store parts of a swap.

$$\begin{aligned} [L_a^i; S_a^i] \Rightarrow\ & (L_a^i < S_a^i) \wedge \\ & (Op_a^j < L_a^i \ \vee \ S_a^i < Op_a^j), \forall Op_a^j \wedge && ; Op_a^j \neq L_a^i \wedge Op_a^j \neq S_a^i \\ & (Op_b^i < L_a^i \ \vee \ S_a^i < Op_b^i), \forall Op_b^i && ; Op_b^i \neq L_a^i \wedge Op_b^i \neq S_a^i \end{aligned}$$

It can be shown that a system implementing either of these two weaker axioms is still perceived by programmers as equivalent to that implementing the original Atomicity axiom [35] (proof outline is in Appendix A). Generally speaking, for verification purposes, it is more convenient and simple to consider the strictest version of the axioms. For deciding on implementation optimizations, system designers may find the most relaxed form of the axioms more useful.

2.3 Relaxing Sequential Consistency

With many restrictions in optimizing the performance of Sequential Consistency, designers of many existing architectures have chosen to violate Sequential Consistency

[3]This version of the Atomicity axiom is used by Sindhu et al. [63].

by relaxing some of its requirements as follows.

2.3.1 Relaxing the Write Atomicity (or Memory Order)

We first consider the relaxation of the requirement that each memory operation must appear to execute atomically with respect to other memory operations. This relaxation usually occurs as a consequence of optimizing the cache coherence protocol. For example, when a processor writes to a memory location, it also has to invalidate (or update, depending on the policy) all the cached copies of this location at other processors, if any. To maintain the atomicity of write operations, or *write atomicity*, each processor must block accesses to the memory location after it observes the invalidation request (or the update request), until it is certain that every processor has also observed the request. By relaxing the write atomicity, the overall system performance can improve, at the expense of violating Sequential Consistency (although the caches remain coherent).

Figure 2.3(a) provides an example of the effect of relaxing the write atomicity. Suppose all processors have a copy of both memory locations, A and B, in their caches. When processor P1 writes to A, processor P2 observes the new value at A before P3 does. This can happen, for example, with a variable latency interconnection network. P2 then proceeds to write to B without waiting for P3 to observe the new value at A. This causes the write to A to be non-atomic. Finally, if the interconnection network does not guarantee that the cache coherence messages for the two memory locations will be delivered in order, P3 may happen to observe the new value at B before the new value at A. This result is a violation of Sequential Consistency because P2 and P3 do not have the same view of the memory order. In P2's view, the write to A is ordered before the write to B, but the order is reversed in P3's view.

While relaxing the write atomicity can effectively cause two processors to have different views of the memory order, relaxing the read atomicity does not affect the memory order the same way. Relaxing the read atomicity usually means that a single read operation at a large granularity is broken into multiple reads at the smallest granularity, between which other memory operations can intervene.

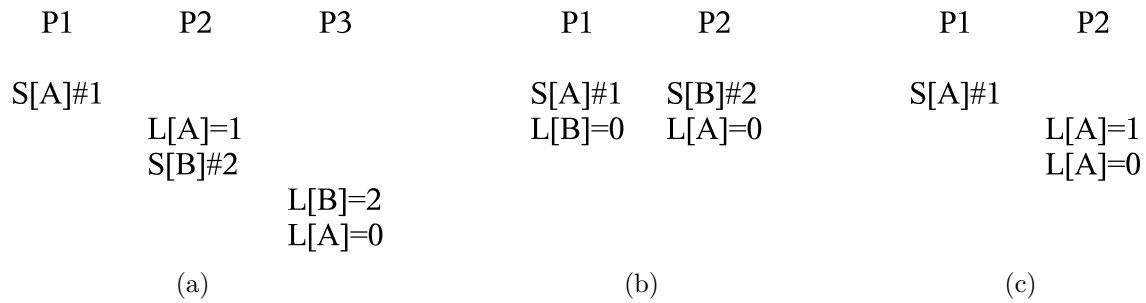


Figure 2.3: Examples of relaxing the SC requirements.
Assume all memory locations initially hold value 0. The notation used here is as follows:
S[A]#1 refers to a store operation writing value 1 to memory location A.
L[A]=1 refers to a load operation reading value 1 from memory location A.

2.3.2 Relaxing the Program Order

The program order requirement can be relaxed in several ways. For example, adding a write buffer to each processor can help hide the latency of write operations by making them non-blocking and allowing them to be executed concurrently with subsequent operations which may then proceed to completion before the write operations themselves. Overlapping accesses to different memory locations and non-blocking reads can further reduce memory latencies. With this type of relaxation, a special type of instruction, called *fence* or *memory barrier*, may be provided to enforce the program order when it is so desired.

To be specific, there are four types of program orders to be considered for the relaxation: Write-to-Read, Write-to-Write, Read-to-Read, and Read-to-Write.

Write-to-Read relaxation is typically due to the addition of write buffers. When the write buffers are not FIFO (First In First Out), Write-to-Write order is also relaxed. It is unusual, however, to relax Write-to-Write order without relaxing Write-to-Read order. Read-to-Read and Read-to-Write orders are normally relaxed together as a consequence of having non-blocking reads and/or out-of-order execution. A relaxed memory model needs not relax all these four types of program orders.

In some memory models, deciding whether two memory operations can be reordered may also depend on their *dependence order* (which will be defined later) and/or whether or not they access the same memory location.

Figure 2.3(b) gives an example of non-SC behavior which may occur when the program order is relaxed. There exists no memory order between these four operations that can create this execution result while preserving the program order of operations from each processor. However, allowing the load operation in processor P1 to overtake its preceding store, for example, can yield some memory orders which produce this execution result; e.g., L[B]=0 < S[B]#2 < L[A]=0 < S[A]#1.

We also note here that the effect of relaxing the write atomicity and the effect of relaxing the program order are not orthogonal. For example, while the execution result shown in Figure 2.3(a) can occur due to relaxing the write atomicity as explained earlier in Section 2.3.1, it can also occur if we instead relax the program order between the two operations from processor P3 (or P2). Figure 2.3(c) shows on the other hand a case which can only occur with relaxing the program order. In this example, assume that a long latency operation is used to determine the memory location to be read by the first load on P2, while the memory location of the second load is immediately known. If the program order between the two loads is relaxed, they may execute out-of-order and the second load may return a value older than what is returned by the first load.

2.3.3 Relaxing the Value Semantics

When the program order is relaxed, the definition of what value a read operation should return may also need an adjustment such that the sequential semantics within each processor appears to be maintained. In other words, the execution result of any single-threaded program should remain sensible, or *self-consistent*, as if there were no reordering. Otherwise, even the task of writing a simple program with no sharing would be difficult.

With write buffers, when a read operation overtakes its preceding write operation made to the same memory location and completes out-of-order, the read should return the value from that particular write even though it may not already be visible to other processors. This feature is called *bypassing* or *read forwarding*. Because the actual memory order in this case will be from the read to the write due to the reordering,

it may seem counterintuitive that the read returns the value from the "future" write. But without this read forwarding, the behavior would be confusing even for programs that do not share memory, which is the more common case.

Condon et al. eliminate the oddity of reading from a "future" write by splitting such write operation into two virtual operations, $ST_{private}$ and ST_{public} [13]. Both virtual operations write the same value and $ST_{private}$, whose value is only visible to its own processor, always precedes its read operations so that they now read the value from the "past" rather than the "future."

2.4 Specifications of Relaxed Memory Models

In this section, we present some relaxed memory models using the axiomatic approach similar to that used to describe the SC model. For each relaxed memory model, we first summarize how it relaxes the SC requirements and then formally describe the effect of such relaxation.

2.4.1 Total Store Order (TSO)

The *Total Store Order* (*TSO*) model is one of the memory models proposed in the SPARC architecture [65, 70]. It relaxes the SC model by adding to each processor a write buffer (FIFO) with read forwarding capability. This is conceptually modeled in Figure 2.4.

Because the write atomicity requirement is still maintained in the TSO model, the memory order remains well-defined and is a total order. For example, we can use the *logical* time when each memory operation completes to represent the memory order. A store completes when its value is written to the memory. A load completes when its returned value is known. Therefore, the **Order** axiom remains the same as in the SC model.

Order: The memory order is a total order.

$$(Op_a^i < Op_b^j) \vee (Op_b^j < Op_a^i) \qquad ; Op_b^j \neq Op_a^i$$

The write buffer allows a load operation to overtake its preceding store operations

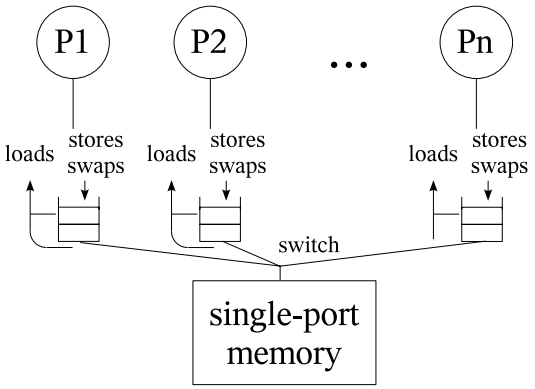


Figure 2.4: A conceptual model of Total Store Order (TSO).

and complete before them in the memory order. Therefore, the program order is not always maintained and we replace the **OpOp** axiom with the following:

LoadOp: The program order from a load to any operation is still maintained in the memory order because a load is blocking.

$L_a^i; Op_b^i \Rightarrow L_a^i < Op_b^i \qquad ; Op_b^i \neq L_a^i$

StoreStore: The program order among stores is maintained in the memory order because the write buffers are FIFO.

$S_a^i; S_b^i \Rightarrow S_a^i < S_b^i \qquad ; S_b^i \neq S_a^i$

Membar: The program order is no longer maintained from a store to its following loads. However, a special instruction called *membar* (memory barrier), denoted as M, can be used to enforce the program order for such case.

$S_a^i; M; L_b^i \Rightarrow S_a^i < L_b^i$

The read forwarding capability allows a load to return a value from the write buffer if the most recent store to that memory location still resides in the write buffer. In this case, the load will complete and appear in the memory order before the store. Therefore, we need to modify the **Value** axiom to include stores which may still reside in the write buffer as potential sources for the value returned by a load.

Value: A load returns the value written by the latest store in the memory order among the stores to the same location which precedes the load in the memory order

(the returned value comes from the memory) and the stores to the same location which precedes the load in the program order (the returned value may come from the write buffer).

$$Val[L_a^i] = Val[Max_<(\{S_a^j | S_a^j < L_a^i\} \cup \{S_a^i | S_a^i; L_a^i\})]$$

The rest of the axioms from the SC model remain unchanged in the TSO model.

Termination: All stores eventually terminate.

$$S_a^i \wedge (L_a^j;)^\infty \Rightarrow \exists L_a^j \in (L_a^j;)^\infty \text{ such that } S_a^i < L_a^j$$

Atomicity: No memory operations can intervene between the load and store parts of a swap.

$$[L_a^i; S_a^i] \Rightarrow (L_a^i < S_a^i) \wedge (Op_b^j < L_a^i \vee S_a^i < Op_b^j), \forall Op_b^j \quad ; Op_b^j \neq L_a^i \wedge Op_b^j \neq S_a^i$$

The execution result shown in Figure 2.3(b) is an example of what could happen under TSO but not SC. This behavior is due to the relaxation of the program order from a store operation to its subsequent load operation.

2.4.2 Processor Consistency (PC)

The *Processor Consistency*[4] (*PC*) model further extends the TSO model by relaxing the write atomicity. The effect that each processor may have a different view of the memory order when the write atomicity is relaxed (as discussed in Section 2.3.1) can be modeled, as shown in Figure 2.5, by assuming that each processor has its own copy of the entire memory. These memory copies are kept coherent on a per-location basis, that is, if we consider only store (write) operations to a given location, their memory order remains the same in every processor's view. This requirement is called the *coherence requirement*.

To model the non-atomicity of a store operation to memory location a issued by processor i, S_a^i, it is replaced by n sub-operations, $S_a^i(1) \ldots S_a^i(n)$, where n is the number of processors. Each memory location a is also replaced by n virtual locations, $a(1) \ldots a(n)$. The sub-operation $S_a^i(j)$ is responsible for the update of location $a(j)$, the virtual copy of location a locally owned by processor j. A load operation, L_a^i,

[4]The Processor Consistency model described here is based on that described by Adve and Gharachorloo [2, 21], which differs from the original definition by Goodman [28, 33].

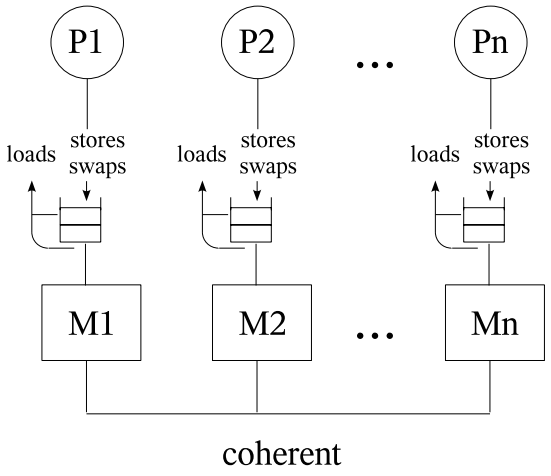


Figure 2.5: A conceptual model of Processor Consistency (PC).

reads the value from location $a(i)$. For uniformity in the naming convention, we may also refer to L_a^i as $L_a^i(i)$ even though there is only one sub-operation for a load. Sub-operations are program ordered in the same way the original operation would be. The order of all sub-operations $Op(i)$ represents processor i's view of the memory order, which can be different from other processors' view. However, when we consider all sub-operations to all memory copies, we still have a single view of the global memory order.

Order: The global memory order is a total order.

$$(Op_a^i(p) < Op_b^j(q)) \vee (Op_b^j(q) < Op_a^i(p)) \qquad ; Op_b^j(q) \neq Op_a^i(p)$$

The coherence requirement is captured by the following axiom:

Coherence: Stores to any given location appear in the same order in every processor's view.

$$S_a^i(p) < S_a^j(p) \Rightarrow S_a^i(q) < S_a^j(q), \forall q \qquad ; S_a^j \neq S_a^i$$

The program order is maintained in the memory order the same way as in the TSO model. We adjust axioms related to the program order, however, to explicitly express the order for all sub-operations.

LoadOp: The program order from a load to any operations is still maintained in the

memory order because a load is blocking.

$$L_a^i; Op_b^i \Rightarrow L_a^i(i); Op_b^i(q) \ \wedge \ L_a^i(i) < Op_b^i(q), \forall q \qquad ; Op_b^i \neq L_a^i$$

StoreStore: The program order among stores is maintained in the memory order because write buffers are FIFO. All sub-operations of the first store appear in the memory order before any sub-operation of the second store.

$$S_a^i; S_b^i \Rightarrow S_a^i(p); S_b^i(q) \ \wedge \ S_a^i(p) < S_b^i(q), \forall p, q \qquad ; S_b^i \neq S_a^i$$

Membar: The PC model does not provide any fence or memory barrier operation. If it does, however, the **Membar** axiom would be similar to that of the TSO model.

$$S_a^i; M; L_b^i \Rightarrow S_a^i(q); M; L_b^i(i) \ \wedge \ S_a^i(q) < L_b^i(i), \forall q$$

The value semantic of the PC model is similar to that of the TSO model except that each processor i looks up its own memory copy, which will be updated only by $S(i)$ sub-operations.

Value: A load performed by a processor returns the value written by the latest store in *that processor's view* of the memory order among the stores to the same location which precedes the load in the memory order (the returned value comes from *the processor's copy* of the memory) and the stores to the same location which precedes the load in the program order (the returned value may come from the write buffer).

$$Val[L_a^i] = Val[Max_<(\{S_a^j(i) | S_a^j(i) < L_a^i(i)\} \cup \{S_a^i(i) | S_a^i(i); L_a^i(i)\})]$$

The **Termination** axiom follows from the SC model with an adjustment to properly address sub-operations of stores.

Termination: All stores eventually terminate.

$$S_a^i \wedge (L_a^j;)^\infty \Rightarrow \exists L_a^j \in (L_a^j;)^\infty \text{ such that } S_a^i(j) < L_a^j(j)$$

While TSO prohibits any operations from intervening between the load and store parts of a swap, PC only prohibits stores to the same memory location as the swap.

Atomicity: In every processor's view of the memory order, *stores to the same location* cannot intervene between the load and store parts of a swap.

$$\begin{aligned}[L_a^i; S_a^i] \Rightarrow \ & (L_a^i(i) < S_a^i(q), \forall q) \ \wedge \\ & (S_a^j(p) < L_a^i(i) \ \vee \ S_a^i(p) < S_a^j(p)), \forall S_a^j(p) \quad ; S_a^j \neq S_a^i\end{aligned}$$

The execution result shown in Figure 2.3(a) is an example of what could happen under PC but not under SC nor TSO. This behavior is due to the relaxation of write atomicity.

2.4.3 Relaxed Memory Order (RMO)

The Relaxed Memory Order (RMO) model is the most relaxed model proposed in the SPARC architecture [70]. It further relaxes the TSO model by making both load and store operations non-blocking in most cases. In addition, each write buffer is virtually split into multiple queues, one for each memory location. Therefore, store operations to different locations can be performed out of order. Unlike the PC model, however, the RMO model does not relax the write atomicity. A conceptual model of RMO is similar to that of TSO in Figure 2.4, except that the program order is much more relaxed in RMO.

With the write atomicity, a single view of the memory order, which is also a total order, can be maintained. Therefore, the **Order** axiom remains the same as in SC and TSO.

Order: The memory order is a total order.

$$(Op_a^i < Op_b^j) \vee (Op_b^j < Op_a^i) \qquad ; Op_b^j \neq Op_a^i$$

Although the program order is much more relaxed in RMO, memory operations still cannot be freely reordered. Determining whether two memory operations from the same processor can or cannot be reordered no longer solely depends on their operation type (load or store), but it may also depend on their *dependence order*, an order adequate to ensure that the execution result is *self-consistent* (i.e., operations from a processor *appear* sequential to itself).

Two memory operations X and Y are dependence ordered, denoted by $X <_d Y$, if and only if they are program ordered, $X; Y$, *and* at least one of the following conditions is true. Note that a swap operation is both a load and a store.

1. Y is a condition that depends on X, and Y is a store. This rule includes all control dependences. It is not applicable to the case where Y is a load because a load has no side-effect and can be speculatively executed.

2. Y reads a register that is written by X. Cases where Y writes a register that is read or written by X are excluded because they are false dependences when register renaming is assumed.

3. X and Y access the same memory location and at least one of them is a store.

4. $X <_d Z$ and $Z <_d Y$. That is, the dependence order is transitive.

For $X <_d Y$, RMO maintains the program order from X to Y in the memory order only when:

- X is a load. If X is a store, it will be placed in the write buffer and bypassed.
- X and Y access the same memory location and Y is a store. This is to maintain cache coherence with the presence of write buffers.

The above statements translate into the following two axioms:

DepLoadOp: An operation dependence ordered after a load is also memory ordered after it.

$L_a^i <_d Op_b^i \Rightarrow L_a^i < Op_b^i$

DepOpStoreSameAddr: An operation dependence ordered before a store to the same location is also memory ordered before it.

$Op_a^i <_d S_a^i \Rightarrow Op_a^i < S_a^i$

Four flavors of memory barriers can be used to enforce the program order.

Membar: Membar enforces the program order. (There are 4 flavors of membars.)

$L_a^i; M_{LL}; L_b^i \Rightarrow L_a^i < L_b^i$
$L_a^i; M_{LS}; S_b^i \Rightarrow L_a^i < S_b^i$
$S_a^i; M_{SL}; L_b^i \Rightarrow S_a^i < L_b^i$
$S_a^i; M_{SS}; S_b^i \Rightarrow S_a^i < S_b^i$

The rest of the axioms are identical to those in TSO.

Value: A load returns the value written by the latest store in the memory order among the stores to the same location which precedes the load in the memory order and the stores to the same location which precedes the load in the program order.

$$Val[L_a^i] = Val[Max_<(\{S_a^j | S_a^j < L_a^i\} \cup \{S_a^i | S_a^i; L_a^i\})]$$

Termination: All stores eventually terminate.

$$S_a^i \wedge (L_a^j;)^\infty \Rightarrow \exists L_a^j \in (L_a^j;)^\infty \text{ such that } S_a^i < L_a^j$$

Atomicity: No memory operations can intervene between the load and store parts of a swap.

$$[L_a^i; S_a^i] \Rightarrow (L_a^i < S_a^i) \wedge (Op_b^j < L_a^i \vee S_a^i < Op_b^j), \forall Op_b^j \quad ; Op_b^j \neq L_a^i \wedge Op_b^j \neq S_a^i$$

The execution result shown in Figure 2.3(a) is an example of what could happen under RMO as well as PC but not under SC nor TSO. While this behavior is possible for PC due to the relaxation of write atomicity, it is possible for RMO due to the relaxation of the program order between the two operations on processor P2 and/or those on P3. Had a membar operation been inserted to prevent such reordering on both processors, this execution would no longer be possible under RMO (but still possible under PC). As another example, the execution result shown in Figure 2.3(c) can occur under RMO but not under SC, TSO, or PC.

The TSO model presented earlier is, in fact, a special case of the RMO model where three of the four memory barriers – M_{LL}, M_{LS}, and M_{SS} – are assumed after every instruction.

2.4.4 Other Relaxed Memory Models

The relaxed memory models we have discussed so far are only examples of many existing models in the literature. For brevity, we will not discuss all of them and refer readers to Adve's and Gharachorloo's tutorial paper for an excellent introduction to various memory consistency models [2]. The original proposals for several memory models are listed in Bibliography: TSO and PSO [63, 65, 70], PC [23, 28], WO (Weak Ordering or Weak Consistency) [16], RC (Release Consistency) [23], Alpha [64], RMO [70], and PowerPC [15].

2.5 Related Work

Adve's and Gharachorloo's tutorial paper provides an excellent introduction to various memory consistency models [2]. Extensive study and discussion on these memory models and related concepts can be found in their dissertations [1, 21].

Relaxed memory models may differ in subtle ways and it is usually helpful to compare them based on possible outcomes which will also allow us to understand how to port programs written for one model to another [21, 32]. For example, as can be seen in this chapter, RMO and PC are strictly weaker than TSO, while RMO and PC are incomparable.

From a hardware designer's perspective, memory consistency models may also be specified or viewed in terms of sufficient conditions that an implementation needs to guarantee, taking into account some specific details of the design such as the type of interconnection network or the cache coherence protocol [39, 54].

Memory consistency models significantly impact the ease of programming a multiprocessor system, as well as the set of hardware and compiler optimizations which may be performed legally. Commercial architectures support a variety of memory models, such as Sequential Consistency (SC), Total Store Order (TSO) and Release Consistency (RC). While SC and TSO present a more intuitive model to programmers, multiprocessors supporting these models need to perform aggressive optimizations to perform as well as those with more relaxed models [27, 34]. Making the several complex elements involved in the design of the memory hierarchy work together to preserve the memory model guarantees is a major challenge for computer architects today.

Chapter 3

TSOtool: Our Methodology for Testing Shared-Memory Multiprocessors

There are several approaches for checking the correctness of shared-memory multiprocessor implementations focusing on the memory subsystem. With formal approaches, verifying that a particular optimization is correct under a given memory consistency model can involve subtle proof methodologies, using automatically or manually generated proofs. Such approaches usually employ a high-level abstraction of the real design to check specific properties of the abstracted implementation. However, they leave the actual implementation of the processor unchecked, which is, in fact, a significant source of complexity and errors in large designs. On the other hand, prior testing-based approaches for multiprocessors are able to test the designs only with programs whose results can be reasoned about a priori or are precomputable. Programs which have data races are generally avoided because multiple legal outcomes may exist for each program due to its timing-dependent nature and a simple architectural, typically not cycle-accurate, model of the processor cannot be used to cross-check the results.

In this chapter, we describe *TSOtool*, our dynamic testing tool which is aimed at solving the above problem associated with the testing approach. Section 3.1 provides

an overview of TSOtool. Section 3.2 describes its operation in more detail. We defer the discussion of our results to Chapter 6. Finally, Section 3.3 discusses related work.

3.1 Overview of TSOtool

TSOtool operates by running a pseudo-randomly generated program with data races on a system under test, observing the values returned by memory read operations, and then checking the observed execution result for validity under the memory model of the machine[1]. TSOtool is able to perform end-to-end checks on a detailed simulation model of the system, or on a real system, using a large space of randomly generated test cases. This approach can expose bugs in the design of the memory system no matter where they may be hiding - for example, in the design of caches, coherence protocols, system interconnects, or memory controllers.

TSOtool was developed to run on commercially available SPARC [65, 70] architecture based platforms running a standard operating system, and it does not need any modifications to either the hardware or the operating system. As a result, we have been able to use it easily and effectively on a variety of multiprocessor systems based on several different SPARC microprocessors. In addition, TSOtool has been used extensively in pre-silicon validation environments. In such environments, TSOtool can optionally use any extra observability to improve the quality of results. In Chapter 6, we will report our experiences using TSOtool, and describe the kinds of bugs we successfully found in the design of several microprocessors and multiprocessor systems, both in the microarchitecture definition and in the implementation of the microarchitecture.

Although the content in this chapter is specific to the TSO memory model and the SPARC architecture, the same approach can also be used, with some modification, to check the compliance of test program runs with other memory models as well as other instruction set architectures.

[1]We originally developed this tool for the TSO (Total Store Order) model and, hence, the name TSOtool.

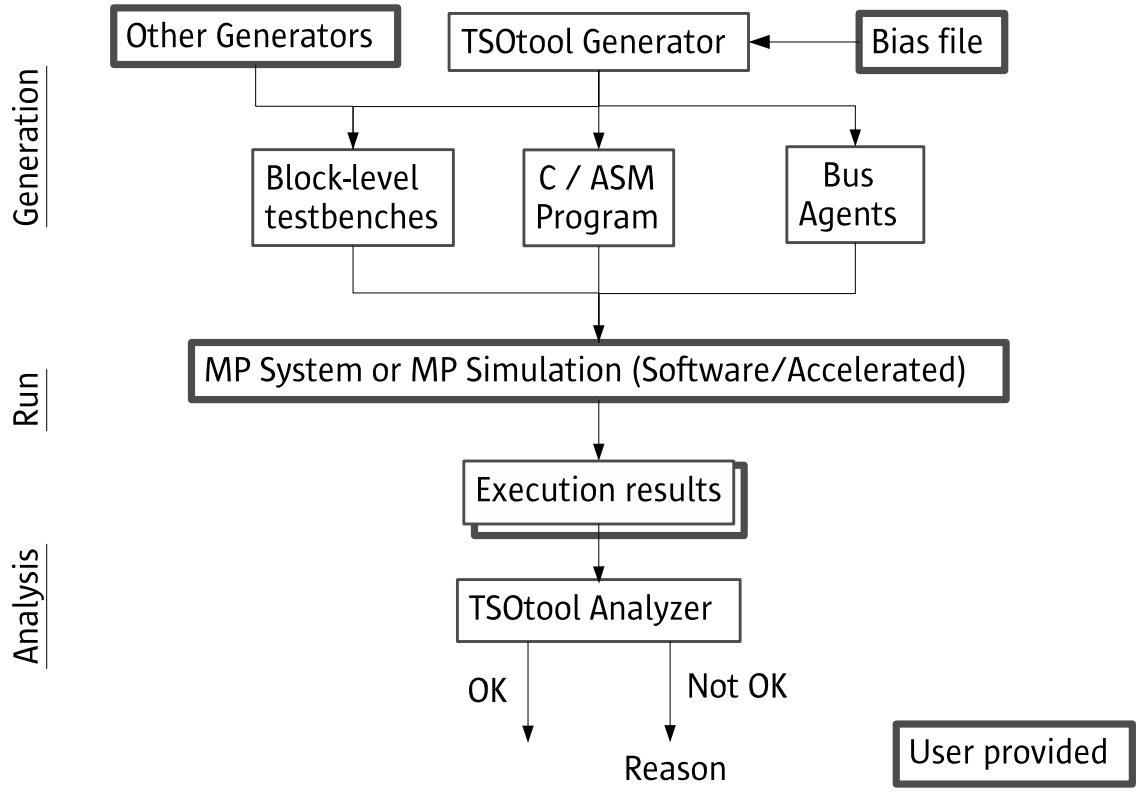


Figure 3.1: TSOtool usage flow.

3.2 TSOtool Operation

There are 3 phases in the TSOtool operation, as illustrated in Figure 3.1.

In Phase 1, TSOtool generates a pseudo-random, multithreaded test program with data races to a relatively small number of shared memory locations. Various properties of the generated program can be controlled by a *bias file* supplied by a user to steer the test generation towards specific kinds of instruction sequences or sharing patterns. The bias file also controls the memory placement such that it may sometimes trigger conflicts in caches or TLBs. The format and syntax of the generated test program is platform dependent.

In Phase 2, the user runs this test program on a platform which supports the desired memory model. If real hardware is available, this environment can be an

actual multiprocessor system, with or without an operating system. It can also be a simulation model of the processor or the system. The simulation models can be at different levels of abstraction, such as architectural, RTL (Register Transfer Level) or gate-level. The simulation may model either the entire processor or only units belonging to the memory subsystem. The verification environment itself can include software simulators, hardware accelerators or FPGA-based emulation machines. We have run TSOtool generated test programs in all of these environments.

In Phase 3, the results of the test program are fed back into TSOtool for analysis. At the end of analysis, a pass or fail is signaled. Note that it is possible that different runs of the same test program may observe different results in the presence of external perturbation (such as operating system activity). Therefore, the analysis result only applies to the correctness of a particular run of the test program.

The rest of this section describes each of these phases in more detail.

3.2.1 Test Generation

In the test generation phase, TSOtool creates a pseudo-random program with data races, based on optional inputs from a user. Users typically get the generator to create a relatively short test with intense sharing. Users can control parameters such as the relative frequency of instruction types, memory layout and loop characteristics. Based on these parameters, TSOtool generates an internal representation of the test program, with each thread represented by a sequence of abstract memory operations. Each abstract memory operation is then mapped to either a set of assembler instructions or a series of instructions in some other language suitable for the test environment. A few adjacent memory operations may be repeated to create loops, in which case the loop count is statically set. Occasionally, we need to randomize events during the test (such as the direction of hard-to-predict conditional branches), so a dynamic software LFSR (Linear Feedback Shift Register) is maintained on each processor and used as a source of random numbers.

Unique store values: Having unique store values in the test program helps TSOtool map every load value back to the store which created it. This feature is

important for the analysis algorithm, as will be explained in the next chapter. We ensure that store values are unique by maintaining two running counters, one in a floating point register and one in an integer register. These counters are used as the source of store values in the test program. The expense of maintaining these counters is minimal - an increment operation for every unique store value.

Load Observability: On physical systems, which provide no additional observability, the test program includes code to observe and save the results of all the load operations in the program. The results are initially buffered in two sets of processor registers, one for floating point results and one for integer results. When a result buffer is full, its contents are flushed to memory. Buffering helps to reduce perturbations in the middle of test operations. In environments where the load results can be observed through other means, code to explicitly save results may not be needed.

Other instructions: In addition to 32-bit, 64-bit and 128-bit loads and stores, some of the other kinds of operations supported by the generator are:

- Memory access instructions to various Address Space Identifiers (ASIs).
- Memory barrier instructions - these require that all previous instructions on the issuing processor are globally visible before the next instruction is issued.
- Various flavors of prefetch, such as prefetch for read-once, write-once, read-many, or write-many. Prefetches may be "strong" or "weak". Strong prefetches may incur TLB miss traps, while weak prefetches are silently dropped in case of a TLB miss. Certain patterns of load accesses can also trigger a hardware prefetch operation in some processors.
- Different types of block load and store instructions which read or write 64 bytes at a time. These have special rules to ensure ordering with respect to other instructions.
- Instructions which flush data from various levels of the cache, or instructions which flush the execution pipeline.

- Compare and swap instructions: A compare and swap (CAS) instruction compares one of its operand with the content in the specified memory location and uses the other operand to perform a swap if the comparison is a match or perform a load if it is a mismatch. To give a CAS a reasonable probability of resolving into a swap, it is emitted with a preceding load of the same size to the same address. The value returned by the load is used as the compare value for the CAS instruction. The compare may still fail occasionally when some store to the same address intervenes between the load and the CAS instructions.

- Non-faulting loads: These are loads which silently return 0 if the address causes a memory fault. For valid memory addresses, the behavior is required to be the same as that of a regular load. Non-faulting loads in the test program are randomly marked to access either faulting or non-faulting addresses.

- Unpredictable conditional branches.

- Sequences of operations which cause cache line replacements and writebacks.

- Inter-processor interrupts.

TSOtool allows users fine-grained control over the test program, as well as the ability to specify desirable sequences of memory operations which are considered likely to exercise known corner-cases in the design, such as a queue in the system becoming full, a hazard condition being created, or specific idioms for uncovering the memory behaviors such as those compiled in ARCHTEST [11]. Users can improve the quality of test cases generated using tools which report test coverage.

3.2.2 Test Run

As mentioned earlier, the generated test program can be mapped to a variety of test environments. On physical systems, we typically run TSOtool on configurations of up to tens of processors with a few thousand memory operations per processor.

In a simulation environment, TSOtool can optionally utilize the additional observability provided by the environment. For example, if the result of load operations

can be directly observed from the simulation, explicit operations to buffer and save them are omitted from the test program.

Simulation environments often have the useful capability to detect errors via runtime checkers monitoring the design. TSOtool can make use of these checkers to detect failures in the course of simulation. In some accelerated simulation environments, however, it is expensive or impossible to observe events in the system or to add runtime checkers. In one such environment, we can improve simulation throughput by a few orders of magnitude by disabling observability features and runtime checkers. In these cases, TSOtool's ability to independently observe the results and analyze them for correctness is very useful.

Systems under test usually have a number of configurable parameters that can alter their behavior in some aspects. This variety should be taken into account for coverage reason. Moreover, some problems, if any, may manifest themselves only in some configuration, or much sooner than in others.

3.2.3 Analysis

The TSOtool analyzer is the key component that differentiates our approach from conventional approaches used in multiprocessor verification. In this analysis phase, the program execution trace will be represented by an *analysis graph*, each node representing one memory operation (not an individual instruction). The analysis graph is formed by unrolling loops and resolving branches in the original program to model the dynamic sequence of memory operations in the test. Nodes representing operations which cover multiple shared words of interest are expanded, so that all loads, stores and swaps in the analysis graph are of a uniform size. The result returned by each load during program execution is attached to the load node.

Before starting analysis, some nodes are pre-processed in the following manner:

- Prefetch instructions, cache or pipeline flushes and cache line replacements and writebacks should have no programmer-visible effects and are ignored for the purpose of analysis.

- Non-faulting loads to illegal addresses are checked for a return value of 0, and then ignored for the rest of the analysis. Non-faulting loads to legal addresses are converted to regular loads.

- Compare and swap instructions are resolved by examining the return value of the instruction. If the CAS completed, the instruction is converted to a swap of the same size, else it is converted to a regular load.

Next, the TSOtool analyzer captures and infers as many relations as possible between memory operations that must hold in order to satisfy the TSO axioms. It accordingly adds edges in the analysis graph to represent the memory order $<$. A cycle in the graph at the end of the analysis signifies a violation of TSO. We describe the algorithm in detail in the next chapter.

The TSOtool analyzer also provides a stand-alone analysis interface through which a program description can be fed along with the values of all loads and stores, and this outcome can be checked for TSO violations. This feature allows us to potentially plug in the results from test programs created by other generators as long as they obey the unique store values requirement. This feature also allows us to analyze hypothetical execution results which are manually constructed or modified from actual execution results, providing a way to exercise one's understanding of the memory consistency model or to perform some helpful analysis while debugging the violations.

3.2.4 Debug

When a TSO violation is detected, TSOtool emits a graphical representation of the relevant area in the analysis graph. The user can click on each edge in the graph to understand the reason for its existence, and hence follow the chain of reasoning used by TSOtool to infer the edge.

TSOtool also emits the analysis graph to a text file in a format comprehensible to users. Users can edit this file and feed it back to TSOtool via the analysis interface if they wish to make an educated guess about which operations are incorrectly reordered or which load result is incorrect and what the correct load result should have been.

This "what-if" analysis is often useful for evaluating the correctness of other possible execution results.

3.3 Related Work

Industrial design teams pay a great deal of attention to functional verification [6, 41, 42, 46, 66]. While formal verification can completely check a design and ultimately prove the absence of design flaws, such guarantee is only with respect to the specified properties; missing or improper properties may still allow problems to escape. Furthermore, formal verification does not scale to large systems with a lot of detail embedded in them and, hence, it is usually employed at block level. Even so, it is not necessarily suitable or beneficial for every block. To complement for the shortcoming of formal verification, pseudo-random code generators are extensively used for the processor verification. Most code generators rely on a self-checking mechanism or an instruction-level simulator to check for correct execution of such programs. However, such checking usually does not work in the presence of data races in multithreaded programs. Therefore, pseudo-random code generators often have to either omit data races entirely, or control the placement of such races carefully. The only error they can check in the presence of data races is an obvious manifestation of a problem like a processor hang, or an error caught by a checker in the simulation environment.

Verification approaches which try to use extra design observability present in simulations to reason about the ordering and outcome of data races are usually tied intimately to design details; they are complex to write and often start with the assumption that the microarchitecture is correct. They are not easily portable across different processor microarchitectures and cannot be used on physical systems where such observability is not available. A simulation-based method developed by Taylor et al. [67] is an example of a verification approach that takes such position. In contrast, TSOtool reasons about correctness at the architectural level, and scales easily across multiple microarchitectures and multiprocessor environments both before and after silicon is available.

Other work aimed specifically at verifying memory models can be broadly categorized into static approaches and dynamic approaches. Static methods of verifying memory consistency models usually depend on formally proving that some model of the system obeys the rules of the memory model [13, 14]. While such methods can find bugs in protocols and optimizations at a high-level, they may miss several bugs which are present only in the implementation. The implementation is a ripe source of bugs, since a high-end microprocessor design consists of millions of lines of code.

Dynamic testing on the other hand can exercise the system in all its details, but is limited to bugs which can be uncovered by the test cases run on the system. ARCHTEST is a program which tries to identify the memory model of a multiprocessor by running a specific set of test cases which look for evidence of various kinds of ordering relaxations [11, 12]. Nalumusu et al. use the tests in the ARCHTEST framework in conjunction with model checking on a Verilog representation of part of the memory system [50]. Both of these approaches require the tests to be fixed idioms, whose outcome can be reasoned about a priori, and cannot work with pseudo-random tests.

Some suggestions have also been made on how checkers for dynamic testing can be implemented in actual hardware [10, 22, 47]. These ideas may further lead to designs which can potentially recover from errors.

An exhaustive approach due to Park and Dill [52] uses an executable specification to enumerate all possible outcomes for small assembler programs under a specified memory model. This approach can be applied to verify the correctness of the assembly language programs including synchronization routines; however, it does not attempt to detect faults in the hardware implementation of the memory model, and does not scale to large programs with thousands of instructions.

Using a constraint graph to model relations (e.g., ordering, causality, and dependency) between memory operations where cycles reflect anomalies is a widely applicable technique in the context of multiprocessing or multithreading [14, 39, 40, 57]. A constraint graph can also be used for analyzing the performance of multi-threaded programs, as shown by Cain et al., to potentially avoid coherence misses that are not required by the memory consistency model [8].

Chapter 4

Algorithms for Verifying Memory Consistency

This chapter discusses the algorithms that are employed by the TSOtool analyzer, the key component that distinguishes our methodology for testing shared-memory multiprocessors, whose task is to check whether an outcome produced by a system on executing a test program complies with the memory consistency model it ought to implement.

Section 4.1 defines the problem and introduces two of its variants; one is in P class while the other is in NP-complete class. Section 4.2 presents our polynomial-time baseline algorithms for solving these two specific variants. The solution for the NP-complete variant is incomplete but nonetheless Section 4.3 further suggests two optimizations that significantly improve the speed of the baseline algorithm for this variant. The incompleteness of the solution for the NP-complete variant is then explained in Section 4.4. Section 4.5 shows, however, that we can efficiently perform backtracking on top of the baseline algorithm and create a complete algorithm. Section 4.6 presents the characterization of these algorithms, indicating their efficiency in practice. Finally, we discuss related work in Section 4.7.

P1	P2
S[A]#1	S[B]#2
L[B]	L[A]

(a) Program

P0	P1
S[A]#1	S[B]#2
L[B]=0	L[A]=0

(b) Execution result

Figure 4.1: An execution result example. Assume initial value of 0 in memory.

4.1 The Problems

First, let us consider a test program shown in Figure 4.1(a). The notation used here and in the rest of the examples is as follows: S[A]#1 refers to a store operation writing value 1 to memory location A. L[A] refers to a load operation reading from memory location A. After executing this test program on a system under test, we are able to determine the value returned by each load operation and we annotate this information back into the program as shown in Figure 4.1(b). L[A]=0 refers to a load operation reading value 0 from memory location A. Our problem here is to determine whether this execution result is allowed by a given memory consistency model.

The problem of verifying whether a multiprocessor test program execution complies with a memory consistency model was first studied by Gibbons and Korach for the Sequential Consistency model [24]. They call the problem *VSC* (*Verifying Sequential Consistency*) and define a number of its variants by:

- Limiting the problem size on some dimensions.
- Providing additional input information:
  - *Read-mapping* is a function that maps each read to the responsible write (one which creates the value that the read sees). This variant of the VSC problem is called *VSC-read*.
  - *Write-order* provides, for each memory location, a total order of when all the write operations to the location are performed. This variant of the VSC problem is called *VSC-write*.

Table 4.1: A summary of complexities of VSC problems.

Variant	**Complexity**
VSC	NP-complete
VSC 2 operations per processor	NP-complete
VSC 2 locations	NP-complete
VSC 3 processors	NP-complete
VSC-read	NP-complete
VSC-write	NP-complete
VSC-conflict	P

– *Conflict-order* provides the memory order for any two memory operations that conflict, that is, they access the same memory location and at least one of them is a write. This is equivalent to having both read-mapping and write-order. This variant of the VSC problem is called *VSC-conflict*.

Table 4.1, reproduced from Gibbons and Korach's full paper [26], summarizes the complexity of the VSC problem and the above variants. Except the VSC-conflict problem, which is in P class, the general VSC problem and other variants are NP-complete. Most of the NP-complete results are proven by a reduction from the *3-Satisfiability* (*3SAT*) problem [20]. The VSC-read problem is proven NP-complete by a reduction from the database *view serializability*[1] problem [51]. Furthermore, the VSC-read problem is NP-complete only with respect to the number of processors (p) and not the number of operations (n), that is, an algorithm with time complexity $O(n^p)$ exists [25].

For relaxed memory models, the work of Gibbons and Korach is, unfortunately, not applicable as is. In this and the following sections, we extend their work by studying the similar problems for relaxed memory models. The most interesting variants of the problem for our purpose are the VSC and VSC-read, since pseudo-randomly generated tests can easily be mapped to these problems and the analysis is performed on architectural results visible to the program. The VSC-conflict problem, though easier to solve because it is in P class, has limited use since write order per

[1]The *conflict serializability* problem is in P class, analogous to the fact that the VSC-conflict problem is in P class.

location is not easily observable in general. We skip the VSC-write problem in our study because we believe that read-mapping can also be observed in the process of observing write order per location. Using the TSO model as an example, we use our terminology in a manner analogous to the above and define the following problems.

4.1.1 The VTSO Problem

Instance: A multithreaded program with:

- Known dynamic memory operation sequences for each thread (processor)
- The memory location and the written value for each operation that has store semantics
- The memory location and the read value for each operation that has load semantics

Question: Are all the TSO axioms satisfied?

Due to the fact that the Termination axiom does not really specify a bound on how long it takes for a written value to be eventually seen by other processors, this axiom cannot be completely checked using finite test cases. Thus, we will omit this axiom from consideration for the rest of the chapter. We reproduce here the rest of the TSO axioms from Chapter 2. The notation used is as follows. The superscript and the subscript may be omitted when they are irrelevant or there is no ambiguity.

L_a^i	a load from location a by processor i.
S_a^i	a store to location a by processor i.
$[L_a^i; S_a^i]$	a swap to location a by processor i; [ ] represents atomicity.
$Val[L_a^i]$	the value read by L_a^i.
$Val[S_a^i]$	the value written by S_a^i.
Op_a^i	either a load or a store.
M	a memory barrier.
;	a per-processor program order
$<$	the memory order

Order: The memory order is a total order.

$(Op_a^i < Op_b^j) \vee (Op_b^j < Op_a^i)$ $\quad ; Op_b^j \neq Op_a^i$

LoadOp: The program order from a load to any operation is maintained in the memory order.

$L_a^i; Op_b^i \Rightarrow L_a^i < Op_b^i$ $\quad ; Op_b^i \neq L_a^i$

StoreStore: The program order among stores is maintained in the memory order.

$S_a^i; S_b^i \Rightarrow S_a^i < S_b^i$ $\quad ; S_b^i \neq S_a^i$

Membar: Membar can be used to enforce the program order from a store to its following loads.

$S_a^i; M; L_b^i \Rightarrow S_a^i < L_b^i$

Value: A load returns the value written by the latest store in the memory order among the stores to the same location which precedes the load in the memory order and the stores to the same location which precedes the load in the program order.

$Val[L_a^i] = Val[Max_<(\{S_a^j | S_a^j < L_a^i\} \cup \{S_a^i | S_a^i; L_a^i\})]$

Atomicity: No memory operations can intervene between the load and store parts of a swap.

$[L_a^i; S_a^i] \Rightarrow (L_a^i < S_a^i) \wedge (Op_b^j < L_a^i \vee S_a^i < Op_b^j), \forall Op_b^j \quad ; Op_b^j \neq L_a^i \wedge Op_b^j \neq S_a^i$

4.1.2 The VTSO-read Problem

The VTSO-read problem is the VTSO problem with additional information, called read-mapping, which maps each load operation to the corresponding store operation which created that read value. A VTSO problem where all written values are unique is in effect a VTSO-read problem.

It can be shown that VTSO and VTSO-read are NP-complete as one can always convert a multithreaded program written for the SC model into a program for the TSO model by inserting a memory barrier after every store operation to enforce the SC behavior. i.e., The VSC and VSC-read problems reduce to the VTSO and VTSO-read problems respectively.

4.1.3 The VTSO-conflict Problem

The VTSO-conflict problem is the VTSO-read problem with additional information specifying, for each memory location, the total order of store operations to the location.

We show that the VTSO-conflict remains in P class by presenting a polynomial time algorithm in the next section.

4.2 Baseline Algorithms

Although the ability to solve the VTSO problem is the most general solution, the lack of crucial information such as read-mapping makes it difficult to design an algorithm that performs well. Therefore, we limit our scope in this work only to the VTSO-read and VTSO-conflict problems. With TSOtool, we impose a constraint on our generated test programs such that each store in a test program writes a different value. This allows us to trivially map each load to the responsible store, and thus gives us the read-mapping function. During pre-silicon validation, additional observability may allow us to determine the total store order per location, leading to the much less complex VTSO-conflict problem.

The following features are common to all algorithms described in this section. A program and its execution result are represented by an *analysis graph*, a directed graph whose nodes represent dynamic operations (loads or stores) in the program, and edges represent ordering relations in the global memory order $<$. Since $<$ is transitive, any path in the graph implies the existence of the $<$ relation between the source and the destination of the path. A legal outcome should not correspond to an analysis graph with cycles, since this would violate the irreflexivity and anti-symmetry properties of $<$. Note that $<$ reflects the perceived memory order, that *appears* to programmers, and does not necessarily correspond to the order in terms of actual time.

A synthetic node is added at the root of the analysis graph acting like a set of stores writing initial values to all memory locations. A set of atomic operations, e.g., a

load-store pair representing a swap, is modeled in the graph by forcing incoming edges incident to any node in the set to point to its first node; similarly, outgoing edges from any node in the set are redirected to leave from its last node. This *Atomicity enforcement* automatically ensures that the Atomicity axiom holds for all relations embedded in the graph at all times. A read-mapping function $w(L)$ maps each load L to the store which writes the value returned by L. A failure is directly signaled if there exists a load reading a value never written to that memory location. An inverse of the read-mapping is also computed and cached in each store node; it represents the set of all loads that read the value written by that store.

We first present a polynomial time algorithm for the VTSO-conflict problem as it is the easier problem because more information is available.

4.2.1 Algorithm for VTSO-conflict

Given an analysis graph representing a multithreaded program and its execution result, the read-mapping function $w(L)$, and the total order of all stores for each location, edges are added using the following rules.

Static Edges: In the first step, program order edges are added to the graph according to the following 3 rules. These edges are independent of execution results.

A1: $L; Op \Rightarrow L < Op$ (LoadOp axiom)

A2: $S; S' \Rightarrow S < S'$ (StoreStore axiom)

A3: $S; M; L \Rightarrow S < L$ (Membar axiom)

For the remaining rules, let S, S', and L be accesses to the same memory location; where $S = w(L)$ and $S' \neq S$.

Observed Edges: For all loads, the edges specified by the following two rules are added based on the load results.

A4: $\neg\ S; L \Rightarrow S < L$ (Value axiom)
This is because S must be in one of the two store sets in the Value axiom for L.

A5: $S'; L \Rightarrow S' < S$ (Value axiom)
This must be true because if assumed otherwise, $S < S'$, and given $S'; L$, L cannot

read the value written by S according to the Value axiom. For this rule, we only need to consider the latest store S' preceding L, because prior stores from the same thread are already ordered before S'.

Conflict Ordering Edges: With the knowledge of the store order, we finally add the following edges, which essentially orders all the conflicting operations.

A6: $S < S'$ according to the known store order per location, which is a total order.

A7: $S < S' \Rightarrow L < S'$; for all L reading the value written by S (Value axiom)
We only need to consider S' that immediately follows S for that location. This rule enforces the Value axiom by making sure that S must be the most recent ($Max_<$) store for L because every store ordered after S will be ordered after L also. If not, $S' < L$ is true[2] and it would be illegal for L to read the value written by S because that value would have already been overwritten by S'.

Theorem 4.1. *After all the edges have been added, a cycle exists if and only if there is a TSO violation.*

Proof: The reasons to justify the existence of edges added by each rule are already given above. If a cycle exists in the analysis graph, $<$ is not a valid order (irreflexivity and anti-symmetry are violated). In other words, the Order axiom is violated and, hence, a TSO violation. If a cycle does not exist in the graph, a topological sort of the graph gives a total order on all operations. We will show that this total order is valid, satisfying all the TSO axioms (we leave the Termination axiom unchecked as noted earlier; it is vacuously satisfied in any case):

Order: The topological sort gives a total order on all operations.

Atomicity: Satisfied by the Atomicity enforcement.

LoadOp, StoreStore, Membar: Satisfied by rules A1 to A3.

Value: We show that $Val[L_a^i] = Val[Max_<(\{S_a^j | S_a^j < L_a^i\} \cup \{S_a^i | S_a^i; L_a^i\})]$.

For each L, let $S = w(L)$. We will show that:

$S = Max_<(\{S_a^j | S_a^j < L_a^i\} \cup \{S_a^i | S_a^i; L_a^i\})$

[2]Because there is a total order on all operations, for $x \neq y$, $\neg(x < y) \equiv y < x$.

Case 1: $S; L$

- Rule A5 ensures that S is $Max_{<}\{S_a^i | S_a^i; L_a^i\}$.
- Rule A7 excludes every S' ordered after S from $\{S_a^j | S_a^j < L_a^i\}$.

Case 2: $\neg\ S; L$

- Rule A5 ensures that $Max_{<}\{S_a^i | S_a^i; L_a^i\} < S$.
- Rule A4 ensures that $S < L$.
- Rule A7 ensures that S is $Max_{<}\{S_a^j | S_a^j < L_a^i\}$.

Therefore, $S = Max_{<}(\{S_a^j | S_a^j < L_a^i\} \cup \{S_a^i | S_a^i; L_a^i\})$ in both cases and the Value axiom holds true. □

It may appear at first glance that rule A5, which adds edges between stores, is not needed since a total store order per location is already provided. However, it is indeed required since the given store order may specify a relation which conflicts with the Value axiom, and this rule will catch such conflict.

Rules A4 and A5 are critical for VTSO-conflict, and they are different from the simple addition of $S < L$ in VSC-conflict. This is the consequence of relaxing the program order from stores to loads and adjusting the value semantics accordingly so that each processor remains self-consistent. In other words, this difference arises from the fact that the Value axiom of SC has only one term, which is: $Val[L_a^i] = Val[Max_{<}\{S_a^j | S_a^j < L_a^i\}]$. Note, however, that these rules, A4 and A5, for VTSO-conflict will also work correctly for VSC-conflict since $S; L \Rightarrow S < L$ in SC and, therefore, the Value axiom of TSO becomes equivalent to that of SC.

Time Complexity: Let n be the total number of nodes (operations) in the analysis graph and e be the total number of edges. Assuming appropriate input format and data structures, the complexity of this algorithm is bounded by the final topological sort, which is $O(n+e)$. e is $O(n)$ because the number of edges that can be added by each rule is bounded by $O(n)$ and, therefore, this algorithm is $O(n)$. This also shows that the VTSO-conflict problem is a P problem.

4.2.2 Baseline Algorithm for VTSO-read

We now present a polynomial-time algorithm for the VTSO-read problem. Note that this problem is NP-complete and, therefore, our polynomial-time algorithm is inherently incomplete (assuming P $\neq$ NP). Section 4.4 provides an intuitive discussion regarding its incompleteness.

Given an analysis graph representing a program with its execution result and the read-mapping function $w(L)$, edges are added using the following rules:

Static Edges: Rules **R1**, **R2** and **R3** are the same as **A1**, **A2** and **A3** respectively.

For the remaining rules, let S, S', L, and L' be accesses to the same memory location; where $S = w(L)$, $S' = w(L')$, and $S' \neq S$.

Observed Edges: Rules **R4** and **R5** are the same as **A4** and **A5** respectively.

Inferred Edges: Even though the store order is not known, we can still infer edges similar to *Conflict Ordering Edges*. These last two rules, which follow from the Value axiom, try to infer the order between operations to the same memory location that may potentially conflict.

R6: $S < L' \Rightarrow S < S'$ (Value axiom)
Assuming otherwise, $S' < S$ (and given $S < L'$) will lead to a contradiction because L' cannot read the value written by S' as it would have already been overwritten by S. Rule R6 can also be viewed as the contrapositive of rule A7.

R7: $S < S' \Rightarrow L < S'$; for all L reading the value written by S (Value axiom)
This is essentially the same as rule A7, except for the complication that, in the VTSO-read problem, we do not know which S' is the one that immediately follows S in the per-location total store order, and therefore we need to apply this rule for all currently applicable stores S'.

To apply rule R6 to every pair of $S < L'$, we can either traverse the analysis graph backward from each L' to search all its predecessors for any S or traverse the graph forward from each S to search all its successors for any L'. Similarly, to apply rule R7 to every pair of $S < S'$, we can either traverse the graph backward from S' or forward from S. We choose to traverse the graph forward for both rules R6 and R7 so that we can apply them together in one graph traversal; starting from each store,

S, we apply rule R6 if the traversal reaches a load to the same location, L', or apply rule R7 if it reaches another store to the same location, S'.

Note that traversing the analysis graph to apply rules R6 and R7 relies on the transitivity of the memory order, $<$, which is still in the process of being derived. To solve this circular dependency, we iterate over these two rules until no further edges can be added to the graph, that is, a fixed point is reached. The graph is then checked for cycles. If a cycle exists, it implies that there is no valid memory order for this execution (because $<$ is no longer anti-symmetric) and, hence, a violation of the TSO model. Figure 4.2 outlines this baseline algorithm for VTSO-read.

Intuitively, this algorithm tries to infer as much information about ordering as possible. The rules in this algorithm are selected such that they can be efficiently implemented; they are not necessarily complete, as will be shown in Section 4.4. Nevertheless, if the total store order per location for every location is available and embedded in the analysis graph, this instance of the VTSO-read problem becomes the VTSO-conflict problem, for which this algorithm is indeed complete.

Time Complexity: A simple, pessimistic upper bound on the time complexity of this algorithm is $O(n^5)$, where n is the number of nodes (operations) in the analysis graph: The number of iterations is bounded by the number of all possible edges, $O(n^2)$, since each iteration adds at least one edge. The time complexity of each iteration is at most $O(n^3)$ since there are $O(n)$ stores, and we need to spend at most $O(n^2)$ time to traverse the graph to find all successors for each store (to apply rules R6 and R7). Thus, the algorithm in this section has polynomial running time independent of the number of processors.

4.2.3 VTSO-read Example

Figure 4.3(a) illustrates an example of a 4-thread program outcome which violates TSO. There are two memory locations involved: A and B. Figure 4.3(b) illustrates graphically how a cycle is formed in the analysis graph:

- First, static edges E1, E2, and E3 are added using rules R1 and R2 which simply establish program order relationships using the LoadOp and StoreStore axioms.

Input: An execution result with per-processor operation sequences consisting of loads, stores, and membars (a swap is a pair of load and store); and a function $w(L)$, which maps a load to the responsible store.

Output: A boolean value indicating whether or not the given execution result complies with the TSO axioms.

```
1: {rules R1, R2, and R3}
2: for all processor p do
3:    for all operation node n of p in the program order do
4:       if n is a load then
5:          add edges from last load and membar to n
6:       else if n is a store or a membar then
7:          add edges from last load, store, and membar to n
8:       end if
9:    end for
10: end for

11: {rules R4 and R5}
12: for all load L do
13:    S ← w(L)
14:    if ¬ S; L then add edge S < L {rule R4}
15:    S' ← last store to same location such that S'; L
16:    if S ≠ S' then add edge S' < S {rule R5}
17: end for

18: {rules R6 and R7 – done in iterations}
19: repeat
20:    for all store S do
21:       for all x such that S < x do
22:          if x is a load from the same location as S and w(x) ≠ S then
23:             add edge S < w(x) {rule R6}
24:          else if x is a store to the same location as S then
25:             add edge L < x for all load L such that w(L) = S {rule R7}
26:          end if
27:       end for
28:    end for
29: until no more edges are added or a cycle is found

30: flag a TSO violation if a cycle is found in the graph
```

Figure 4.2: Baseline algorithm for VTSO-read.

- Next, observed edges E4 to E7 are added by applying rule R4 to all loads in the graph. Note that rule R4 does not create an edge from S[B]#92 to L[B]=92 on P3 because they are on the same processor without any membars in between.
- Next, observed edge E8 is added by applying rule R5 to L[A]=2 on P1.
- Next, tracing the successors of S[B]#91 on P1, we reach L[B]=92 on P3 (through the presence of edge E8) and apply rule R6 to infer edge E9.
- Finally, tracing the successors of S[B]#92 on P3, we reach L[B]=91 on P4 and apply rule R6 to infer edge E10.

A cycle in the graph (shown in bold) is formed by edges E9 and E10 indicating a conflicting order between S[B]#91 and S[B]#92, signifying a TSO violation.

4.3 Optimizations for VTSO-read

In this section, we examine two optimization techniques that can significantly improve the performance of the baseline algorithm for the VTSO-read problem presented in the previous section.

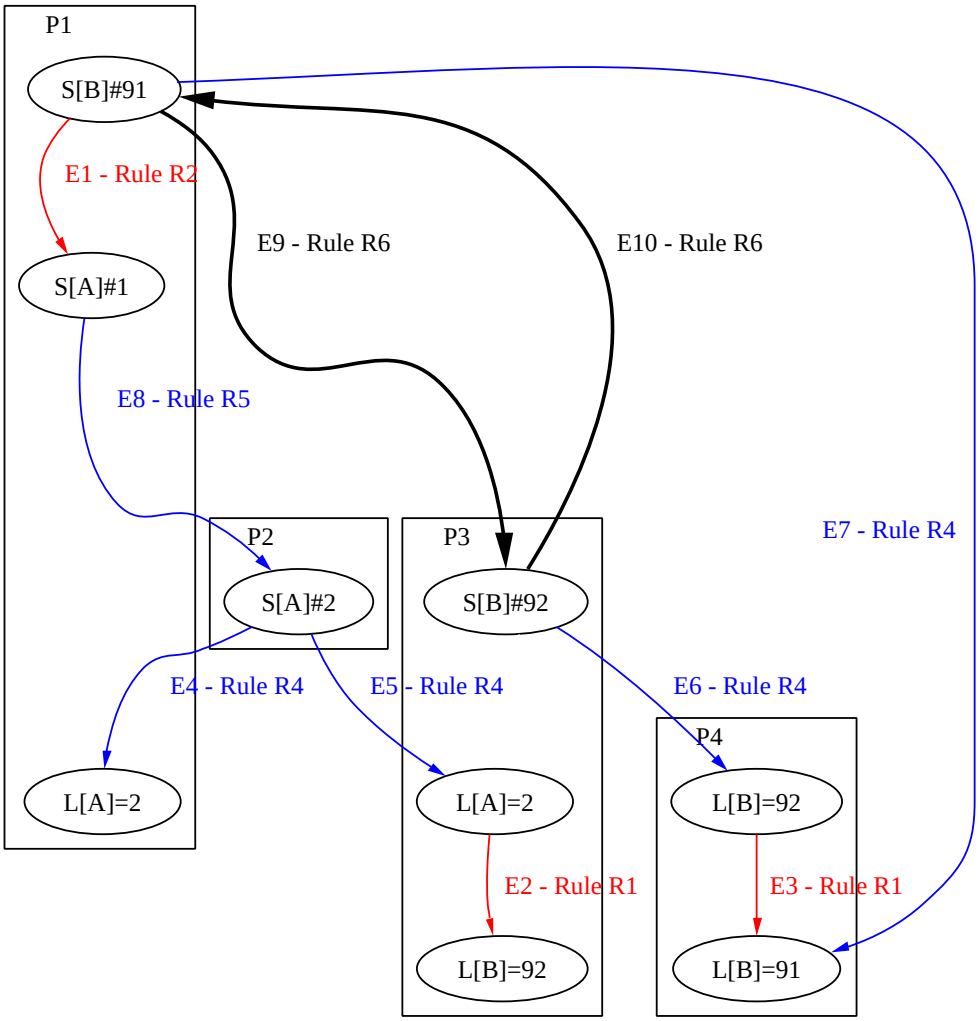
4.3.1 Vector Clocks

The first optimization is based on *vector clocks* and *timestamps* which are widely used techniques in reasoning about distributed computing [5, 37]. They are used mainly for tracking causality (i.e., order) of events in the system. An event can be a sending or receiving of a message or any relevant operation performed by a process[3]. Every process independently maintains its own local timestamp which is incremented on every event occurrence. It also maintains a vector clock which is a vector of timestamps, each element tracking what it knows as the most recent (largest) value of the local timestamp at the corresponding process (including itself). When one process sends a message to another, it also attaches its current vector clock to the message.

[3]For our purpose, a process is a software abstraction of a processor.

P1	P2	P3	P4
S[B]#91	S[A]#2	S[B]#92	L[B]=92
S[A]#1		L[A]=2	L[B]=91
L[A]=2		L[B]=92	

(a) Execution result



(b) Analysis graph

Figure 4.3: An execution result which violates TSO.

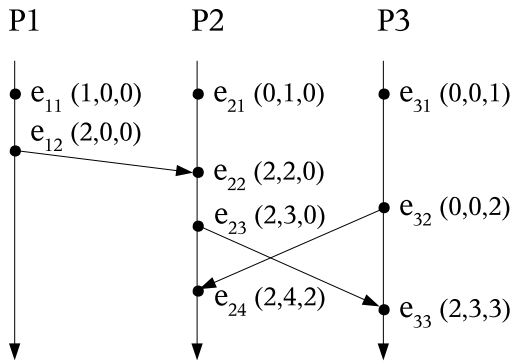


Figure 4.4: Vector clocks example. Events at the source of arrows are message sending, and events at the sink are message receiving. (t_1, t_2, t_3) denotes a vector clock where t_1, t_2, and t_3 are the timestamps on the respective processes P_1, P_2, and P_3 that are currently known to the owner of this vector clock.

The receiver learns from the attached vector clock the most recent timestamps at all processes that the sender is aware of, and merges this information into its vector clock by updating each element to the larger of the corresponding elements from the two vector clocks. Figure 4.4 illustrates how vector clocks are updated. We can quickly determine the order between two distinct events, e_a in process P_i and e_b in process P_j (needs not be distinct from P_i), by comparing the vector clocks maintained by P_i and P_j at the time the two events occur. e_a is ordered before e_b if every element of the corresponding vector clock for e_a is less than or equal to that for e_b, and vice versa. Note that it is also possible that two events cannot be ordered by their vector clocks, e.g., e_{24} and e_{33} in Figure 4.4.

The concept of vector clocks fits well with the SC model where each processor performs its operations sequentially (that is, program order implies memory order). Thus, once we know the memory order between two operations from different processors, $Op^i < Op^j$, we can conclude that all operations from processor j after Op^j are also memory ordered after Op^i and all preceding operations from processor i. Therefore, for a given operation Op^i, our analysis graph needs not have edges from Op^i to all its successors; it can maintain only edges from Op^i to the earliest Op^j such that $Op^i < Op^j$, for all processor j. This helps bound the number of outgoing edges per

node to be at most p, the number of processors. This bounded set of outgoing edges from each node is essentially a vector clock, which is specifically called *reverse time vector clock* because it tracks the earliest successors rather than the most recent predecessors. Note that the vector clocks in our implementation are off-line constructs that exist only during the analysis. They do not involve any maintenance by the hardware or the test program as it is running.

Under the TSO model, however, program order does not imply memory order since a load can overtake its preceding stores. Nevertheless, program order among stores implies memory order, and similarly for loads. Therefore, we can still apply this optimization by splitting the instruction stream of one TSO processor into two *virtual SC* processors; one contains only loads and the other contains only stores. Note that, in TSO, program order $L; S$ also implies $L < S$, and $S; M; L$ implies $S < L$, and we shall represent these ordering constraints with an edge in the analysis graph between such operations which are now separated in the two virtual SC processors.

For more relaxed memory models, we may require more than two virtual SC processors to represent one actual processor, potentially reducing the effectiveness of this optimization technique.

4.3.2 Transitivity

The second optimization is based on an observation on the transitivity of memory order. In the following discussion, we use S, S', and S'' to denote three distinct stores writing to the same memory location, and use L' and L'' to denote two loads reading from S' and S'' respectively. Regarding the application of rule R7, given $S < S'$ and $S'; S''$ (which implies $S' < S''$, under TSO), we need not apply rule R7 to $S < S''$ here because no further information will be inferred beyond what would already be inferred from separately applying rule R7 to $S < S'$ and $S' < S''$. Similar optimization is also applicable to rule R6 although it is less obvious.

Theorem 4.2. *Let L' be the earliest load on a given processor such that $S < L'$ and L' does not read from S, i.e., $w(L') \neq S$. Given $L'; L''$ (which implies $L' < L''$, under TSO), applying rule R6 to $S < L''$ is unnecessary.*

Proof: We show that applying rule R6 to $S < L''$ to infer $S < S''$ is unnecessary because, given the circumstance, we would already know $S < S'$ and $S' < S''$. $S < S'$ is inferred from applying rule R6 to $S < L'$.

Lemma 4.3. *Given $L'; L''$, $S' < S''$ can already be inferred*[4].

Proof. We only need to consider the "adjacent" L' and L'', that is, without another L^* reading from a different store to the same location, S^*, such that $L'; L^*; L''$.
Case 1: $S'; L'$

We will also have $S'; L''$, and by rule R5, we can already infer $S' < S''$.
Case 2: $\neg\ S'; L'$

We infer $S' < L'$ by rule R4, and with $L' < L''$, we have $S' < L''$. Because L' and L'' are "adjacent", the condition in Theorem 4.2 does not apply to this $S' < L''$. Therefore, rule R6 will be applied to $S' < L''$ to infer $S' < S''$. □

Since $S < S'$ and $S' < S''$ are already known, applying rule R6 to $S < L''$ to infer $S < S''$ is unnecessary. □

This observation helps us substantially bound the analysis graph traversal while we iterate over rules R6 and R7 because we no longer have to apply these rules to every $S < L'$ and $S < S'$ for a given store, S. It is sufficient to start from S and look only for the earliest of such L' and S' in each virtual SC processor. Note, however, that the earliest node in a virtual SC processor which is memory ordered after S (according to its reverse time vector clock) does not necessarily access the same memory location as S. Therefore, we add to each node a data structure which allows us to look up its successors in the program order by the memory location they access.

4.3.3 Optimized Baseline Algorithm for VTSO-read

Figure 4.5 outlines the optimized baseline algorithm to iterate over rules R6 and R7, utilizing the presented techniques.

[4]This lemma somewhat reflects the cache coherence.

Input: An execution result with per *virtual SC* processor operation sequences, each sequence consisting of either stores only or non-stores only (the load-store pair of a swap is split accordingly); and a function $w(L)$, which maps a load to the responsible store.

Data structure: An offline *reverse time vector clock* at each node x, denoted by $x.rtvc[]$ where $x.rtvc[p]$ points to the earliest node in *virtual SC* processor p such that $x < x.rtvc[p]$. Initial $x.rtvc[]$ for all x are precomputed with backward topological sort.

```
1: {rules R6 and R7 – done in iterations}
2: repeat
3:   for all store S do
4:     {rule R6}
5:     for all virtual SC processor p which contains only non-stores do
6:       L' ← S.rtvc[p] or its earliest successor in the program order such that
               L' accesses the same location as S, and w(L') ≠ S
7:       if ¬ S < w(L') then
8:         add edge S < w(L')
9:         update S.rtvc[p] and propagate up to all its predecessors
10:      end if
11:    end for

12:    {rule R7}
13:    for all virtual SC processor p which contains only stores do
14:      S' ← S.rtvc[p] or its earliest successor in the program order such that
               S' accesses the same location as S
15:      for all load L such that w(L) = S do
16:        if ¬ L < S' then
17:          add edge L < S'
18:          update L.rtvc[p] and propagate up to all its predecessors
19:        end if
20:      end for
21:    end for
22:  end for
23: until no more edges are added or a cycle is found
```

Figure 4.5: Optimized baseline algorithm for VTSO-read (rules R6 and R7).

Time Complexity: Although the total number of edges in the graph at a given point in time is bounded by $O(pn)$, where p is the number of processors and n is the number of nodes, the number of iterations required in the worst case can actually be more than that because an edge inferred in one iteration may be rendered redundant and removed due to a stronger edge inferred in a later iteration. Thus, the number of iterations is still bounded by the total number of possible edges, which is $O(n^2)$. The complexity of each iteration, however, is now reduced to $O(pn)$ because, for each of $O(n)$ stores, we need to consider applying rule R6 or R7 to at most one successor on each of $O(p)$ virtual SC processors. Hence, the total complexity is $O(pn^3)$. In practice, these optimizations improve the speed of our implementation by almost two orders of magnitude.

4.4 Incompleteness of Baseline Algorithm for VTSO-read

The baseline algorithm for VTSO-read presented in the previous section is incomplete because even when the analysis graph at the fixed point is acyclic, it does not explicitly ensure that the Order axiom is satisfied. Figure 4.6(a) illustrates a case where an existing relation is not inferred by the algorithm; the edges in the graph are depicted at the point when the fixed point has been reached. For clarity, it is redrawn as Figure 4.6(b) by keeping only the relevant elements (edges from a store to corresponding loads reading its value are also omitted to not overcrowd the graph). The notation here is: S[A]#1 refers to a store which writes value 1 to location A, while L[B]=11 refers to a load to address B which reads value 11; P_i denotes operations on processor i. Notice that S[A]#1 and S[A]#2 are left unordered by the baseline analysis. However, we can reason that S[A]#1 $<$ S[A]#2 must be true. If not, S[A]#2 $<$ S[A]#1 by the Order axiom; but with this order and the fact that only one of the two values, either 11 or 12, can survive in location B after S[A]#2, both loads from location B (which are now ordered after S[A]#2) must read the same

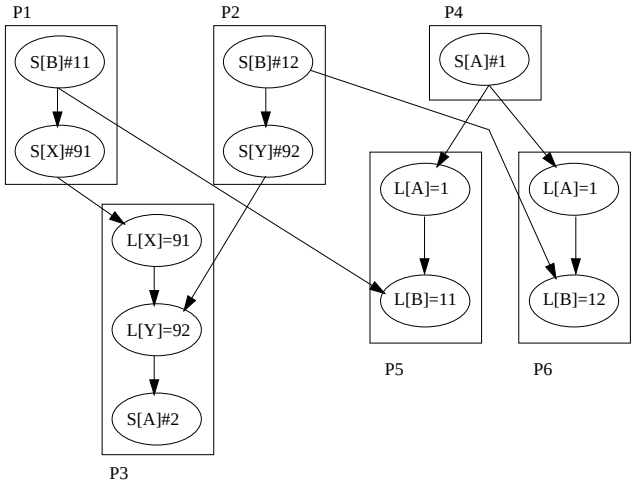
value. This example illustrates a missing relation, but not yet a missed TSO violation; simply adding a similar, mirrored set of nodes to a different location D (two stores to D ordered before S[A]#1, and two loads to D ordered after S[A]#2) creates an instance of a real TSO violation. In this case, the two stores S[A]#1 and S[A]#2 cannot be ordered, but such a violation would be missed by the incomplete baseline algorithm.

One may attempt to design a rule to infer the missing edge in this example. Consider the following hypothetical rule:

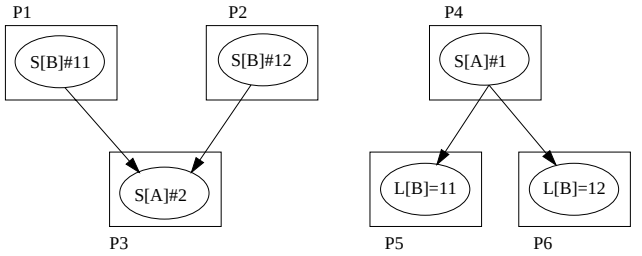
R8: $CommonPred(L, L') < CommonSucc(S, S')$

L and L' are loads to the same location reading different values written by S and S' respectively. $CommonPred(L, L')$ is the latest node that precedes both L and L' in the current snapshot of the memory order being derived, while $CommonSucc(S, S')$ is conversely defined. While this rule will catch the missing edge in the example shown in Figure 4.6(a), it still misses the edge in a slightly modified scenario shown in Figure 4.6(c) because there no longer is a common successor of S[B]#11 and S[B]#12. Note that membars between the store and the load in the same processor are omitted from the picture (or readers may assume the SC model). S[A]#1 < S[A]#2 is a missing edge because, assumed the opposite order, both L[A]=2 nodes will be ordered before S[A]#1 by rule R7, making S[A]#1 the common successor of S[B]#11 and S[B]#12 and, hence, only one value, either 11 or 12, can survive in location B after S[A]#1.

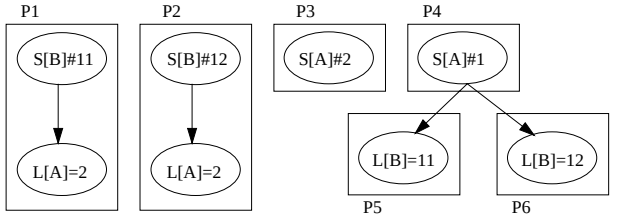
Figure 4.6(d) illustrates that missing edges is not the only form of incompleteness. One can reason that S[A]#2 cannot be ordered before both S[A]#1 and S[A]#3 because that would lead to the same contradiction seen earlier with Figure 4.6(c) in the case when we incorrectly order S[A]#2 < S[A]#1. However, such a constraint cannot be captured in our directed graph representation where we only draw an edge to order two operations when such an order is certain. Despite knowing that S[A]#1 < S[A]#2 *or* S[A]#3 < S[A]#2 (or both) must be true in this example, we can draw neither edge because their presence is not certain when considered individually. To create a TSO violation that would be missed by the baseline algorithm, we can add two similar, mirrored sets of nodes such that none of the stores to location A can be ordered first.



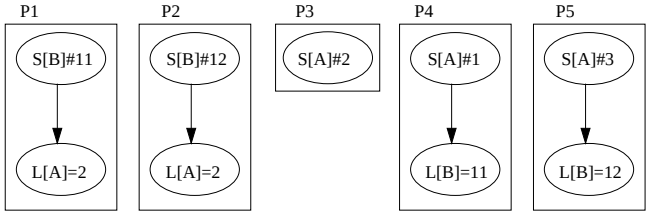
(a)



(b)



(c)



(d)

Figure 4.6: Examples of incompleteness. Membars are omitted.

4.5 Complete Algorithm for VTSO-read

To completely verify TSO compliance, we will attempt to determine if there exists a *total operation order* (*TOO*), which completely orders all operations (loads and stores) in the program, that also satisfies the rest of the TSO axioms.

A simplistic approach to determining if a valid TOO exists would be to perform a topological sort on the analysis graph after the completion of the baseline algorithm, and check if all the axioms still hold (the same baseline algorithm can be conveniently used to determine the validity of a TOO, as earlier pointed out). The topological sort effectively creates an arbitrary "tie-break" decision between operations left unordered by the baseline algorithm. Unfortunately, we have found that most often, this sort does not yield a valid order. This is because when we arbitrarily assign an order between a pair of previously unordered operations during topological sort, it often has ordering implications on other unordered operations; this creates conflicts and usually ends up violating the Value axiom.

Since a straightforward algorithm based on topological sort does not work, we discuss three techniques in the following sections towards improving the chances of finding a valid TOO. In all cases, we assume the baseline algorithm has inferred all its edges and terminated without cycles in the graph.

4.5.1 Heuristic for Topological Sort (*Heu*)

During the construction of a valid TOO, our first heuristic ensures that each store value is completely consumed before it is overwritten, i.e., the topological sort must pick all loads reading the current store value before it can pick another store writing to the same location. This means that every time a store node is picked by the topological sort, rule R7 must be immediately applied to order all its loads before the unpicked stores. An alternative, but equivalent, implementation of this heuristic is to track the *active* store (the most recent store picked by the topological sort) for each memory location and allow the topological sort to further pick only loads that read the values written by the *active* stores or the stores preceding the loads in the program order that have not been *active*. When all loads that read the value written

by an *active* store have been picked, the store becomes *inactive* and a new store can now be picked and made *active*. (For the SC model, this heuristic is similar to the conditions used to determine the validity of frontiers in the $O(n^p)$ backtracking algorithm by Gibbons and Korach [25].)

Time Complexity: A typical topological sort has the complexity of $O(n+e)$ where n is the number of nodes and e is the number of edges, which is $O(pn)$ in this case (because each node has $O(p)$ outgoing edges due to our optimization using vector clocks). In addition, this heuristic may spend $O(p)$ effort to apply rule R7 every time each of $O(n)$ store nodes is picked by the topological sort. Consequently, the total effort for applying rule R7 is also $O(pn)$ and, hence, the total complexity remains $O(pn)$. Note that this time complexity is for a case when the algorithm succeeds in finding a valid TOO. The heuristic may terminate much sooner when a TOO cannot be found, in which case, the analysis is inconclusive and TSO compliance is optimistically assumed.

Although this heuristic is intuitive and fast, we find that it is inadequate; it helps find a valid TOO only when there is relatively low sharing, i.e., p/a is small (where p is the number of processors and a is the number of memory locations). Section 4.6 provides more results.

4.5.2 Deriving Edges During Topological Sort (*Deriv*)

We can extend the heuristic technique in the previous section thus: Each time a store node is picked by the topological sort, rules R6 and R7 are reapplied iteratively to the whole analysis graph until a new fixed point is reached. Careful implementations can minimize the computation by applying the rules only to the affected nodes. (In our implementations, such optimizations are also applied to the baseline algorithm during iteration.)

Time Complexity: Although this heuristic has to go through as many fixed points as the total number of stores which is $O(n)$, the total number of iterations required to apply rules R6 and R7 throughout these $O(n)$ fixed points is still bounded by the total number of possible edges, $O(n^2)$. The complexity per one iteration is $O(pn)$, same

as that of the baseline algorithm. Therefore, the worst-case time complexity remains $O(pn^3)$. Again, this time complexity is for the case when the algorithm succeeds in finding a valid TOO. It may terminate much sooner when this heuristic fails.

Despite the additional effort spent in deriving more edges, this algorithm's effectiveness in finding a valid TOO is still limited with intense sharing. Nevertheless, in practice, it provides significant improvement in TOO completion rate over the previous heuristic.

4.5.3 Backtracking (*Heu+Back*, *Deriv+Back*)

Since the presented heuristics are only best-effort and had unsatisfactory rates of completion, we decided to implement backtracking on top of both the heuristics described above. When the topological sort gets stuck (no instructions can be picked without violating any TSO axioms), instead of giving up, we backtrack to where the most recent tie-break decision was made and pick a different operation to be ordered. Notice that, given two consecutive stores in the memory order, the relative order among loads being performed between the two stores does not matter because the state of memory and all store buffers does not change during this interval. With this observation, we can backtrack directly to where the most recent store was picked by the topological sort.

For the heuristic *Heu*, adding backtracking is relatively simple. Adding the feature to the *Deriv* algorithm is less straightforward because the effect of going through many iterations of deriving edges may be more drastic. We maintain our data structures such that we can checkpoint and undo updates made to the analysis graph. Edges that are derived after a store is picked by the topological sort will be associated with the store. When we backtrack and undo the picking of a store, we remove all the derived edges associated with it and recompute vector clocks for all the affected nodes.

Time Complexity: By using a similar argument to that which Gibbons and Korach use to explain the bounds on their backtracking algorithm based on searching the frontier graph [25], the worst-case complexity of our backtracking algorithms is also $O(n^p)$. At each step during backtracking, the additional cost of finding a new fixed

point incurred by *Deriv* is $O(pn^3)$. This results in $O(n^p \times pn^3)$ in total for *Deriv+Back*. However, in practice, the number and depth of backtracks for *Deriv+Back* is small, resulting in marginal increase in analysis time over *Deriv* which, in return, allows us to achieve a 100% completion rate, when a valid TOO does exist.

4.6 Characterization of Algorithms for VTSO-read

In this section, we characterize our analysis algorithms using real execution results obtained during the testing of a recent multiprocessor system designed and built at Sun Microsystems. Our results show that *Deriv+Back* performs very well; it completely analyzes programs with 512K memory operations distributed across 60 processors and finds a valid TOO for each program within 5 minutes. On average, the analysis time is less than 2.6 times that of the incomplete baseline algorithm which may miss errors. Therefore we find that *Deriv+Back* greatly increases our confidence in the correctness of the results generated by the multiprocessor, and allows us to potentially uncover more bugs in the design than was previously possible.

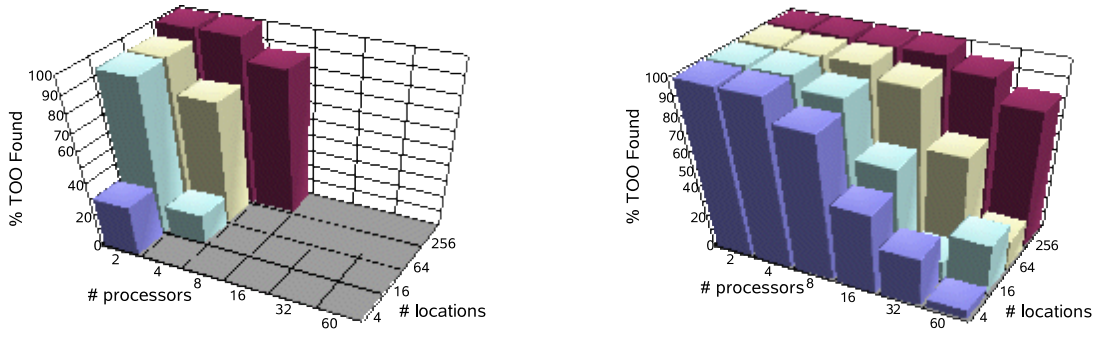
On the other hand, *Heu+Back* (which does not iteratively derive additional edges during the topological sort) does not perform well at all; on all tests except the ones with a small number of processors, it did not finish in a reasonable amount of time. Therefore, we ignore it from further consideration. Also recall that all of the algorithms *Heu*, *Deriv*, and *Deriv+Back* are applied on top of the baseline algorithm, that is, after it has reached a fixed point. Applying them directly, without first running the baseline algorithm, we found they were much less effective, essentially intractable: the effectiveness of *Heu* and *Deriv* in finding a valid TOO reduced drastically and the time spent in *Deriv+Back* exploded as the number of backtracks increased substantially. While we studied these variations for research purpose, we do not consider them interesting and therefore omit their detailed results from this section.

System under test: We performed the following experiments on an actual multiprocessor system designed and built by Sun Microsystems. The system we ran the test programs on has 60 processor cores. Test threads are bound to different processor cores, and run mostly concurrently since the system is quiet except for background

operating system activity. We ran pseudo-random multi-threaded programs with the following instruction mix: 33.3% loads, 33.3% stores, 30% swaps, 1.7% membars, and 1.7% others. We varied the number of threads/processors (p) and the number of memory locations (a) used by the programs, as well as the size of the programs (denoted as n, the total number of memory operations across all processors). The execution results of these programs were saved and later analyzed on a different system based on a previous generation 1.2 GHz Sun's UltraSPARC-III+ processor. For each tuple (n, p, a), 16 different pseudo-random programs were generated, executed, and analyzed. Unless noted otherwise, the presented results are the average over these 16 runs for each tuple. Analysis is the major factor determining test throughput because it takes the most amount of time compared to other steps in our methodology. Running the test itself takes on the order of a few milliseconds per test on a real system.

We also applied our verification methodology to test the same processor design in a pre-silicon software simulation environment. Analysis time, however, was not a concern in this case because simulation time was so much longer that we could, in fact, only use small test programs. Furthermore, software simulators usually scale up to only a few processors and cannot handle large whole-system simulations. Although we can potentially deal with the much simpler VTSO-conflict problem (which is in P) if total store order for each memory location can be observed during simulation, such ordering information is often not readily available in practice because a single point of ordering may not exist in complex systems.

Figure 4.7 shows the effectiveness of *Heu* and *Deriv* in finding a valid TOO for n=128K. (For larger number of operations, n, their effectiveness decreases as expected.) *Deriv* provides significant improvement over *Heu* but it is still incomplete when data sharing is intense. With backtracking, *Deriv+Back* always finds a valid TOO in our experiments. A key finding from our experiment is that when backtracking is necessary, the number of backtracks is at most 75, which is small for the large problem sizes used in our experiments, and the depth of the backtracks is never more than 1 level. This means that the additional overhead due to backtracking is marginal, compared to just running *Deriv*. We also note that the analysis time



(a) % TOO found by *Heu*

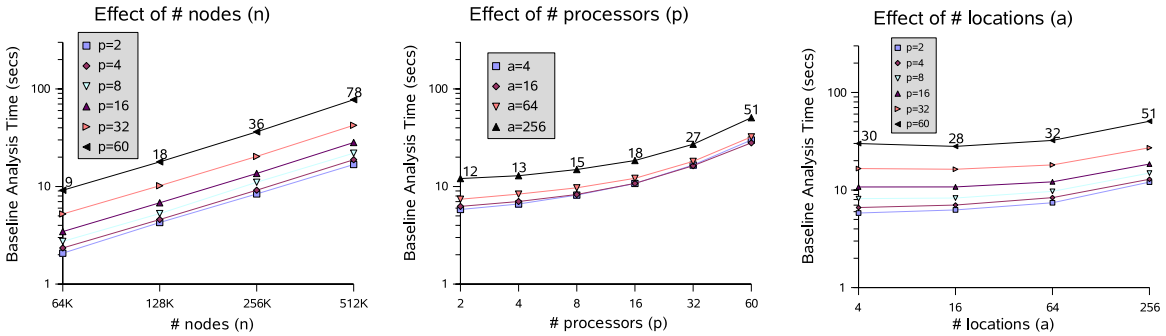
(b) % TOO found by *Deriv*

Figure 4.7: Effectiveness of *Heu* and *Deriv* in finding valid TOO's.

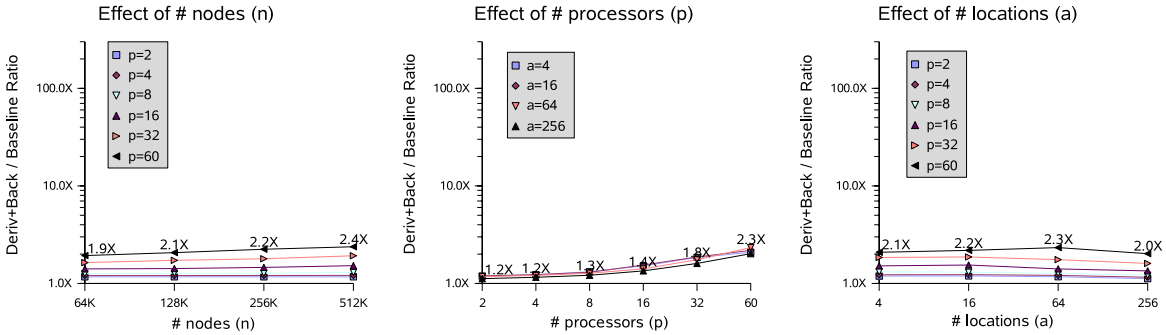
overhead incurred by *Heu* is virtually constant and minimal, about 10%, while the overhead incurred by *Deriv+Back* is significant and grows with all of p, n, and a. Analyzing the largest test programs in our experiment, with n=512K, p=60, and a=256, takes, on the average, 118% more time than the baseline algorithm for cases that require backtracking (while *Deriv* would take 108% more time for cases not requiring backtracking, just a slightly smaller overhead). With a lower processor count (16 and below), the analysis time overhead is usually less than 80% over the baseline algorithm.

We deem the extra overhead in terms of analysis time worth the extra assurance that the program results are indeed TSO compliant, especially for large processor configurations where the errors may be subtle and test methodologies are limited.

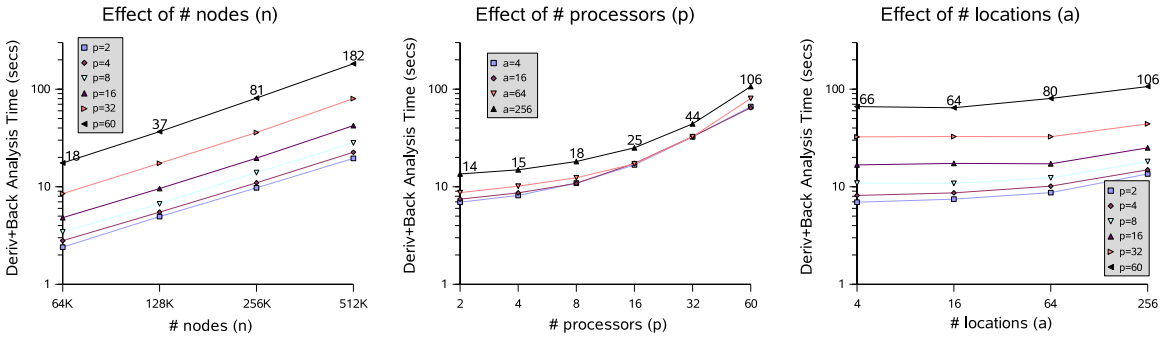
Figure 4.8 shows the effect of n, p, and a on the analysis time. The absolute analysis time of the baseline algorithm and *Deriv+Back* are plotted in Figure 4.8(a) and 4.8(c) respectively. Our algorithms scale linearly with n and more than linearly with p but are relatively less sensitive to a. Figure 4.8(b) shows the ratio of the analysis time of *Deriv+Back* over the baseline. Since the graphs are plotted using log scale over the same range on Y-axis, we can view Figure 4.8(c) as being the superposition of Figure 4.8(a) and 4.8(b). As can be seen, the slope in Figure 4.8(b) is less than that in Figure 4.8(a), which means the increasing analysis time seen in



(a) Analysis time of *Baseline*.



(b) Ratio of analysis time of *Deriv+Back* over *Baseline*.



(c) Analysis time of *Deriv+Back*.

Figure 4.8: Analysis time of *Baseline* and *Deriv+Back* vs. n (averaged over a), p (averaged over n), and a (averaged over n).

Table 4.2: Baseline analysis time and slowdown ratio of *Deriv+Back* for n=256K, averaged over p and a.

	LD-biased	LD-ST equal	ST-biased
Baseline (seconds)	14.9	16.5	17.5
Slowdown ratio	1.45	1.73	2.05

Figure 4.8(c) are dominated by the increasing analysis time in Figure 4.8(a). This interpretation suggests that our backtracking technique can scale reasonably (as long as the baseline algorithm scales).

We also repeated the same experiments using 2 other instruction distributions in the pseudo-random test generator: one biased toward load instructions, with 50% loads and 16% stores, and the other biased toward store instruction, with 50% stores and 16% loads (percentages of other instructions were kept the same). On the average, as the percentage of stores increases, we find that the analysis takes more time. Both the absolute analysis time of the baseline and the slowdown ratio of *Deriv+Back* are affected, as shown in Table 4.2.

We conjecture that a higher store density requires longer analysis time for *Deriv+Back* because there are potentially more values that are not observed at all, and hence, the baseline algorithm can infer fewer relations which would be helpful for *Deriv+Back* during backtracking. With no loads at all, on the other hand, the analysis would run very quickly because any ordering would be acceptable under TSO axioms. Therefore, we expect a tipping point, as we bias the test more towards stores, where the runtime starts to decrease.

Although *Deriv+Back* has not discovered any bugs in the real system that are missed by the baseline algorithm, we tested it with TSO violations based on the examples in Figure 4.6, and it successfully found the missed cycles as expected. Being a backtracking algorithm, however, it cannot avoid the exponential analysis time complexity for such cases. We expect to explore other heuristics in order to find a smaller portion of an execution trace that contains TSO violations.

4.7 Related Work

The problem of verifying whether a multiprocessor test program execution complies with a memory consistency model was first studied by Gibbons and Korach for the SC memory model [24]. They called the problem *Verifying Sequential Consistency* (*VSC*) and proved that the basic VSC problem is NP-complete with respect to the number of operations in the program, as are several variations of the problem, when the number of processors is unbounded. In particular, the *VSC-read* problem, which assumes the presence of a mapping function of every load to the store which creates the value, is also NP-complete when the number of processors is unbounded and Gibbons and Korach propose an algorithm based on searching a frontier graph which has a worst case running time of $O(n^p)$ for n operations and a fixed number of p processors [25]; however, this algorithm is impractical for realistic values of p. The *VSC-conflict* problem, which is the VSC-read problem augmented with the total store order per location, however, is in P. Cantin et al. define the *Verifying Memory Coherence* (*VMC*) problem similar to VSC, except that VMC involves only one memory location [9]. They show that VMC is also NP-complete, but, unlike VSC-read, VMC-read is in P.

The most interesting variants of the problem for our purpose (TSOtool) are the VSC and VSC-read, and their extension to other memory models, since pseudo-randomly generated tests can easily be mapped to these problems and the analysis to be performed is based only on architectural results visible to the program. The VSC-conflict problem, though easier to solve, is not very useful on real systems, since store order per location is not easily observable in general.

Vector clocks have been used in several works related to the verification of Sequential Consistency. Cain and Lipasti employ them in a distributed algorithm to dynamically verify correctness of program execution with respect to SC [7]; however, these vector clocks are online and need to be maintained for each processor and at each shared memory location by some additional hardware. Plakal et al. use offline vector clocks to statically verify that a directory-based protocol implements Sequential Consistency [54]. Based on the proofs in this work, Meixner and Sorin propose

some additional hardware mechanism, involving online vector clocks, in the processor and the memory hierarchy that can dynamically verify SC compliance [47]. In contrast to all these works, our approach is a post-mortem dynamic verification and we construct our vector clocks offline, imposing no overhead on the test program or hardware implementation.

Taylor et al. use a set of informal rules, which is similar and can be considered a subset of ours, to reason about ordering of events in test execution [67]. However, their work is microarchitecture dependent and the completeness or efficiency of their algorithm is not described.

Detecting data races, which occur when one thread accesses a memory location while another thread is modifying it without proper synchronization, is a related problem to verifying Sequential Consistency. In particular, the underlying question in these two problems is to determine whether the order of events during an execution of a program obey some specific rules. Adve et al. and Gharachorloo et al. have also shown that certain relaxed memory models will appear as though they were SC while executing a program that is written with sufficient synchronization, i.e., a data-race-free program [3, 23]. This establishment allows techniques for one problem to benefit the other [22, 53].

Chapter 5

Transactional Memory

Transactional memory (*TM*) has recently been gaining interest from researchers in architectures and programming of multiprocessors as it is a promising concept for developing high productivity computing platforms where the task of writing parallel programs can be simplified, making them more reliable, while their performance can also be enhanced. These premises are critical in mobilizing the paradigm shift toward multiprocessors which, apparently, are becoming inevitable.

Section 5.1 discusses the motivation for transactional memory. Section 5.2 provides a brief survey of proposals for implementing it. Given the subtle issues involved with concurrency and atomicity, it is important that transactional memory systems be carefully specified, thus, we present in Section 5.3 a formal specification of a realistic TM system based on an axiomatic framework similar to that employed in the earlier chapters. Then, we present in Section 5.4 how the methodology and the analysis algorithms in the previous chapters, originally developed for testing traditional memory consistency models, can be extended to testing TM. Finally, we discuss related work in Section 5.5.

5.1 Motivation for Transactional Memory

Parallel programs often require proper synchronization between processes or threads to ensure correct behavior. Such synchronization is usually in the form of providing

mutual exclusion between different execution streams via acquisition and release of locks. Unfortunately, lock-based synchronization has a number of programmability disadvantages as well as performance problems in scaling to large systems [31, 62]:

- *Priority inversion* occurs when a lower-priority process is preempted while holding a lock needed by higher-priority processes.
- *Convoying* occurs when a process holding a lock is descheduled, preventing other processes requiring the same lock from making progress.
- *Deadlock* can occur if processes attempt to acquire the same set of locks in different orders. Deadlock avoidance can be awkward, particularly if the set of locks is not known in advance.
- An "unhappy tradeoff between concurrency and comprehensibility" exists; while coarse-grain lock-based algorithms are relatively simple, they may allow less concurrency than fine-grain ones.

To eliminate these problems, Herlihy and Moss proposed an implementation called *transactional memory* (*TM*) which can be used to provide atomicity in the context of a multiprocessor [31]. In transactional memory systems, programmers can declare a custom block of code a transaction whereby all its operations *appear* as if they either execute atomically or never execute. Historically, database designers have studied transactions in great detail and have been concerned about the issues of Atomicity, Consistency, Isolation and Durability (ACID); these issues, except perhaps durability, are concerns in transactional memory systems as well. In particular, the term "atomicity" in TM context usually refers to both atomicity and isolation.

5.2 Flavors of Transactional Memory

Although direct hardware support for transactional memory has not yet been materialized in practice, it has nevertheless been an area of active research in parallel

architectures and algorithms. Several forms of transactional memory have been proposed [4, 29, 31, 48, 49, 58, 61, 62], spanning over a large design space. There are many design concepts and criteria that distinguish these proposals [49, 62]:

- Transaction atomicity guarantees may be provided by software only, hardware only, or some combination of both hardware and software. Software TM has the advantage of having great flexibility in implementations and without requiring any special hardware. However, it has not readily been deployed in practice mostly because its performance is not always competitive and it usually requires that memory accessed by transactions be statically declared and allocated, a constraint that may not easily be met in practice. Hardware TM, on the other hand, takes no or less impact on performance, and it can freely access memory in general. Implementing hardware transactions is typically done as an extension to the mechanism for maintaining cache coherence. With resource constraints in hardware, however, there usually is a limit on transaction size, both in terms of space (the number of accessed memory locations) and time (how long a transaction can remain active). Supporting unbounded transactions may involve the design of additional hardware mechanisms [4], or it may be done through a combination of hardware and software; for example, falling back to software transactions when hardware transactions fail [48], or using the availability of bounded hardware transactions to accelerate implementations of software TM [62].

- Two transactions *conflict* when their executions overlap, they access the same memory location, and at least one access is a write operation. When such a conflict arises, at least one of the transactions must abort and roll back. *Conflict detection* may be performed immediately on every memory access or it may be deferred until later, e.g., when transactions commit. Conflict detection is called *eager* for the former case and *lazy* for the latter.

- Because a transaction may have to roll back, both new and old values in memory locations it modifies need to be maintained during its execution. *Version*

management in a TM system is called *eager* if the new values immediately replace the old values, which are then backed up in a separate storage, or *lazy* if the new values are separately buffered and only replace the old values when a transaction successfully commits.

- *Contention management* defines how conflicts are dealt with. For example, a transaction conflicting with others may choose to always abort itself, always abort others, or abort the younger transactions, etc.

- The granularity of conflict detection may be at the word level (often at the cache line level in most implementations) for systems that support conventional programming, or at the object level for systems that support object-oriented programming, typically software TM systems.

- Transactions may be nested. In some TM systems, nested transactions are simply flattened out, in which case, all of them must commit successfully in order for the outermost to also succeed. On the contrary, some other TM systems may allow inner transactions to fail without failing the outer ones and losing all the work.

- Transactions may be allowed to coexist with non-transactional memory accesses. This allows migration to TM systems to take place gradually.

It is easy to see that in return for a simpler programming model, transactional memory imposes a greater burden on the system designer. Commercial shared memory multiprocessors today are already very complex machines involving, for example, hardware multi-threading, several levels of caches and multiple coherence protocols. TM implementations may require several other complexities like transaction caches, speculative writes, atomic reads and writes to hardware state, commit broadcasts, etc. Given the subtleties involved with preserving ordering and atomicity guarantees to the programmer, while still allowing a high degree of parallelism for good performance, it is clear that aggressive verification is imperative to ensure that such a system works reliably.

5.3 Formal Specification of Transactional Memory

In this chapter, we extend our methodology and analysis algorithms presented earlier for traditional memory consistency to transactional memory. In order to do so, we first need a formal specification of TM defined in a similar way.

We allow a transactional memory program to have both transactional and non-transactional memory operations. Non-transactional operations are governed by traditional memory consistency rules, except that they may not *appear* to intervene between operations within a transaction in the global memory order. This models a realistic multiprocessor system since it is likely that a system with support for transactions will still need to support existing non-transactional code for that instruction set architecture, as long as the memory locations accessed by transactional and non-transactional instructions are non-intersecting. Of course, transactional memory systems which require that all instructions be part of a transaction are a special case of our formulation. We also make the assumption that reordering between transactions on the same processor as well as reordering of instructions within a transaction are not allowed. Only committed transactions are important for the purposes of verification of architectural results, since aborted transactions are assumed to have no programmer-visible effect on memory. We do not require that nested transactions are flattened. In some systems, inner transactions may abort without aborting the outer ones. However, the fact that the outermost transaction successfully commits should still indicate that all instructions including the successful inner transactions execute atomically. Therefore, nested transactions still *appear* to programmers as a single transaction and this is how we will model them.

The notation used in our formal specification is as follows. The superscript and the subscript may be omitted when they are irrelevant or there is no ambiguity.

L_a^i	a load from location a by processor i.
S_a^i	a store to location a by processor i.
$Val[L_a^i]$	the value read by L_a^i.
$Val[S_a^i]$	the value written by S_a^i.
Op_a^i	either a load or a store.

M	a memory barrier.
;	operator for per-processor program order.
$<$	operator for global memory order.
[ ]	a transaction boundary (no nesting).

An order is a relation that is irreflexive, anti-symmetric and transitive. Two kinds of orders are used in the definition of these axioms: ";" denotes a per-processor program order, and "$<$" denotes a global memory order in which operations *appear* to be performed by memory. $[Op_1; Op_2]$ denotes that Op_1 and Op_2 are inside the same transaction, but due to the transitivity of ";", they are not necessarily consecutive operations in program order, that is, there may or may not be Op_3 such that $[Op_1; Op_3; Op_2]$. Similarly, $[Op_1]$ does not imply that Op_1 is the first, the last, or the only operation in the transaction. Moreover, without explicitly specifying Op_1 inside a transaction boundary, Op_1 may refer to either a non-transactional or transactional operation.

The following are the axioms for a TM system employing the TSO memory model for non-transactional operations[1]:

TransOpOp: Program order within a transaction implies memory order.

$[Op_1; Op_2] \Rightarrow Op_1 < Op_2$

TransMembar: Memory barriers are implicit around each transaction.

$Op_1; [Op_2] \Rightarrow Op_1 < Op_2$

$[Op_1]; Op_2 \Rightarrow Op_1 < Op_2$

TransAtomicity: No other memory operations can intervene between two consecutive operations in a transaction.

$[Op_1; Op_2] \wedge \neg[Op_1; Op; Op_2] \Rightarrow (Op < Op_1) \vee (Op_2 < Op)$

Viewing a swap instruction as a two-instruction transaction, the Atomicity axiom in the TSO model is subsumed by the TransAtomicity axiom.

Order: The memory order is a total order.

$(Op_a^i < Op_b^j) \vee (Op_b^j < Op_a^i) \qquad ; Op_b^j \neq Op_a^i$

[1] Other memory consistency models can also be incorporated using this framework.

LoadOp: The program order from a load to any operation is maintained in the memory order.

$L_a^i; Op_b^i \Rightarrow L_a^i < Op_b^i \qquad ; Op_b^i \neq L_a^i$

StoreStore: The program order among stores is maintained in the memory order.

$S_a^i; S_b^i \Rightarrow S_a^i < S_b^i \qquad ; S_b^i \neq S_a^i$

Membar: A memory barrier ensures that a store preceding it in program order is globally ordered before a load succeeding it in program order.

$S_a^i; M; L_b^i \Rightarrow S_a^i < L_b^i$

Value: A load returns the value written by the latest store in the memory order among the stores to the same location which precedes the load in the memory order and the stores to the same location which precedes the load in the program order.

$Val[L_a^i] = Val[Max_<(\{S_a^j | S_a^j < L_a^i\} \cup \{S_a^i | S_a^i; L_a^i\})]$

This version of the Value axiom permits optimizations that are allowed by the TSO memory model (a load can see the result of a store on the same processor before that store has completed in global order); however, it is also correct for a system using only transactions or for a system with sequentially consistent semantics for non-transactional operations.

Termination: All stores eventually terminate.

$S_a^i \wedge (L_a^j;)^\infty \Rightarrow \exists L_a^j \in (L_a^j;)^\infty$ such that $S_a^i < L_a^j$

All of the above axioms together specify the behavior of a realistic TM system using TSO semantics for non-transactional operations. The TransAtomicity, TransOpOp, TransAtomicity, Order, Termination and Value axioms completely specify a transactions-only system (without explicit memory barriers and inbuilt atomic swap operations), while the Order, Atomicity, Termination, Membar, LoadOp, StoreStore and Value axioms specify a traditional multiprocessor system based on the TSO memory model.

As earlier pointed out in Section 2.2.3, it is important to note that these axioms describe the behavior of a TM system that *appears* to programmers, but *do not* describe or suggest how it should be implemented. For example, accesses to different

memory locations within a transaction may be reordered as long as their dependence order is maintained.

5.4 Transactional Memory Verification

Our methodology to test an implementation of transactional memory is a relatively straightforward extension of our prior work for testing the traditional memory consistency model. We discuss the specific treatment of TM in the generation and analysis phases.

5.4.1 Test Generation

We generate a pseudo-random multiprocessor test program with both transactional and non-transactional operations which access a relatively small number of shared memory addresses. Transactional and non-transactional operations are controlled to access non-intersecting sets of addresses if so required by the TM system. The test case is instrumented to observe the architectural results of running the test, such as the value read by each non-transactional load instruction or each load instruction in a committed transaction. On a real system or in a hardware emulation environment, these results can be buffered in processor registers in order to minimize test perturbation, and only flushed to memory when the register buffer gets full. In some simulated systems, the simulation environment has a means to obtain these architectural results without any instrumentation overhead. To minimize overhead, the value written by every generated store instruction is statically determinable and does not have to be explicitly stored as part of the results. In order to allow us to map each read value observed in the program back to the store which created it, we ensure that each store value used in the program is unique. This is an important requirement for the analysis algorithm, as will be explained in the next section.

Various properties of the generated program such as instruction mix, statistical distribution of transaction length, number of shared memory addresses, sequences of instruction patterns, etc., can be controlled by the user. The test generator needs to

be aware of the specific types of instructions of the TM system, e.g., the mechanism to begin, commit or abort a transaction, but is otherwise fairly portable, especially when it is targeted to generate test programs in a high level language, such as C. The test can includes all operations (including non-transactional operations) supported by the instruction set. For example, for a typical instruction set architecture, it would include different-sized loads and stores, compare and swap, prefetches, flushes, conditional branches, non-faulting loads, inter-processor interrupts, non-cacheable operations, etc.

For a transaction which aborts (either due to an explicit abort instruction in the test, or due to contention with another transaction), the test case may forever retry the transaction, for a TM system that is expected to guarantee forward progress; therefore, a test which fails to complete before a timeout is considered an error. For a TM system that does not guarantee forward progress of transactions, a transaction which remains aborted after some number of retries will be re-executed one last time without the transaction semantics, becoming a sequence of non-transactional operations. In such a system, the status indicating whether or not each transaction commits successfully will be recorded and passed to the analysis phase.

5.4.2 Analysis

The architectural results of the test program, comprising a description of the dynamic order of all its operations (including transaction boundaries and whether they successfully committed) and the values read and written by all loads and stores, are fed into an analysis algorithm. No other visibility into the test execution is assumed, nor are any specifics about the implementation of the TM system, for instance, whether it is done entirely in hardware, software or a combination of the two. Additional ordering information can be used if it is available, however; for example, in a hybrid hardware-software TM system, software may be able to record some global ordering information. At the end of analysis, a pass or fail is signaled. Since it is possible that different runs of the same test program may obtain different results in the presence of external perturbation, the analysis result refers to the correctness of only that

particular run of the test program.

Similar to Chapter 4, we call the problem of verifying that an execution of a test program complies with the specification of transactional memory *Verifying Transactional Memory* (*VTM*), with two variants of interest: *VTM-read* and *VTM-conflict*. VTM-read is VTM with the *read-mapping* function which maps each load to the store responsible for the value it reads. VTM-conflict is VTM with the *conflict-order* which provides the memory order for any conflicting memory operations – two memory operations conflict when they access the same memory location and at least one of them is a store. Furthermore, it is easy to see that VSC (Verifying Sequential Consistency) and its variants are special cases of VTM and its variants where every operation is made a transaction.

In the context of databases, VTM-read is analogous to the *view serializability* problem, which is an NP-complete problem, while VTM-conflict is analogous to the *conflict serializability* problem, which is a P problem [51].

5.4.3 Analysis Algorithms

For analysis purpose, the dynamic sequence of program instructions on each processor is converted to a sequence of nodes in a graph, called the *analysis graph*. Transactions which aborted do not appear in this graph since they should have no programmer-visible effect. If they happen to cause undesired architectural side-effects, we may be able to detect such discrepancy during the analysis. Nodes representing instructions which do not have programmer visible effect on memory such as prefetches and flushes are also removed. Compare and swap instructions are resolved into either a swap or an ordinary load. Edges in this graph represent the memory order $<$. Note that $<$ reflects the order that *appears* to programmers, and does not necessarily correspond to the order in terms of actual time.

A synthetic node is added at the root of the analysis graph acting like a set of stores writing initial values to all memory locations. *TransAtomicity enforcement* is a key aspect of our analysis algorithm with respect to transaction atomicity: incoming edges incident to any node in a transaction must point to its first node; outgoing

edges from any node in a transaction must similarly leave from its last node. This guarantees that the TransAtomicity axiom holds for all relations embedded in the graph at all times. A read-mapping function $w(L)$ maps each load L to the store which writes that value. A failure is directly signaled if there exists a load reading a value never written to that memory location. An inverse of the read-mapping is also computed and cached in each store node; it represents the set of all loads that read the value written by that store.

Baseline Algorithm

The baseline algorithm adds edges by applying the following rules.

Static Edges: In the first step, program order edges are added to the graph according to the following 6 rules, which depend only on the test program and are independent of run results. The first three rules are related to transactions. The next three capture TSO ordering requirements for non-transactional operations.

T1: $[Op_1; Op_2] \Rightarrow Op_1 < Op_2$ (TransOpOp axiom)

T2: $Op_1; [Op_2] \Rightarrow Op_1 < Op_2$ (TransMembar axiom)

T3: $[Op_1]; Op_2 \Rightarrow Op_1 < Op_2$ (TransMembar axiom)

R1: $L; Op \Rightarrow L < Op$ (LoadOp axiom)

R2: $S; S' \Rightarrow S < S'$ (StoreStore axiom)

R3: $S; M; L \Rightarrow S < L$ (Membar axiom)

For the remaining rules, let S, S', L, and L' be accesses to the same memory location; where $S = w(L)$, $S' = w(L')$, and $S' \neq S$.

Observed Edges: For all loads, the edges specified by the following two rules are added based on the load results.

R4: $\neg S; L \Rightarrow S < L$ (Value axiom)
This is because S must be in one of the two store sets in the Value axiom for L.

R5: $S'; L \Rightarrow S' < S$ (Value axiom)
This must be true because if both $S < S'$ and $S'; L$ are true, L cannot read the value

written by S according to the Value axiom. We only need to consider the latest store S' preceding L, because prior stores from the same thread are ordered before S'.

Inferred Edges: The last two rules try to infer the order between operations to the same memory location which, however, involve different data values.

R6: $S < L' \Rightarrow S < S'$ (Value axiom)
Assuming otherwise, $S' < S$ (and given $S < L'$) leads to a contradiction because L' cannot read the value written by S' as it would have already been overwritten by S.

R7: $S < S' \Rightarrow L < S'$ (Value axiom)
Assuming otherwise, $S' < L$ (and given $S < S'$) leads to a contradiction and L cannot read from S.

Rules R1-R7 are exactly the same rules from the baseline algorithm for TSO in the previous chapter, and rules R6 and R7 are similarly repeated until the analysis graph reaches a fixed point. Cycles in the graph, if any, indicate that there is no valid memory order for this execution and, hence, a violation of transactional memory semantics.

Time Complexity: Same as the baseline algorithm for TSO, $O(n^5)$, where n is the number of nodes in the analysis graph (the total number of operations). Same algorithmic optimizations also apply, resulting in $O(pn^3)$, where p is the number of processors.

Complete Algorithm

The *Deriv+Back* algorithm from the previous chapter can be straightforwardly extended to transactional memory, with special attention on the following points:

- TransAtomicity enforcement must be applied at all times.
- When backtracking is needed, we need to backtrack to the most recent store and, if it is inside a transaction, backtrack further to undo the picking of the entire transaction. This may result in unpicking more than one store.

Time Complexity: Same as the original *Deriv+Back*, $O(n^p \times pn^3)$.

5.4.4 Example

Figure 5.1 is an example of producer-consumer synchronization with a single producer and a single consumer. This synchronization can be achieved without locks: the producer checks the flag, produces data, and set the flag; the consumer checks the flag, consumes the data and reset the flag. However, this lock-free mechanism relies on the premise that accesses to data and flag shall not be reordered, either by hardware or software (e.g., due to a programmer mistake). With transactional memory, the ordering constraint in software can be overlooked once the critical sections are embedded in transactions. This makes programming TM systems less error-prone and, hence, attractive. The notation for this example is as follows: S[d]#11 refers to a store which writes value 11 to location d (data), while L[f]=1 refers to a load from location f (flag) which returns value 1. Figure 5.1(a) shows the code with data and flag being accessed in the correct order. An example of possible outcomes is annotated with the code sequence. In Figure 5.1(b), the consumer accesses the data and flag in the opposite order by mistake. Under the TSO model, this code may produce undesirable results such as that is annotated herein, a case where the value 11 produced by P1 is lost. Embedding this same code in transactions, however, will preclude such undesirable results. Figure 5.1(c) illustrates why the result shown in Figure 5.1(b) is not valid under the TM model: edges E1, E2, and E3 are derived via rule R4, E3' is created by TransAtomicity enforcement on the dashed edge E3, and the cycle is formed by E2 and E3' (shown in bold).

5.4.5 Characterization of Algorithms for VTM-read

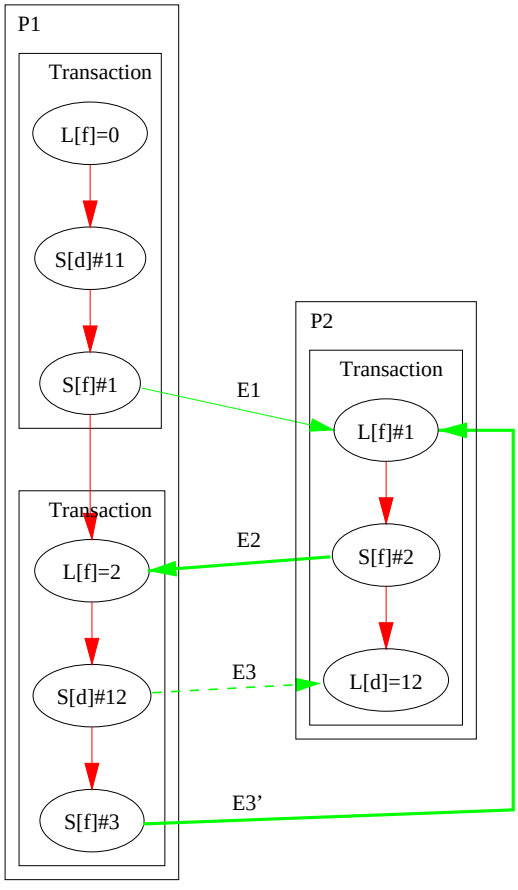
We characterize our analysis algorithms on a research prototype of a transactional memory system called *Transactional memory Coherence and Consistency* (*TCC*) which is being developed at Stanford University [29]. In TCC, all operations are always contained within transactions. TCC is currently available in the form of a detailed software architectural simulator, which is a C++ application designed to be linked directly with the simulated program. Its software libraries provide API interfaces to handle transactions for programs developed in C/C++. Since TCC

P1	P2
L[f]=0	
S[d]#11	
S[f]#1	
	L[f]=1
	L[d]=11
	S[f]#2
L[f]#2	
S[d]#12	
S[f]#3	

(a) Correct code for TSO

P1	P2
L[f]=0	
S[d]#11	
S[f]#1	
	L[f]=1
	S[f]#2
	*L[d]=**12***
L[f]#2	
S[d]#12	
S[f]#3	

(b) Flawed consumer P2 – undesirable result but TSO compliant



(c) Flawed consumer P2 – undesirable result not TM compliant

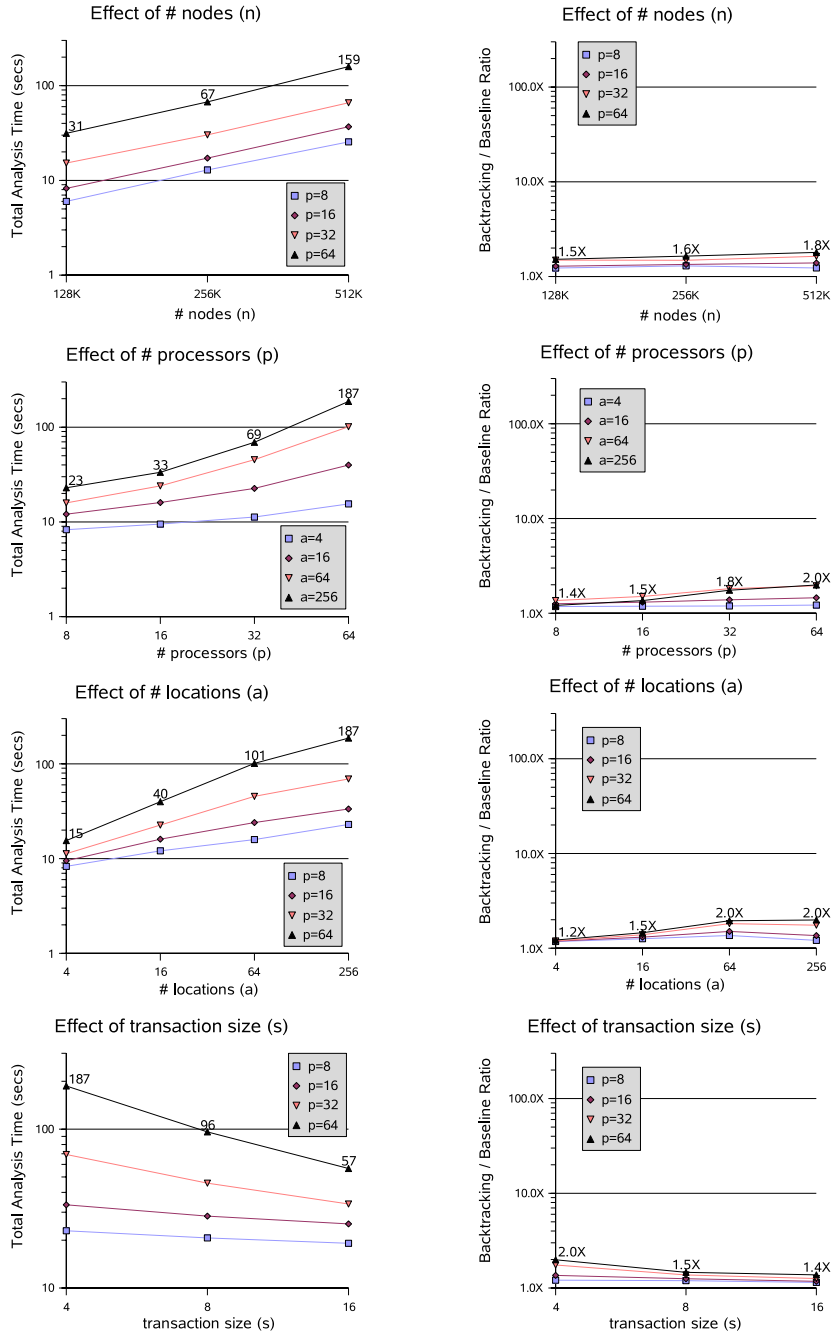
Figure 5.1: Producer-consumer example (using increasing flag values to make store values unique).

programs consist only of transactions, we never utilize the mix of transactional and non-transactional operations that our analysis is capable of handling in these experiments.

Our test generator generates a C program for TCC based on various generator controls like instruction distributions and transaction size. The C program contains memory operations in multiple threads to shared addresses, including transaction boundaries and instrumentation to observe the result of every load instruction in the program. This program is compiled with a C compiler and linked with the TCC libraries and the TCC simulator. The resulting binary is executed and it generates a dump file containing the values returned by load operations made to the shared memory. These load values are, then, extracted from the dump and annotated back to the program to form an execution result to be fed to the analysis algorithm.

For our experiments, we had access to 2 models of TCC which we will refer to as *TCC-A* and *TCC-B*. TCC-A was the first TCC model and was fairly mature and stable. TCC-B extends TCC-A in several ways making it a more scalable and aggressive design. At the time of our experiments, TCC-B was running many programs correctly, but was still in a relatively early phase of development. Examples of transactional memory semantics violations on TCC-B are discussed in Chapter 6. Therefore, we performed the characterization of our analysis algorithms using only TCC-A.

We fixed the sizes and configuration of the memory hierarchy to a reasonable setting, and varied only the number of operations in each test program (n), the number of processors (p), the number of shared memory location (a), and the size of each transaction in terms of the number of memory operations (s). Figure 5.2 summarize the effect of these variables; the absolute analysis time of the backtracking algorithm is plotted in Figure 5.2(a), while the performance difference between the baseline and the backtracking algorithm, shown as the slowdown ratio, is plotted in Figure 5.2(b). In all cases, the backtracking algorithm takes at most twice the amount of time spent in the baseline algorithm to achieve a complete analysis using the backtracking algorithm. The corresponding graphs in this figure are plotted using the same log scale on the Y-axis to illustrate that the slowdown due to the addition of backtracking is not a major contribution to the increasing analysis time when we vary



(a) Analysis time of *Deriv+Back*. (b) Slowdown ratio.

Figure 5.2: Analysis time of *Deriv+Back* and its slowdown ratio. The graphs for n, p, and a are plotted with s=4 and averaging out the parameter that is not shown in each respective graph. The graph for s are plotted with a=64 and averaging out n.

different parameters, i.e., the slopes of the graphs in Figure 5.2(b) are significantly lower than those in Figure 5.2(a).

The total analysis time grows with all parameters except for the transaction size where the analysis time actually shrinks. There are 2 probable reasons:

- The effective number of nodes is smaller with transactions because each transaction is effectively a single node due to the TransAtomicity enforcement.
- The effective number of addresses is also smaller because a single transaction may access several addresses. Consider the following analysis:

  Define:

  $a_t(s)$ = the expected number of addresses *touched* by a transaction of length s.

  $a_e(s)$ = the *effective* number of addresses = $a/a_t(s)$.

  We can state $a_t(s)$ as a recurrence relation:

  $$a_t(s) = a_t(s-1) + \frac{a - a_t(s-1)}{a}; a_t(1) = 1$$

  The second term is the probability that the last instruction in the transaction will touch an address that is previously untouched. This assumes equal probability of each address being accessed.

  The solution of the above recurrence relation is:

  $$a_t(s) = \sum_{i=0}^{i=s-1} (\tfrac{a-1}{a})^i$$

  For simplicity, however, it is not too inaccurate to estimate $a_t(s) \approx min(a, s)$. As can be seen, $a_e(s)$ is always less than or equal to a.

This also helps explain our observation that backtracking is required mostly only for test programs with the smallest transaction size in our experiments.

5.5 Related Work

To our knowledge, our work is the first to outline a practical methodology to test implementations of transactional memory.

Xu et al. propose a technique that essentially captures the programmer's intent and infers critical sections, i.e., transaction boundaries, in multithreaded programs from their dynamic execution paths, and use the inferred information to detect serializability violation during the execution [72]. Also, Adve and Hill study data race detection [3] which is a similar problem to detecting violations of transactions serializability. However, both these works assume that the order of synchronization events or conflicting operations is known or observed, while our work does not need to know transaction ordering. Furthermore, the consistency of the values read in critical sections is typically not checked in data race detection.

Conventional approaches for testing atomicity and ordering are based either on test-cases limited to specific idioms whose results can be reasoned about in advance, or on extraction of global ordering information using extra observability in the system. Extracting global order at the hardware level tends to be complex, especially in large systems which use aggressive optimizations to maintain parallelism while preserving the illusion of transaction atomicity. Furthermore, it likely depends on knowledge sepcific to the design, making the technique not portable across different processors and systems.

Chapter 6

Results

This chapter presents the results of applying our testing methodology to several shared-memory multiprocessor systems designed and built at Sun Microsystems. We also successfully apply our methodology to a research prototype of transactional memory called *Transactional memory Coherence and Consistency* (*TCC*) which is being developed at Stanford University [29].

6.1 Testing TSO Implementations

TSOtool has found numerous bugs during both the design simulation and silicon bring-up processes on various types of processors and systems at Sun Microsystems. In this section, we present the results of deploying TSOtool on 6 different microprocessor designs, without explicitly identifying each one of them for proprietary reasons.

Sun currently has several teams working towards the development of both new generation and derivative microprocessors. Despite employing widely different architectural techniques, all of these processors and systems support the TSO memory model. This is not surprising since switching to a more relaxed memory model introduces potential incompatibility problems with existing software, and is therefore almost as difficult as making changes to the instruction set.

A major advantage of our approach, which reasons about correctness at the memory model interface without incorporating knowledge of implementation details, is

Table 6.1: Classification of bugs found by TSOtool on various processors.

CPU	Architecture	Design	Monitor
CPU1	0	3	0
CPU2	0	3	3
CPU3	0	11	8
CPU4	0	17	8
CPU5	2	20	5
CPU6	5	14	1
Total	7	68	25

that it allows us to deploy TSOtool on several different processors and systems efficiently. Even in cases where we use the observability in simulation environments to extract load results, the dependence on the details of the environment is usually quite small. Most environments usually support a mechanism to trace the dynamic instruction sequence executed, along with the architectural results of each instruction. This is sufficient for TSOtool to obtain the load values read by the test program.

Table 6.1 lists the number of bugs in six processor designs based on the SPARC architecture that were uncovered by TSOtool from its inception in late 2001 until the end of year 2003. During this time, many designs were still in pre-silicon phase and most of these bugs were caught by regression suites of relatively short test programs, about 100 to 1000 memory operations per thread, with 1 to 4 threads. This total of 100 bugs can be classified based on whether they were architecture bugs (the design worked as intended, but the microarchitecture specification was wrong), design bugs (the designer missed a corner case which violated the specification), and monitor or checker bugs (the problems were in runtime checkers or other parts of the simulation environment). Most of these bugs involved complex interaction between multiple functional units and require a detailed understanding of the design to root-cause.

CPU1 to CPU4 are derivative processors based on an earlier design that include significant changes and enhancements in cache hierarchy, memory controller and bus interface. The core pipeline remained unchanged in all these derivatives. In these derivatives, TSOtool did not expose architecture bugs (since the architecture was

Table 6.2: Bugs found by TSOtool in various functional areas.

CPU	Pipe	Caches	TLB	Ld/St Unit	Mem. Ctlr	Inter-connect
CPU1	0	3	0	0	0	0
CPU2	1	4	0	0	1	0
CPU3	0	17	0	0	0	2
CPU4	0	8	0	0	8	9
CPU5	3	11	6	4	0	1
CPU6	0	5	0	10	0	0
Total	4	48	6	14	9	12

already stable), but did find bugs in the design and the verification environment. CPU5 and CPU6 are completely new designs and in these cases, TSOtool uncovered architecture bugs which were overlooked by the architects.

Excluding the 7 architecture bugs, Table 6.2 shows the classification of the bugs in terms of functional units. For CPU1 to CPU4, the presence of bugs was mainly in the cache units, memory controller and bus interface units, consistent with the derivative nature of these processors. (The CPU2's bug in the pipe unit is a monitor problem, not a design problem.)

To illustrate the nature of bugs which TSOtool found, consider the example in Figure 6.1(a), illustrating relevant operations from a bug found by TSOtool during the silicon bring-up process in one of the processors. Also shown in the figure is the analysis graph indicating how the TSO memory model is violated:

- Applying rule R1 to order loads according to the program order, and the fact that SWAP is both a load and a store, gives edge E1 - SWAP $<$ LD. Note that we can model SWAP as a single node due to the *Atomicity enforcement*.

- Applying rule R4 to order between the store that creates the value and the loads (including SWAP in this case) that read the value gives E2 - BST $<$ SWAP and E3 - BST $<$ LD.

- Applying rule R5 to LD on P2 which does not read the value from its preceding SWAP, which means that the value read by LD is newer, gives E4 - SWAP < BST, forming a cycle with E2. (Alternatively, edge E4 can be viewed as a consequence of applying rule R7 to order all loads that read the value from BST before SWAP overwrites it.)

After careful debugging, the designers realized that the following sequence of events had lined up to create this problem. This particular processor had a write cache which acted as a buffer for writes.

1. The BST instruction on P1 caused an invalidate to be issued on the system bus for the cache line containing address A.

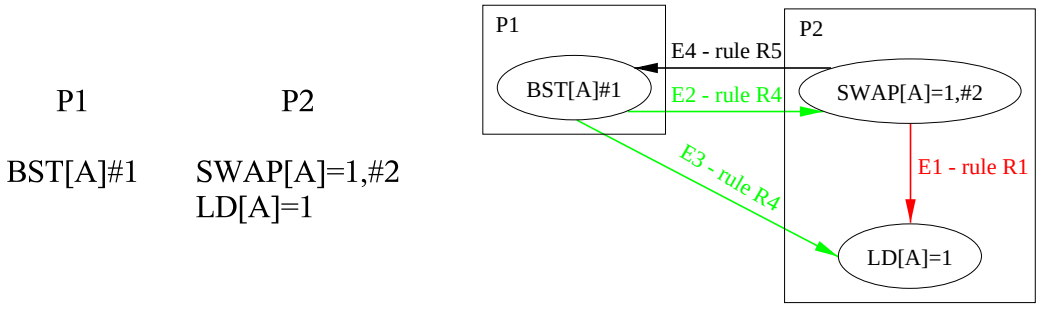
2. The SWAP on P2 was issued and found the corresponding line present in its write cache in the modified state, while the invalidate was still in flight.

3. The BST invalidate reached the L2 cache and write cache on P2. The SWAP had to re-read the updated data from main memory, and the line was installed in L2 cache with the BST data.

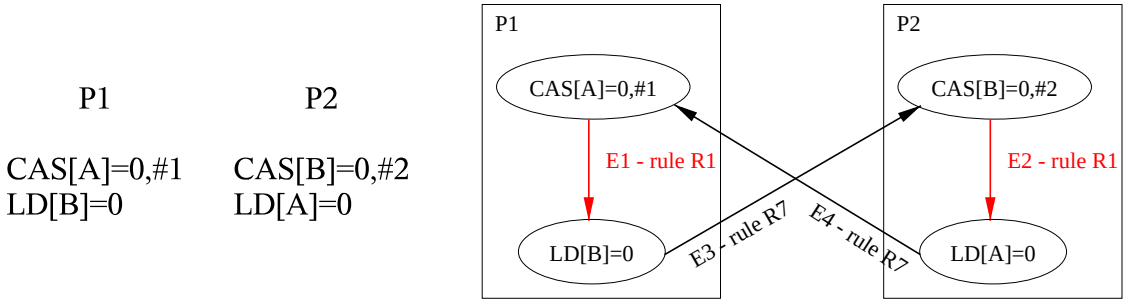
4. The store part of SWAP wrote the new data in the write cache but, because of an incorrect optimization that the processor performed when the tag is in a certain state at the time the SWAP is issued, did not modify the tag of that line in the write cache to dirty. This led to the updated data being lost when the line was later evicted from the write cache.

Notice that this type of problem may remain latent for a long time in a system design, leading only occasionally to mysterious data corruption or system crashes. Only proactive testing using test programs with aggressive data races can expose such problems during the verification or debug phase.

Figure 6.1(b) illustrates another real bug found on a different processor in the pre-silicon verification environment. CAS refers to the atomic compare and swap instruction. In this scenario, both CAS instructions completed the swap successfully. The analysis graph depicts a violation of the TSO model detected by:



(a) Bug 1



(b) Bug 2

Figure 6.1: Examples of UltraSPARC bugs found by TSOtool.
Initial value in locations A and B is 0.
BST[A]#1 is a 64-byte block store where the word at location A is written with the value 1.
SWAP[A]=1,#2 is a swap to location A reading the value 1 and writing the value 2.
CAS[A]=0,#1 is a successful compare and swap, which can be treated as a swap.
LD[A]=1 is a load from location A reading the value 1.

- Applying rule R1 to order loads (including CAS) according to the program order gives edge E1 - CAS[A] < LD[B] and E2 - CAS[B] < LD[A].
- Applying rule R7 to the store part of CAS[A], which is memory ordered after the synthetic root node writing the initial value 0 to A, implies that LD[A] reading the initial value is also memory ordered before CAS[A], resulting in edge E3 - LD[A] < CAS[A]. Similarly for edge E4 for location B.

These edges lead to a cycle in the analysis graph, signifying an atomicity violation in the CAS instruction. This bug turns out to be a problem created by a performance optimization in the microarchitecture which had been thought to be valid. The optimization caused the lock for the atomic swap to be released early, before the store part of the swap was complete. In some cases, this optimization was incorrect and opened a window for another store to sneak in and cause an atomicity violation.

Figure 6.2 shows that, without the *Atomicity enforcement*, detecting this bug is not obvious because there are no apparent cycles in the analysis graph. Furthermore, no other operations appear to intervene between the load and store parts of the atomic CAS and, hence, the Atomicity axiom appears to be preserved. Only when we also consider the Order axiom, which requires that a total operation order exists, would the problem manifest itself. With a total operation order, CAS.ST[A] and CAS.ST[B] must be ordered one way or the other. But either way, one CAS.ST would now intervene between the load and store parts of the other CAS; for example, if CAS.ST[A] < CAS.ST[B], this order with edge E4 and E6 shows that CAS.ST[A] does intervene between CAS.LD[B] and CAS.ST[B].

The followings are root-causes of some other failures detected by TSOtool:

- A prefetch cache dropped an invalidate request, and later returned stale data to the pipeline.
- Cacheable and non-cacheable stores went through different write queues; in some cases, the ordering between these queues was violated.
- The DRAM controller dropped a speculative load request due to a buffer full condition, leading to data corruption later.

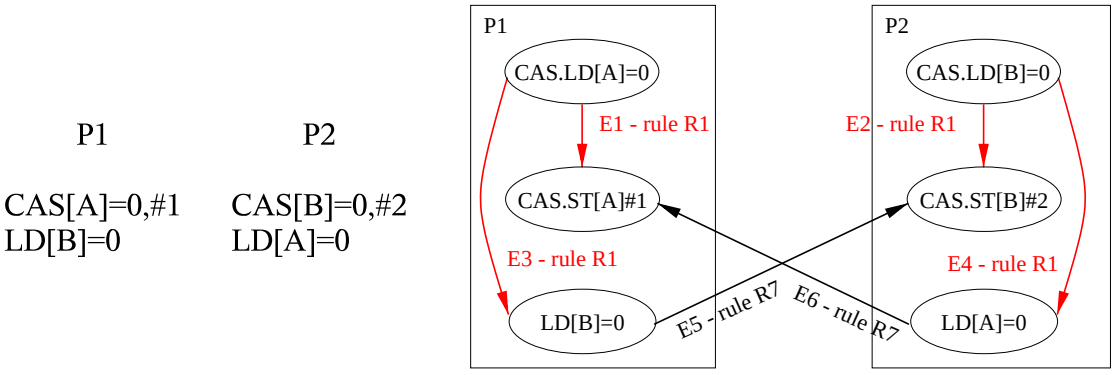


Figure 6.2: Importance of the *Atomicity enforcement* - without which, Bug 2 would not be obvious.
CAS.LD[A]=0 is the load part of the CAS reading the value 0.
CAS.ST[A]#1 is the store part of the CAS writing the value 1.

- Bus controller flow control logic caused a deadlock scenario in the system.

Besides such hardware bugs, TSOtool also found two bugs in the operating system. These bugs were related to the way the operating system emulated some of the memory instructions of the architecture. This demonstrates the immense power of an end-to-end checker for the complete memory system.

6.2 Testing TM Implementations

In this section, we report our experiences of employing the presented methodology on a transactional memory system, *Transactional memory Consistency and Coherence* (*TCC*) [29], which is briefly described in Section 5.4.5.

For our experiments, we had access to 2 models of TCC which we will refer to as *TCC-A* and *TCC-B*.

We performed *bug hunting* experiment with these 2 TCC models by varying a number of parameters of our test generation such as the number of processors, sizes of transactions, etc. We also varied the sizes and configuration of the simulated memory hierarchy. The goal was to create extremely stressful test-cases for the system and exercise corner cases in the design. Finally, the result of each test program was

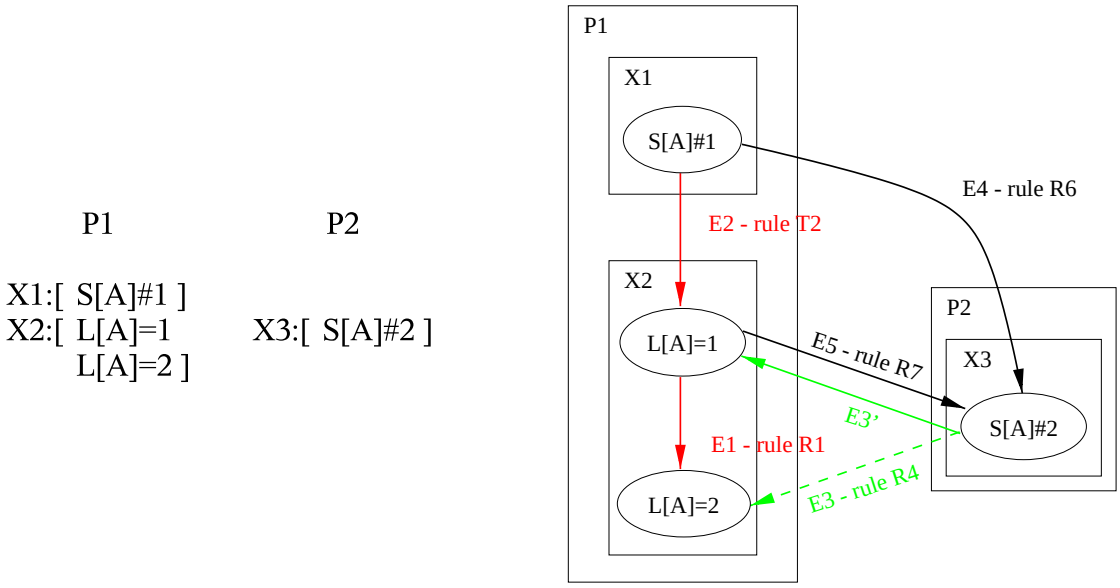
analyzed using the backtracking algorithm described in the previous section. We ran tests in this mode on both TCC-A and TCC-B models.

Despite the fact that we started our verification effort on the TCC-A model after it was stable and was already running test programs and benchmarks, our methodology helped uncover a previously unknown corner case bug in the software libraries. The effect of this bug was that 2 loads to the same memory location inside the same transaction returned different values in some cases, as demonstrated in Figure 6.3(a). This discrepancy of the load values means that the transaction was not atomic because it allowed another store to complete in between and modify the location. Detecting this as a cycle in the analysis graph requires the following steps:

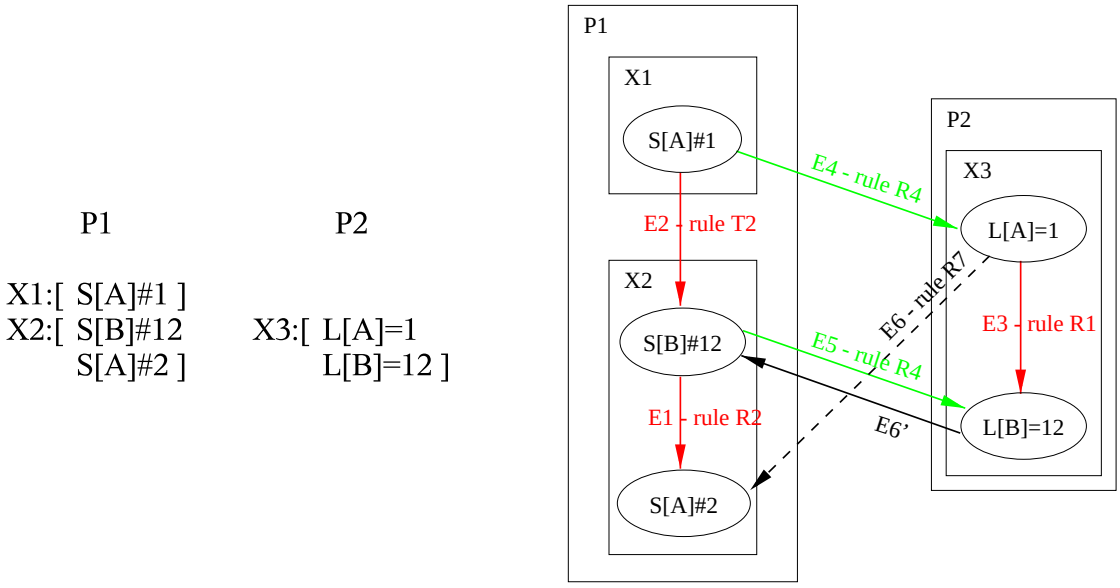
- Applying rule R1 and T2 creates edges E1 and E2 which maintain the program order on P1.
- Applying rule R4 gives edge E3 that orders S[A]#2 before L[A]=2, which turns into E3' due to the *TransAtomicity enforcement*.
- Given S[A]#1 < L[A]=2, edge E4 - S[A]#1 < S[A]#2 is inferred by rule R6.
- Finally, given S[A]#1 < S[A]#2, L[A]=1 must be ordered before S[A]#2 according to rule R7. This adds edge E5 which forms a cycle with E3'.

The root cause of this bug was that the compiler inappropriately optimized out the first load in the second transaction and, instead, directly took the value from the register that still carried the written value of the previous store, which belonged to the first transaction. This was because it could statically determine that both operations access the same address and it was not aware that the memory content could be altered. Had there been no store to that address performed by a committed transaction on another processor, it would be correct to perform the optimization. This bug was fixed by having TCC libraries correctly inform the compiler that memory content may be altered immediately after a transaction has committed.

This example is another case which illustrates that since our methodology performs end-to-end checking from the point of view of programmer-visible results, it



(a) Bug in TCC-A



(b) Bug in TCC-B

Figure 6.3: Examples of TCC bugs found by TSOtool. X1:[ ] denotes a transaction, X1.

can uncover not only problems in hardware design, but also problems in software components which participate in providing transaction guarantees.

For the TCC-B model, which was newly developed and less mature, our methodology detected illegal program results (cycles in the graph) in about 9% of test-cases run. One common manifestation was similar to that previously described where 2 loads to the same address disagree on the value except that the first load may or may not see the value carried over from some previous transaction (this is a different problem from the one uncovered in TCC-A). Another bug manifestation, depicted in Figure 6.3(b), involved 2 memory locations and at least 2 threads. It is detected by the following steps:

- Rule R4 gives S[B]#12 < L[B]=12, which in turn gives the order X2 < X3 due to the *TransAtomicity enforcement*.
- Given S[A]#1 and S[A]#2, we infer L[A]=1 < S[A]#2 by rule R7, and hence, X3 < X2, which conflicts with the order previously inferred.

This and other failure signatures are currently being investigated by the developers of the TCC-B model.

Note that without the presence of transaction semantics, all bug manifestations shown in this section would not be considered as memory inconsistencies.

We also studied at a high level how aggressively the TCC design was exercised while executing all the generated tests. For example, we measured the number of violations (which cause a transaction to be rolled back and retried) per transaction committed. In our experiments, this number was in the broad range of 0.33 to 33.84 (although the lower end is not a very small number, indicating that our tests may be biased too much toward sharing; increasing the number of shared addresses could help broaden the range). We also use the valid total operation order (TOO) obtained by the backtracking algorithm to approximate the degree of execution concurrency between test threads by measuring the extent of interleaving of operations from different threads relative to the total size of the program. Based on such a metric, we see a large range of achieved concurrency, almost covering both extremes (i.e., maximum

and minimum concurrency). During the course of applying our methodology to test a multiprocessor system, such measurable data can be used as feedback to further tune test parameters with the overall goal of increasing test coverage.

6.3 Summary

Our TSOtool methodology has proven effective in testing multiprocessor implementations; it has found hundreds of bugs to date, both during pre-silicon and post-silicon validation. Many of these bugs were subtle and uniquely found only by TSOtool.

Being an end-to-end check allows us to potentially uncover problems anywhere in the entire stack, from architecture to software. Moreover, by relying only on programmer-visible results, our approach is independent from the underlying microarchitecture of systems under test, making it relatively simple to be applied to a broad range of designs and platforms that implement the same memory consistency model. The methodology also applies to different memory consistency models as long as they can be formally specified using a similar axiomatic framework.

We note that an actual implementation may be designed such that it only allows a subset of the permissible outcomes according to its memory consistency model. Our methodology, however, is not a tight check for confirming that the implementation works as intended, i.e., we test the implementation against user's expectation rather than designer's. This is analogous to, for example, testing that a linear feedback shift register (LFSR) exceeds a certain period, for an application where the exact output sequence is not of a concern, without checking that it actually represents the feedback polynomial it intends to.

Chapter 7

Conclusions and Future Work

Traditional frequency scaling and exploiting instruction level parallelism (ILP) seem to have reached a point of diminishing return where complex and power-hungry microprocessor designs gain relatively little performance improvement. Although data level parallelism (DLP) can be more straightforward and efficient to exploit, it is highly application dependent and may not be present in typical programs. Therefore, the recent trend in improving microprocessor performance has been more and more toward exploiting thread level parallelism (TLP), energizing research and development interest in the multiprocessor arena. For example, chip multiprocessors (CMPs) are now commercially available after enjoying many years in research.

Shared-memory multiprocessors are complex systems especially because parallel computing is not at all natural to humans. To cope with the complexity, a memory consistency model is used to formally specify the behavior of a system at a high level with respect to memory operations from multiple processors. This memory model essentially serves as a contract between the system designers and the programmers, with strong implication on important aspects of the system such as performance and programmability, in particular. Given a certain memory model, an aggressive implementation may try to perform beyond what is allowed by the model, provided that programmers will not notice it doing so. For example, an implementation may speculatively execute memory operations and rewind on a detection of memory consistency violations.

With continuous increase in design complexity, verification clearly becomes an overwhelming challenge.

7.1 Thesis Summary

With our focus on functional verification in this thesis, we have presented our testing methodology that checks the behavior of multiprocessor implementations against an aspect of programmer's expectation, which is formally specified in terms of a memory consistency model. While it is based on pseudo-random testing widely used in the industry [6, 41, 46, 66], our emphasis is on testing scenarios where multiple processors actively share data, cases that were usually constrained, avoided, or not thoroughly checked due to the complexity involved in the correctness checking. This type of scenario, however, is very capable of exposing subtle problems in many designs and, thus, to allow for effective testing of these scenarios, our analysis algorithms which check the execution results against the memory consistency model were invented.

We have developed a tool, called TSOtool, based on the proposed methodology and algorithms and have successfully applied it to several multiprocessor designs both in the industry and the academia. TSOtool has been very effective; it has found hundreds of bugs in various designs so far, many of which were subtle and uniquely found only by this tool. Furthermore, being an end-to-end check has allowed us to potentially uncover problems anywhere in the entire stack, from architecture to software.

We have presented two key algorithms for our analysis, along with optimization techniques that significantly improve their performance. The first algorithm is sound but incomplete – it will not report false bugs but may miss real bugs. It has polynomial runtime and is fast in practice. The second algorithm is both sound and complete with exponential runtime in theory due to backtracking but has been shown to be reasonably efficient in practice, taking approximately twice the time required by the first algorithm. So far, we have not encountered real cases where the complete analysis would detect problems that were missed by the incomplete analysis.

During post-silicon validation, the efficiency of our analysis algorithms is an important factor that helps us achieve sufficiently high testing throughput. (On the other hand, analysis time is not a concern in pre-silicon validation due to simulation time being much longer.) In addition, the completeness of our backtracking algorithm greatly improves our confidence in the results.

7.2 Future Directions

There are several directions for future research which may be classified into 3 categories:

1. Further research on analysis algorithms.

2. Further research on improving methodologies for verification and testing.

3. Further research on related concepts.

We have only considered in this thesis the analysis algorithms for verifying memory consistency of test program executions when the mapping between each memory read operation and its responsible write operation (i.e., read-mapping) is known. In our methodology, this mapping corresponds to imposing a constraint that each write operation uses a unique value. Future research aiming at eliminating this constraint in the test programs will lead to a more general solution to the problem. Another algorithmic challenge lies in the backtracking when an execution result contains violations of a memory consistency model that were missed by the fast baseline algorithm, a case when an exhaustive search is unavoidable. We make an observation, however, that it is sufficient to only show that a subset of the execution result violates the memory consistency model. Future work may study how to efficiently identify such a subset. (The converse is not true for a valid execution result – it still has to be proven valid in its entirety, not piecewise, i.e., the problem is not *decomposable*.)

Rather than employing an off-line, post-mortem analysis like our algorithms, one may develop an on-line analysis which can detect memory inconsistencies on their

outset, potentially making debugging easier or even allowing for recovery. This on-line analysis may require additional hardware support and can also be made into a hardware checker itself for a performance advantage. Examples of research in this direction are the work by Cain and Lipasti [7] and by Meixner and Sorin [47].

During post-silicon validation, one of the biggest challenges in debugging problems in multiprocessor systems is, in fact, to reliably reproduce the problems at the first place. Reproducibility is required because observability is usually so limited that there is not enough information to understand and debug a problem on its first encounter and, therefore, more information needs to be collected from multiple occurrences of the same problem. Unfortunately, due to possible non-determinism and lack of controllability, it is very unlikely that re-executing the same program would result in the same interleaving of operations as well as the same hardware events (state of control logics, for instance). There are a number of research projects which aim at recording information that is sufficient for replaying a program (e.g. [55, 71]). These techniques, however, have been used primarily for debugging software problems such as data races. While they may be effective in reproducing the operations' interleaving, their effectiveness in reproducing the hardware events which actually led to the memory inconsistencies has yet to be studied. We also note that it is sometimes not important to reproduce a problem using the same test program as long as we know how to make the problem manifest itself with short time-to-failure (TTF).

Test coverage has been one of the key metrics used for tracking progress and quality of simulation-based verification during pre-silicon validation. Appropriate usage of coverage information as feedback to guide test generation may help designers find more bugs with less simulation time [18, 68, 69].

Dynamically detecting data races (e.g. [3, 17, 53, 56, 60]), serializability violations (e.g. [19, 72]), and memory consistency violations (e.g. [22, 25, 44, 47]) are very well related problems. Future research may try to leverage ideas and techniques for solving one problem to solve another.

Finally, with the never-stopping growth of microprocessor design complexity, we can expect to welcome a continuous stream of novel research in verification.

Appendix A

Equivalence of Definitions of the Atomicity Axiom

We presented in Section 2.2.3 the following three versions of the Atomicity axiom for the Sequential Consistency memory model:

Atomicity: No memory operations can intervene between the load and store parts of a swap.

$$[L_a^i; S_a^i] \Rightarrow (L_a^i < S_a^i) \wedge (Op_b^j < L_a^i \vee S_a^i < Op_b^j), \forall Op_b^j \quad ; Op_b^j \neq L_a^i \wedge Op_b^j \neq S_a^i$$

Atomicity-A: *Stores* cannot intervene between the load and store parts of a swap.

$$[L_a^i; S_a^i] \Rightarrow (L_a^i < S_a^i) \wedge (S_b^j < L_a^i \ \vee \ S_a^i < S_b^j), \forall S_b^j \qquad ; S_b^j \neq S_a^i$$

Atomicity-B: Operations from *any processor to the same location* or operations from *the same processor to any location* cannot intervene between the load and store parts of a swap.

$$\begin{aligned} [L_a^i; S_a^i] \Rightarrow\ & (L_a^i < S_a^i) \wedge \\ & (Op_a^j < L_a^i \ \vee \ S_a^i < Op_a^j), \forall Op_a^j \wedge && ; Op_a^j \neq L_a^i \wedge Op_a^j \neq S_a^i \\ & (Op_b^i < L_a^i \ \vee \ S_a^i < Op_b^i), \forall Op_b^i && ; Op_b^i \neq L_a^i \wedge Op_b^i \neq S_a^i \end{aligned}$$

The equivalence of these three versions can be proven by showing that an SC-compliant total operation order (TOO) satisfying a weaker version of the axiom, **Atomicity-A** (or **Atomicity-B**), can be transformed into another SC-compliant

TOO satisfying the strictest version of the axiom, **Atomicity**. The other direction of the transformation is trivially true.

Proof outline: As an example, we show one such transformation for **Atomicity-A**. Given an SC-compliant TOO which satisfies **Atomicity-A**, we note that the intervening operations, if any, between the load and the store part of a swap can only be loads and they must not belong to the same processor executing this swap because that would violate the program order requirement of the SC model (the OpOp axiom). Furthermore, they cannot be the load part of other swap operations because that would essentially lead to having the store part of one swap intervene between the load and store part of another swap, which contradicts **Atomicity-A**. Therefore, we can construct another SC-compliant TOO which satisfies **Atomicity** simply by moving the load part of each swap operation to immediately precede the store part. Readers can verify that other axioms of the SC model hold true with this transformation. □

The equivalence of these versions can also be similarly proven for some other memory models such as the TSO model.

Bibliography

[1] Sarita V. Adve. *Designing memory consistency models for shared-memory multiprocessors*. PhD thesis, University of Wisconsin at Madison, 1993.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. Technical Report WRL-TR 95/7, Digital Western Research Laboratory, September 1995.

[3] Sarita V. Adve, Mark D. Hill, Barton P. Miller, and Robert H. B. Netzer. Detecting data races on weak memory systems. In *ISCA'91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 234–243. May 1991.

[4] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA'05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327. Feb 2005.

[5] Roberto Baldoni and Michel Raynal. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2), 2002.

[6] Bob Bentley and Rand Gray. Validating the Intel Pentium 4 processor. *Intel Technology Journal*, (Q1):8, February 2001.

[7] Harold W. Cain and Mikko H. Lipasti. Verifying sequential consistency using vector clocks. In *SPAA'02: Proceedings of the 14th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 153–154, 2002.

[8] Harold W. Cain, Mikko H. Lipasti, and Ravi Nair. Constraint graph analysis of multithreaded programs. In *PACT'03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, pages 4–14, Washington, DC, USA, 2003. IEEE Computer Society.

[9] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. The complexity of verifying memory coherence. In *SPAA'03: Proceedings of the 15th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 254–255, 2003.

[10] Kaiyu Chen and Sharad Malik. Runtime validation of multithreaded processors. Technical report, Dept. of Electrical Engineering, Princeton University, May 2005.

[11] William W. Collier. Multiprocessor diagnostics. http://www.mpdiag.com/archtest.html.

[12] William W. Collier. *Reasoning about parallel architectures*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[13] Anne E. Condon, Mark D. Hill, Manoj Plakal, and Daniel J. Sorin. Using lamport clocks to reason about relaxed memory models. In *HPCA'99: Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, 1999.

[14] Anne E. Condon and Alan J. Hu. Automatable verification of sequential consistency. In *SPAA'01: Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 113–121, 2001.

[15] Francisco Corella, Janice M. Stone, and Charles M. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report 18638(81566), IBM Research Division, T.J. Watson Research Center, January 1993.

[16] Michel Dubois, Christoph Scheurich, and Fayé Briggs. Memory access buffering in multiprocessors. In *ISCA'86: Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, 1986.

[17] Perry A. Emrath, Sanjoy Ghosh, and David A. Padua. Detecting nondeterminacy in parallel programs. *IEEE Software*, 9(1):69–77, 1992.

[18] Shai Fine and Avi Ziv. Coverage directed test generation for functional verification using Bayesian networks. In *DAC'03: Proceedings of the 40th Design Automation Conference*, pages 286–291, 2003.

[19] Cormac Flanagan and Stephen N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *POPL'04: Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 256–267, 2004.

[20] Michael R. Garey and David S. Johnson. *Computer and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.

[21] Kourosh Gharachorloo. *Memory consistency models for shared-memory multiprocessors.* PhD thesis, Stanford University, December 1995.

[22] Kourosh Gharachorloo and Phillip B. Gibbons. Detecting violations of sequential consistency. In *SPAA'91: Proceedings of the 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 316–326, 1991.

[23] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip B. Gibbons, Anoop Gupta, and John L. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA'90: Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, 1990.

[24] Phillip B. Gibbons and Ephraim Korach. The complexity of sequential consistency. In *SPDP'92: Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*, pages 317–325, 1992.

[25] Phillip B. Gibbons and Ephraim Korach. On testing cache-coherent shared memories. In *SPAA'94: Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 177–188, 1994.

[26] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4):1208–1244, 1997.

[27] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *ISCA'99: Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 162–171, Washington, DC, USA, 1999. IEEE Computer Society.

[28] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[29] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA'04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102. IEEE Computer Society, Jun 2004.

[30] Sudheendra Hangal, Durgam Vahia, Chaiyasit Manovit, Juin-Yeu Joseph Lu, and Sridhar Narayanan. TSOtool: A program for verifying memory systems using the memory consistency model. In *ISCA'04: Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 114, Washington, DC, USA, 2004. IEEE Computer Society.

[31] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA'93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. May 1993.

[32] Lisa Higham and Lill Anne Jackson. Porting between Itanium and Sparc multiprocessing systems. In *SPAA'06 (to appear): Proceedings of the 18th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, 2006.

[33] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Defining and comparing memory consistency models. In *PDCS'97: Proceedings of the International Conference on Parallel and Distributed Computing Systems*, pages 349–356, 1997.

[34] Mark D. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8):28–34, Aug 1998.

[35] Ravi Hosabettu. SPARC V9 atomic transaction. Sun Microsystems, February 2003.

[36] Sunil Kakkar. Advanced processor architectures and verification challenges. HPCA'04: Tutorial at the 12th International Symposium on High-Performance Computer Architecture, February 2004.

[37] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[38] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–693, September 1979.

[39] Anders Landin, Erik Hagersten, and Seif Haridi. Race-free interconnection networks and multiprocessor consistency. In *ISCA'91: Proceedings of the 18th Annual International Symposium on Computer Architecture*, pages 106–115, 1991.

[40] Daniel H. Linder and James C. Harden. Access graphs: A model for investigating memory consistency. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):39–52, 1994.

[41] John M. Ludden, Wolfgang Roesner, Gerry M. Heiling, John R. Reysa, Jonathan R. Jackson, Bing-Lun Chu, Michael L. Behm, Jason Baumgartner, Richard D. Peterson, Jamee Abdulhafiz, William E. Bucy, John H. Klaus, Danny J. Klema, Tien N. Le, F. Danette Lewis, Philip E. Milling, Lawrence A. McConville, Bradley S. Nelson, Viresh Paruthi, Travis W. Pouarz, Audre D. Romonosky, Jeff Stuecheli, Kent D. Thompson, Dave W. Victor, and Bruce Wile. Functional verification of the POWER4 microprocessor and POWER4 multiprocessor system. *IBM Journal of Research and Development*, 46(1):53–76, 2002.

[42] Steven T. Mangelsdorf, Raymond P. Gratias, Richard M. Blumberg, and Rohit Bhatia. Functional verification of the HP PA 8000 processor. *Hewlett-Packard Journal*, August 1997.

[43] Chaiyasit Manovit and Sudheendra Hangal. Efficient algorithms for verifying memory consistency. In *SPAA'05: Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 245–252, 2005.

[44] Chaiyasit Manovit and Sudheendra Hangal. Completely verifying memory consistency of test program executions. In *HPCA'06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 2006.

[45] Chaiyasit Manovit, Sudheendra Hangal, Hassan Chafi, Austen McDonald, Christos Kozyrakis, and Kunle Olukotun. Testing implementations of transactional memory. In *PACT'06 (to appear): Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006.

[46] Shrenik Mehta, Sultan Ahmed, S. Al-Ashari, Dennis Chen, C. Dev, S. Cokmez, P. Desai, R. Eltejaein, P. Fu, J. Gee, Thomas Granvold, A. Iyer, K. Lin, G. Maturana, D. McConn, H. Mohammed, Jamshid Mostoufi, A. Moudgal, S. Nori, N. Parveen, G. Peterson, Michael Splain, and T. Yu. Verification of the UltraSPARC microprocessor. In *COMPCON'95: Proceedings of the 40th IEEE Computer Society International Conference*, page 452, Washington, DC, USA, 1995. IEEE Computer Society.

[47] Albert Meixner and Daniel J. Sorin. Dynamic verification of sequential consistency. In *ISCA'05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 482–493, June 2005.

[48] Mark Moir. Hybrid transactional memory, Jul 2005. Unpublished manuscript.

[49] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. LogTM: Log-based transactional memory. In *HPCA'06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 2006.

[50] Ratan Nalumasu, Rajnish Ghughal, Abdelillah Mokkedem, and Ganesh Gopalakrishnan. The "test model-checking" approach to the verification of formal memory models of multiprocessors. In *CAV'98: Proceedings of the 10th International Conference on Computer Aided Verification*, pages 464–476, London, UK, 1998. Springer-Verlag.

[51] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[52] Seungjoon Park and David L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Transactions on Computers*, 48(2):227–235, 1999.

[53] Dejan Perkovic and Peter J. Keleher. Online data-race detection via coherency guarantees. In *OSDI'96: Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation*, pages 47–57, 1996.

[54] Manoj Plakal, Daniel J. Sorin, Anne E. Condon, and Mark D. Hill. Lamport clocks: Verifying a directory cache-coherence protocol. In *SPAA'98: Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 67–76, 1998.

[55] Milos Prvulovic. CORD: Cost-effective (and nearly overhead-free) order-recording and data race detection. In *HPCA'06: Proceedings of the 12th International Symposium on High-Performance Computer Architecture*, 2006.

[56] Milos Prvulovic and Josep Torrellas. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *ISCA'03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 110–121, 2003.

[57] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. Technical report, Compaq Systems Research Center, December 2001.

[58] Ravi Rajwar, Maurice Herlihy, and Konrad Lai. Virtualizing transactional memory. In *ISCA'05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*.

[59] Smruti Sarangi. Phoenix: Detecting and recovering from permanent processor hardware bugs. 2005.

[60] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *SOSP'97: Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, 1997.

[61] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC'95: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.

[62] Arrvindh Shriraman, Virendra Marathe, Sandhya Dwarkadas, Michael L. Scott, David Eisenstat, Christopher Heriot, William N. Scherer, and Michael F. Spear. Hardware acceleration of software transactional memory. Technical Report UR CSD;TR 887, Dept. of Computer Science, University of Rochester, December 2005.

[63] Pradeep S. Sindhu, Jean-Marc Frailong, and Michel Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox Palo Alto Research Center, December 1991.

[64] Richard L. Sites, editor. *Alpha architecture reference manual*. Digital Press, Newton, MA, USA, 1992.

[65] SPARC International Inc. *The SPARC architecture manual: version 8*. 1992.

[66] Scott A. Taylor, Michael Quinn, Darren Brown, Nathan Dohm, Scot Hildebrandt, James Huggins, and Carl Ramey. Functional verification of a multiple-issue, out-of-order, superscalar Alpha processor – the DEC Alpha 21264 microprocessor. In *DAC'98: Proceedings of the 35th Design Automation Conference*, pages 638–643, 1998.

[67] Scott A. Taylor, Carl Ramey, Craig Barner, and David Asher. A simulation-based method for the verification of shared memory in multiprocessor systems. In *ICCAD'01: Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pages 10–17, Piscataway, NJ, USA, 2001. IEEE Press.

[68] Shmuel Ur and Yaov Yadin. Micro architecture coverage directed generation of test programs. In *DAC'99: Proceedings of the 36th Design Automation Conference*, pages 175–180, 1999.

[69] Ilya Wagner, Valeria Bertacco, and Todd Austin. StressTest: An automatic approach to test generation via activity monitors. In *DAC'05: Proceedings of the 42nd Design Automation Conference*, pages 783–788, 2005.

[70] David L. Weaver and Tom Germond, editors. *The SPARC architecture manual: version 9*. 1994.

[71] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *ISCA'03: Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–135, 2003.

[72] Min Xu, Rastislav Bodík, and Mark D. Hill. A serializability violation detector for shared-memory server programs. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–14, 2005.