University of Pennsylvania
ScholarlyCommons

Technical Reports (CIS) | Department of Computer & Information Science

1-1-2010

Generating Litmus Tests for Contrasting Memory Consistency Models - Extended Version

Sela Mador-Haim
University of Pennsylvania, selama@seas.upenn.edu

Rajeev Alur
University of Pennsylvania, alur@cis.upenn.edu

Milo M.K. Martin
University of Pennsylvania, milom@cis.upenn.edu

Follow this and additional works at: http://repository.upenn.edu/cis_reports

Part of the Computer Sciences Commons

Recommended Citation

Mador-Haim, Sela; Alur, Rajeev; and Martin, Milo M.K., "Generating Litmus Tests for Contrasting Memory Consistency Models - Extended Version" (2010). *Technical Reports (CIS)*. Paper 934.
http://repository.upenn.edu/cis_reports/934

University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-10-15.

This paper is posted at ScholarlyCommons. http://repository.upenn.edu/cis_reports/934
For more information, please contact repository@pobox.upenn.edu.

Generating Litmus Tests for Contrasting Memory Consistency Models - Extended Version

Abstract
Well-defined memory consistency models are necessary for writing correct parallel software. Developing and understanding formal specifications of hardware memory models is a challenge due to the subtle differences in allowed reorderings and different specification styles. To facilitate exploration of memory model specifications, we have developed a technique for systematically comparing hardware memory models specified using both operational and axiomatic styles. Given two specifications, our approach generates all possible multi-threaded programs up to a specified bound, and for each such program, checks if one of the models can lead to an observable behavior not possible in the other model. When the models differs, the tool finds a minimal "litmus test" program that demonstrates the difference. A number of optimizations reduce the number of programs that need to be examined. Our prototype implementation has successfully compared both axiomatic and operational specifications of six different hardware memory models. We describe two case studies: (1) development of a non-store atomic variant of an existing memory model, which illustrates the use of the tool while developing a new memory model, and (2) identification of a subtle specification mistake in a recently published axiomatic specification of TSO.

Disciplines
Computer Sciences

Comments
University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-10-15.

This technical report is available at ScholarlyCommons: http://repository.upenn.edu/cis_reports/934

Generating Litmus Tests for Contrasting Memory Consistency Models* - Extended Version**

Sela Mador-Haim, Rajeev Alur, and Milo M.K. Martin

University of Pennsylvania

Abstract. Well-defined memory consistency models are necessary for writing correct parallel software. Developing and understanding formal specifications of hardware memory models is a challenge due to the subtle differences in allowed reorderings and different specification styles. To facilitate exploration of memory model specifications, we have developed a technique for systematically comparing hardware memory models specified using both operational and axiomatic styles. Given two specifications, our approach generates all possible multi-threaded programs up to a specified bound, and for each such program, checks if one of the models can lead to an observable behavior not possible in the other model. When the models differs, the tool finds a minimal "litmus test" program that demonstrates the difference. A number of optimizations reduce the number of programs that need to be examined. Our prototype implementation has successfully compared both axiomatic and operational specifications of six different hardware memory models. We describe two case studies: (1) development of a non-store atomic variant of an existing memory model, which illustrates the use of the tool while developing a new memory model, and (2) identification of a subtle specification mistake in a recently published axiomatic specification of TSO.

1 Introduction

Well-defined memory consistency models are necessary for writing correct and efficient shared memory programs [2]. The emergence of mainstream multi-core processors as well as recent developments in language-level memory models [4, 19], have stirred new interest in hardware-level memory models. The formal specification of memory models is challenging due to the many subtle differences between them. Examples of such differences include different allowed reorderings, store atomicity, types of memory fences, load forwarding, control and data

* The authors acknowledge the support of NSF grants CCF-0905464 and CCF-0644197, and of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

** This is an extended version of the CAV2010 paper, including detailed section on operational models and added implementation details

dependencies, and different specification styles (operational and axiomatic). Architecture manuals include litmus tests that can be used to differentiate between memory models [17, 23], but these litmus tests are not complete, and coming up with new litmus tests requires identifying the subtle difference between memory models this test is meant to detect.

Our goal is to aid the process of developing specifications for hardware-level memory models by providing a technique for systematically comparing memory model specifications. When there is a difference between the two memory models, the technique generates a litmus test as a counter-example, including both a program and an outcome allowed only in one of the models. Such a technique can be used in several different scenarios. One case is comparing two presumably equivalent models, for example comparing an axiomatic specification given as a set of first order logic formulas to an operational specification that describes the model as a state transition system. Alternatively, we may also want to check whether one model is strictly weaker (or stronger) than the other.

Our approach is based on systematic generation of all possible programs up to a specified size bound. For each program, we check if one of the models can lead to an observable behavior that is not possible in the other model. To produce the set of observable behaviors for a program under a given memory model, we use two different search techniques depending on whether the model specification is operational or axiomatic. When there is an observable behavior in one memory model that is not allowed by the other model, the approach outputs the program and the contrasting behavior. Because we explore starting with the smallest programs, this is a minimal litmus test.

We employ several techniques to make this approach practical. A naive enumeration of all test programs up to the specified bound produces too many programs, so we employ optimizations to reduce the number of programs that need to be examined. We use symmetry reductions based on value, address and thread symmetries. Furthermore, we identify and skip redundant programs that will not expose any new differences by analyzing the conflict graph of the program. We use partial order reduction techniques to optimize exploration of operational models and an incremental SAT approach for axiomatic models.

We tested this approach by comparing the axiomatic and operational specifications of six different memory models: Sequential Consistency (SC), SPARC's TSO, PSO and RMO [23] and non-store-atomic relaxations of TSO and PSO. Our technique finds the known differences, but it also uncovered some errors in two of our specifications, which we corrected. Finding differences takes less than a second in most cases and only several minutes in the worst cases we encountered. We tested the scalability of this technique and found that we can explore all programs up to six read and write operations plus any number of fences in a few minutes. Our results indicate these bounds are adequate to detect subtle differences.

We performed two case studies. We developed a specification of a non-store-atomic variant of PSO, which illustrates that the tool quickly identifies subtle specification mistakes. In another case study, we contrasted SOBER's axiomatic

In all tests, initially X=Y=0

Test A1. Write→Read

T1	T2
Y = 1; r1 = X;	X = 1; r2 = Y;

Outcome: r1=r2=0

Test A2. Write→Write

T1	T2

X = 1;
Y = 1;
	r1 = Y;
	fence;
	r2 = X;

Outcome: r1=1; r2=0

Test A3. Read→Read

T1	T2
X = 1; fence; X = 2;	r1 = X; r2 = X;

Outcome: r1=2; r2=1

Test A4. Read→Write

T1	T2
r1 = X; Y = 1;	r2 = Y; X = 1;

Outcome: r1=r2=1

Test A5. Store atomicity test

T1	T2	T3	T4

X = 1;	Y = 1;	r1=X;
		fence;
		r2=Y;
			r3=Y;
			fence;
			r4=X;

Outcome: r1=r3=1; r2=r4=0

Test A6. Coherence test

T1	T2	T3	T4

X = 1;	X = 2;	r1=X;
		fence;
		r2=X;
			r3=X;
			fence;
			r4=X;

Outcome: r1=r4=1; r2=r3=2

Test A7. Independent write reorders

T1	T2	T3

X = 1;
Y = 2
	Y = 1;
	X = 2;
		r1=X;
		fence;
		r2=X;
		r3=Y;
		fence;
		r4=Y;

Outcome: r1=r3=2; r2=r4=1

Fig. 1: Litmus tests for local and global reordering of memory accesses

specification of TSO [6] with an operational specification of TSO and showed our technique detects a recently discovered specification error [8].

2 Axiomatic specification for memory models

A *memory consistency model* is a specification of the shared memory semantics of a parallel system [2]. The simplest memory model is *Sequential Consistency* (SC) [18]. An execution of a concurrent program is sequentially consistent if all reads and writes appear to have occurred in a sequential order that is in agreement with the individual program orders of each thread. In order to improve system performance and allow common hardware optimization techniques such as store buffers, many systems implement *weaker memory models* such as SPARC's TSO, PSO and RMO [23], Intel's x86 [17], Intel's Itanium [25], ARM and PowerPC [3].

Consider for example the program in Test A1 (Fig. 1). Executing under SC, at least one of the writes must occur before any of the reads, and therefore the outcome $r1 = 0; r2 = 0$ is not allowed. A processor that has a store buffer, on the other hand, can defer the writes to the main memory and effectively reorder the writes after the reads, and thus reading zero for both registers is allowed. SPARC's TSO and x86 both allow this relaxation. Other memory models allow further relaxations such as write after write and read after read (RMO, Itanium, PowerPC). Some memory models such as SC are *store atomic*, in the sense that

all threads observe writes in the same order, but other memory models are non-store-atomic and allow different threads to observe writes from other threads in a different order (such as PowerPC), as illustrated by Test A5 in Fig. 1.

The purpose of a memory model specification is to express precise constraints on which values can be associated with reads in a given multi-threaded program. One method of specifying a memory model is axiomatic style, given by a set of axioms that define which *execution traces* are allowed by the model and in particular which writes can be observed by each read. An execution trace is a sequence of memory operations (Read, Write, Fence) produced by a program. Each operation in the trace includes an identifier of the thread that produced this operation, and the address and value of the operation for reads and writes.

Axiomatic specifications usually refer to the program order, $<_p$. For two operations x and y, $x <_p y$ if both x and y belong to the same thread and x precedes y in the execution trace. The program order, however, is not necessarily the order in which memory operations are observed by the main memory. The memory order, $<_m$, is a total order that indicates the order in which memory operations affect the main memory. A read observes the latest write to the same address according to $<_m$.

We define store atomic memory models using two types of axioms: a *read-values* axiom and an *ordering* axiom. The *read-values* axiom states that each read observes the latest write to the same location according to the memory order. To support load forwarding, reads may observe local writes that precede them in program order, even if such write is ordered after the read in the memory order. We handle this forwarding in same style as in Burckhardt et al [5], by defining a function $sees(x, y, <)$, which is true if y is a write and $y < x$ or $y <_p x$. The *read-values* axiom for store-atomic memory models is:

Read values Given a read x and a write y to the same address as x, then x and y have the same value if $sees(x, y, <_m)$ and there is no other write z such that $sees(x, z, <_m)$ and $y <_m z$. If for a read x there is no write y such that $sees(x, y, <_m)$ then the read value is 0.

All our store atomic memory model specifications use the same read-values axiom, but differ in the definition of the *ordering* axiom, specifying which memory orders are allowed by the model. For example, TSO allows reordering only writes after later reads, and therefore the TSO reordering axiom is:

TSO-reordering For every x and y, $x <_p y$ implies that $x <_m y$, unless x is a write and y is a read.

The ordering axiom for PSO relaxes TSO by allowing reordering writes with other writes to a different location, as shown in Test A2. The ordering axiom for PSO is:

PSO-reordering For every x and y, if $x <_p y$ then $x <_m y$ in the following cases: 1. x is a read. 2. Either x or y is a fence. 3. Both x and y are writes and they both have the same address.

The ordering axiom for RMO further relaxes PSO by allowing reordering reads with other reads (as in Test A3) and reads with later writes (as in test A4) as long as there is no data or control dependency between them. The ordering axiom for RMO is:

RMO-reordering For every x and y, if $x <_p y$ then $x <_m y$ in the following cases: 1. There is a dependency between x and y 2. Either x or y is a fence. 3. Both x and y access the same address and y is not a read.

In non-store-atomic models, threads may observe stores in different orders, so we can no longer use one global memory order. Instead, we define an order $<_t$ for each thread t, which we call the *view* of thread t. Each view includes all operations and not only writes, so that we could, for example, restrict the position of a write in a thread relative to reads by another thread, which, as we see later, is helpful to ensure transitive causal order between operations.

As in the store-atomic case, loads see the latest stores to the same address except in the case of forwarding, but the relevant order for loads in thread t is view order $<_t$. We modify the read-values and ordering axioms to observe the latest write in the relevant view:

Non-store-atomic read-values Given a read x in thread t and a write y to the same address as x, then x and y have the same value if y is the most recent write according to $sees(x, y, <_t)$. If for a read x in thread t there is no write y such that $sees(x, y, <_t)$, the read value is 0.

To define NPSO, the non-store atomic version of PSO maintains the same order restrictions between operation from the same thread as in the case of PSO:

NPSO ordering For every x and y, if $x <_p y$ then for every t $x <_t y$ must hold in the following cases: 1. x is a read. 2. Either x or y is a fence. 3. Both x and y are writes and they both have the same address.

The non-store-atomic case requires adding another axiom for coherence, stating that there is a total order between writes to the same address:

NPSO coherence For every two write operations x and y that write to the same address, and for every two threads, t and t', if $x <_t y$ then $x <'_t y$.

The above axioms represent our first attempt at specifying a model which is a non-store atomic relaxation of PSO in an axiomatic style, but, as we describe in Section 5, this specification is too weak. In Section 5.3, we use our technique to develop the missing axioms for NPSO.

3 Operational specification framework

Another method of specifying a memory model is using an operational style, which abstracts actual hardware structures such as a store buffer. This section describes operational specifications for several memory models that we defined as a part of this work.

3.1 The Component Model

We introduce a component based framework that enables us to specify different memory models based on a small number of different components. The components can be connected in different ways to specify the operational semantics of different memory models.

Our specification of the operational semantics of weak memory models is based on a set of components that run as concurrent processes. The components can communicate with each other by sending and receiving messages. All communication between components are done via rendezvous message passing. We define the following five components:

- *Processor*: a component that represents a single thread. Its main function is to send read, write and fence operations. It also tracks dependencies
- *SimpleMem*: represents a single memory location
- *WriteQueue*: used to reorder writes by delaying them
- *SetRead*: access past values by keeping a history set
- *FutureRead*: access future values using a prophecy variable

Each of these components is a process that runs concurrently with other components. Components can initiate actions, send messages to other components, and may react to messages from other components. Next, we define the glue logic that binds these model together and determines which messages go to which components.

The components are presented here using Promela [16], the specification language of the Spin model checker. Each component is defined as a Promela process (The Promela code for all components is found in the appendix). Each component have one input channel and one output channel. Both are rendezvous channels (channels with a queue of size 0). Messages record types, corresponding to actions as defined in section 1, with the following fields:

- $m.type$ is the message type, which is one of $\{Write, Read, Fence, Reply, Complete, Forward\}$.
- $m.addr$ is the memory location
- $m.val$ is a read/write value
- $m.proc$ is the thread/processor identifier
- $m.ins$ identifies the instruction in the program that generated the message

The glue logic consists of redirection statements that transfer messages from one component to another component, based on the value of the message. For example, the code connecting processors with memory elements in Promela, sending each message `m` to the relevant memory location according to `m.addr` is:

```
:: pout[pr]?[m] -> pout[pr]?m; min[m.addr]!m;
```

3.2 Sequential Consistency

SC (Sequential Consistency [18]) is the simplest memory consistency model. In this model, all memory operations are observed in program order by all threads. The specification for SC consists of two components: Processor and SimpleMem.

The Processor component's main job is to send messages corresponding to the operations of a single threads. It contains a sequence of operators, and every time it is activated it sends a message corresponding to the next operation in the sequence (read request, write or fence). It also keeps track of read values that are returned from the main memory. It receives two types of messages. First, messages of type Reply, which are sent back from the memory as a response to read request and contain a value. The processor keeps track of this value by storing it in a queue of returned values. This queue of returned values can be used to compare read values between executions. The second type of incoming message is Complete, announcing that a read request is completed. Usually, a Reply message is followed immediately by a Complete message. However, as we explained in Section 3.6, there are cases where the component guesses a future value, and the read is complete only when the guess is confirmed with an actual write. In this case, the processor keeps track of incomplete requests, and blocks writes until all reads to the same address are complete.

The second component, *SimpleMem*, represents the main memory. This is a simple component containing one variable that keeps track of the current value of a single memory location. It does not initiate any actions on its own. It receives and responds to two types of incoming messages: in case of a Read request, it sends back a Reply message with its current value, immediately followed by a Complete message. In case of an incoming Write message, it updates its internal value and send a Forward message to other components with the same information as in the received Write message. The forwarding feature is not used for SC, but we use it later.

We model SC for n processors and m memory locations using components $Pr[0], ...Pr[n-1]$, which are instances of Processor ($Pr[i]$ is Processor instantiated with a processor ID i) and components $M[0], ...M[m-1]$, which are instances of SimpleMem. The binding logic is shown in Fig. 2. The solid lines represent the flow of request messages, which are Read, Write, and Fence. The dashed lines represent response messages, which are Reply and Complete. As seen in the diagram, Processors in this specification for SC communicate directly with the main memory, and this way all memory operations are seen in program order.

3.3 Total Store Order

In TSO (Total Store Order [23]), writes can be reordered after newer reads, but writes are observed in their program order as well as any other pair of operations (it relaxes Test A1, but no others). Writes can be effectively reordered after reads by postponing the writes, and sending them to main memory at a later time. Using a FIFO buffer, delayed writes are guaranteed to be sent to the main memory in order.

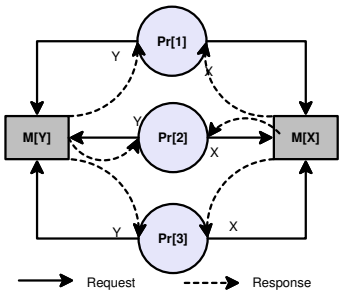

Fig. 2: Component diagram of SC

The *WriteQueue* component implements such a queue that stores incoming writes and delay the time they are sent to memory. When WriteQueue receives an incoming write message, it stores this message in the queue. An incoming fence message drains the queue, and thus prevents any local write to be delayed after the fence. In case of an incoming read message, WriteQueue looks for writes that match the address of the incoming read, and send a reply message with the value of the latest matching write. This is done in order to support forwarding of local write.

In addition, WriteQueue may perform a dequeue and send outgoing write messages. Because WriteQueue runs as a concurrent process, it can be interleaved in any possible way with the other processes, and as a result sending writes from the queue is performed non-deterministically.

Using WriteQueue combined with the two previous components, we can specify TSO: For n processors and m memory locations, the components are $Pr[0], ...Pr[n-1]$ Processor instances, $M[0], ...M[m-1]$ SimpleMem instances, and $WQ[0], ...WQ[n-1]$ WriteQueue instances. The connections between these component for TSO is shown in Fig. 3 (left).

All write requests by the same processor to all memory locations go through the same WriteQueue component. Due to the FIFO queue, the order between writes from the same processor cannot change. In case of a read request, WriteQueue either responds with the latest value in the queue or forwards them to main memory, and thus reads are observed in-order. Writes, however, can be observed after reads when the FIFO queue sends a write request to the main memory only at a later time.

3.4 Partial Store Order

PSO (Partial Store Order [23]), relaxes TSO by allowing to reorder writes after writes (but not writes to the same address) in addition to the reordering of writes after reads, relaxing tests A and B. For the implementation of PSO, we do not need any new component. We can rewire the WriteQueue components in a different way in order to allow reordering writes with other writes. We need a FIFO queue to preserve the order for writes to the same address, and hence

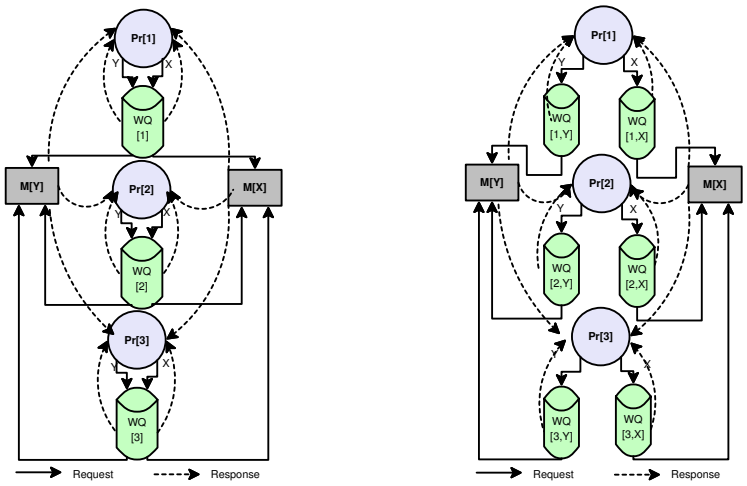

Fig. 3: Component diagram of TSO (left) and PSO (right)

for each processor we use a different instance of WriteQueue for each memory location. The glue logic in this case sends the outgoing reads and writes from the processor to the corresponding queue handling the write to this address. Different queues can be activated and send out writes at different times, non-deterministically. For example, suppose we write to X and then write to Y. The queue that holds writes to Y may become active and send the write it holds before the queue that holds the write to $\widehat{X}$, and thus effectively reorder writes.

In this case we have $n \times m$ instances of WriteQueue, denoted $WQ[i, j]$ for all $i = 0, ...n$ and $j = 0, ...m$. The connections between the components are shown in Fig. 3 (right).

3.5 Partial Store/Load Order

We consider another model we call PSLO (Partial Store/Load Order), where reads can be reordered in addition to writes (but we cannot reorder reads with later writes). In PSLO, writes are reordered in the same way as in PSO: we can reorder writes with later reads and reorder writes with writes to different locations. The only aspect of the model which is changed with respect to PSO is the order of reads.

The *SetRead* component reorders reads by keeping track of past values. A value that was written in the past to a certain location is feasible unless there was a fence operation after this write or a local write to the same location. In both of those cases, SetRead empties everything from the set of values except for the latest written value. SetRead makes use of the ability of the memory component to forward writes. Each time there is a write to the memory, the memory forwards the write to SetRead, and this value is added to the set. In response to a read

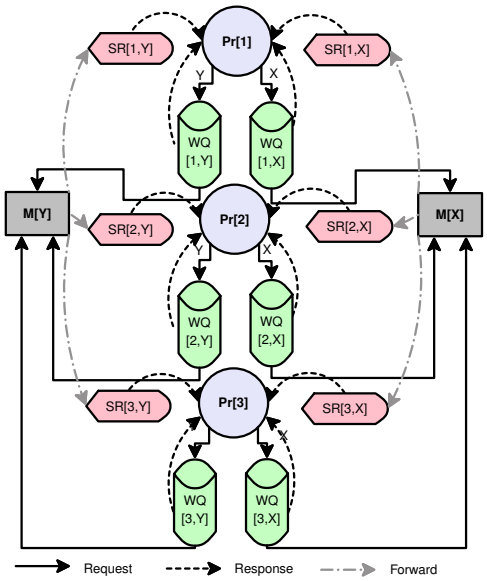

Fig. 4: Component diagram of PSLO

T1
Read $X \rightarrow r1$
Write $X \leftarrow 1$
Read $Y \rightarrow r2$
Write $Y \leftarrow 1$

Fig. 5: RMO reordering

request, SetRead selects non-deterministically one of the values in the set and returns a reply message containing this value.

Unlike writes, many memory models do allow reordering reads to the same address. In case we reorder reads, the first read can observe the current value of X and then the next read can observe a past value of X, which is already overwritten. Therefore, we can reorder reads by looking at past values.

In order to specify PSLO, we use the same components as in PSO, with the addition of the components $SR[i, j]$ for $i = 0, ...n$ and $j = 0, ...m$ which are instances of SetRead, the components are connected as shown in Fig. 4.

3.6 Relaxed Memory Order

SPARC's RMO (Relaxed Memory Order [23]) is a store atomic memory model that relaxes all the local orders except for the reordering of reads and writes with writes to the same address and dependent instructions. The difference between

PSLO and RMO is that in RMO reads can be reordered with later writes in the program order.

Reordering writes before reads means we need to either read from the future or effectively write to the past. Both are not straightforward to implement, but the former option is easier. A standard way to "read the future" is using prophecy variables [1]. In a response to a read request, we can guess a future value speculatively and then later ensure this speculative guess is justified.

Additionally, we need to make sure that postponed reads are not reordered with future writes to the same address. Because RMO may reorder both reads and writes. Consider, for example, the program in Fig. 5. RMO may not reorder *Read X→ r1* after the write to X. However, RMO may reorder the write to X after the two later operations and thus the read can be performed after both reading and writing to Y. To perform this, we need to queue reads together with the writes in the same queue. We therefore need a component that: 1. predict a value for the read and returns it to the processor. 2. Sends a read request to the queue, with a special flag that tells WriteQueue to queue the read together with the writes instead of responding to it immedietely. And finally: 3. Upon receiving a reply to the read, compare it to the guessed value.

The *FutureRead* component implements this idea. In response to a read request it either returns the current value or guesses a future value speculatively, keep track of the guessed value and then sends a "probe" read. Whenever there is a reply to a read, it is compared to the guessed value and in case they are different, the program cannot proceed. All guessed values have to be satisfied by the end of the program.

Using the FutureRead component, we can model SPARC's RMO. The implementation of this model is similar to that of PSLO, with the addition of $FR[i, j]$ instances of FutureRead between the process and each write queue. We keep SetRead at the end of the queue so that reads to the same address would be reordered. The diagram for RMO is shown in Fig. 6.

3.7 Non-Store-Atomic RMO

The NRMO model is a variation of PowerPC and ARM memory models. In this non-store atomic version of RMO, there is no global memory order. Each thread may observe stores from different threads in a different order, and therefore the NRMO specification does not use one main memory as in the previous models. Instead, each thread has its own local memory. We do need to preserve coherence and ensure all writes to the same address would be observed in a total order. For n processes and m memory locations, we can achieve this using $n \times m$ instances of WriteQueue we call *Coherence Queues*. Each coherence queue $CQ[i, j]$ is responsible for updating the private memory for location j in thread i. When a thread writes a value to address j, the same write is sent to all CQ's $CQ[0, j]...CQ[n-1, j]$ simultaneously, and therefore all of the coherence queues for the same location contains writes in the same order, but each of them may dispatch writes to the private memory at different time. As a result, writes to

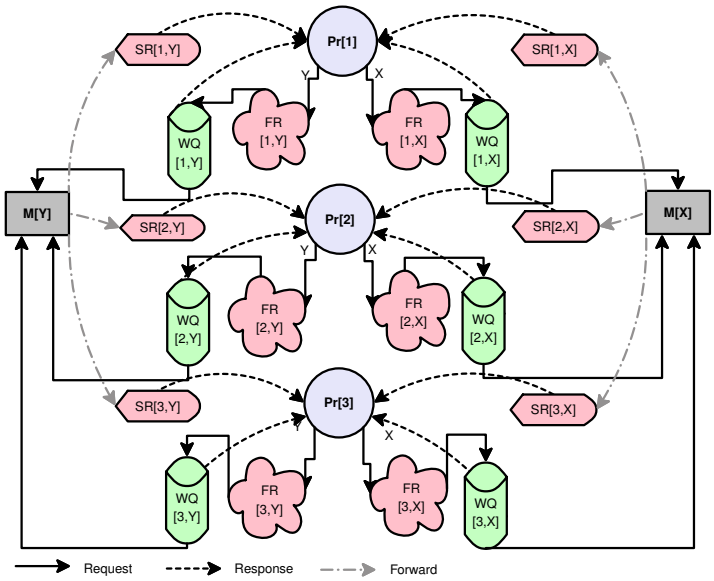

Fig. 6: Component diagram of RMO

different addresses can be seen by different processors in different orders, but writes to the same address are observed in the same total order.

We need to be careful when we connect the Coherence Queues to the Processors. Connecting the queues to the processors directly would give us a model which is too strong. Any model that allows reordering writes, such as PSO (and also in NRMO, which subsumes it), should allow ordering the writes from different threads independently. However, using only coherence queues, writes to different addresses cannot be ordred independently. In order to avoid this issue, we need another layer of queues between each Processor and the Coherence Queues, so that different writes could be ordered in each processor independently of other processors. This is done by maintaining the $n \times n$ instances of WriteQueue, denoted $WQ[i, j]$, as in PSO and RMO.

Another subtle issue is forwarding of local writes. The invariant we need to preserve is that as long as there are pending writes from $Pr[i]$ on the way to the local memory of the same processor, we should not be able to observe any value other than the latest such write. In order to support this, we need to forward from $CQ[i, j]$ the value of any write from thread i. Finally, we enable reordering reads after later reads and writes by adding a FutureRead components $FR[i, j]$ and SetRead components $SR[i, j]$ just as in RMO. The connection between components is shown in Fig. 7.

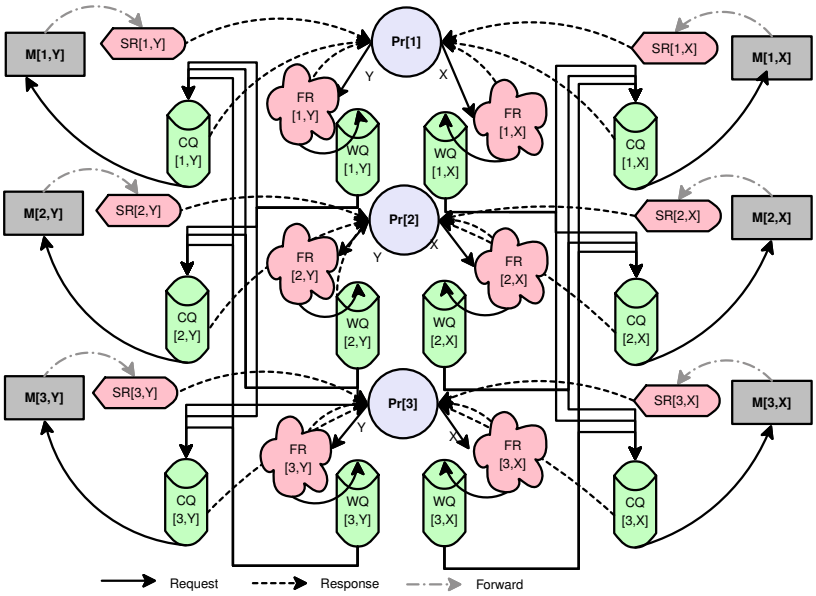

Fig. 7: Component diagram of NRMO

3.8 Non-Store-Atomic TSO

In NTSO, the non-store-atomic version of TSO, every two threads may observe writes from other different threads to different addresses in a different order, but program order is always preserved. This model is similar to the original x86 memory model. Surprisingly, although TSO is one of the simplest memory modes, its non-store-atomic version is our most complicated modular specification.

The source of complication is the need to preserve both program order between writes from the same thread (not necessarily to the same address), as well as a total coherence order between write to the same address (not necessarily from the same thread). In this model, Test A5 should pass and Test A2 must fail. This is done by maintaining two different set of queues: one keeps writes from a single thread in program order, and the other keeps writes to the same address from all threads in a total order. Finally, we need to combine both by allowing writes only when they are available from both sets simultaneously. This requires a kind of guarded glue logic that can be implemented in Promela in the the following way:

```
:: wqout[i].ch[k]?m1 & cqout[i].ch[j]?m2 ->
   m1.val==m2.val; min[k].ch[j];
```

We need $n \times n$ WQ queues, where $WQ[i,k]$ receives writes from thread i and write to the local memory of thread k. This way the model can send writes from the same threads at different times to different models. The connections in this model are presented in Fig. 8.

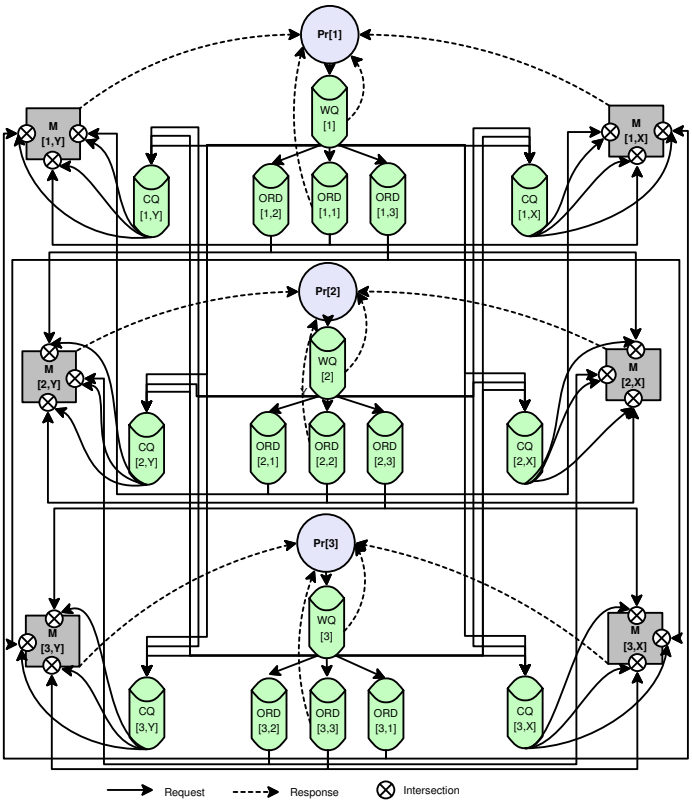

Fig. 8: Component diagram of NTSO

4 Comparing memory models

This section presents a technique for comparing memory models. Our goal is to check the difference between two models, and when the two models are not equivalent, to generate a litmus test that shows the difference between the two. Two memory models M and M' are not equivalent if any program displays different behaviors under M and M'.

Based on a review of published litmus tests in the literature and our own experience, tests that detect differences between memory models tend to be small, and hence an exhaustive search of test programs up to a given bound is a plausible approach for debugging memory model specifications. Given upper bounds for the total number of instructions in a program, the number of operations per thread, the number of threads as well as the number of memory locations, the technique exhaustively explores all programs within these bounds.

We start by defining the test program space for contrasting memory models. We present reduction techniques for trimming down the number of programs to a manageable size. Finally, we discuss techniques to efficiently compare the set

Test B1

T1	T2
Write $Y \leftarrow 1$	Read $X \rightarrow r1$
Read $Y \rightarrow r2$	Fence
Write $X \leftarrow 2$	Read $Y \rightarrow r3$

Test B2

T1	T2
Write $X \leftarrow 1$	Read $Y \rightarrow r1$
Read $X \rightarrow r2$	Fence
Write $Y \leftarrow 2$	Read $X \rightarrow r3$

Test B3

T1	T2
Read $Y \rightarrow r1$	Write $X \leftarrow 1$
Fence	Read $X \rightarrow r2$
Read $X \rightarrow r3$	Write $Y \leftarrow 2$

Fig. 9: Address symmetry (B1 and B2); Thread symmetry (B2 and B3)

of possible outcomes for a given program both for operational and axiomatic specification styles.

4.1 Test programs

A test program is a concurrent program consisting of n threads, $t_1, ...t_n$, where each thread is a sequence of memory operations. A memory operation can be one of:

- Read $Addr \rightarrow reg$ - a read from a constant address to a register
- Write $Addr \leftarrow Val$ - a write of a constant value to a constant address
- Fence - a full memory ordering barrier (fence)

The above three instructions suffice to contrast the models we have considered in this paper. Our methodology as well as the tool can be extended to include other instructions and data dependencies.

4.2 Program enumeration

Even when considering small bounds on test size, the program space can be too big to be explored in a reasonable time. Thus, we reduce the number of tested programs to a smaller number of representatives that are still sufficient for finding differences. First, because all writes are constants, registers in the program are used only for defining the final outcome. Therefore, we assign a unique register to each read. Likewise, the actual values read or written are inconsequential. We are interested only in which stores each load instruction can read. So instead of exploring all different combinations of write values, we assign a unique value for each write. We also restrict the places where we add fences: fences at the beginning or end of a thread have no effect, nor does a fence followed by another fence, so we eliminate all fences that are not between two other instructions.

Next, we use the symmetry properties of the memory model to reduce the number of programs. We use two symmetries: address symmetry and thread symmetry. In Fig. 9, the two programs display address symmetry: we obtain Test B2 from Test B1 by switching the Xs with the Ys. These two programs display the same behaviors and therefore it is sufficient to test only one of them. Similarly, Test B3 is the same as Test B2 with thread T1 switched with T2. By transitivity, any combination of thread and address permutation are equivalent. Hence, Test B1 and Test B3 are also symmetric.

We generate only one representative for each symmetry class by assigning an order between elements in a permutation and sorting them, and then we generate programs with sorted elements only. We sort the addresses according to the order of their appearance in the program, starting from T1 and continuing to the next thread after the end of each thread: the first memory access in T1 is always to location 0, the next memory access could either be to 0 again or to 1 and so on. When the highest address accessed so far is i, the next memory operation involves any address between 0 to $i + 1$. Similarly, we perform thread symmetry reduction by sorting threads according to some lexicographical order between instructions. The order we use is $Write < Read < Fence$, where two writes (or reads) are sorted according to their address. By generating programs so that the threads are sorted according to this lexicographical order and addresses by the order of their appearance, the enumeration algorithm avoids generating symmetric tests.

The function *Enumerate*, shown in Algorithm 1, performs program enumertion with symmetry reduction. It works recursively by adding a new instruction (Read, Write or Fence) at each iteration, until *remain* is 0 (reached the instruction limit). Instructions are added to a thread until the thread length bound is reached, and then the algorithm generates the next thread. The function ensures address symmetry by restricing the address value to a number between 0 to *nextaddr*, so that the first address is always 0, the second address is 0 or 1 and so on. The function uses the variables *sym* and *symi* to maintain thread symmetry: as long as all instructions in thread t are identical to the corresponding instructions in $t - 1$, any new instruction added to t have to be greater or equal to the corresponding instruction in $t - 1$ (according to a predefined order). In case we add to t an instruction which is different that the instruction in the same possition in $t - 1$, it breaks the symmetry, and any subsequent instruction added to the current thread is unrestricted.

4.3 Redundant test elimination

Some test programs are redundant in the sense that these tests are either not going to detect any difference between memory models or are subsumed by smaller programs that detect the same difference. First, we conclude that some programs are redundant simply by looking at the program structure. Consider, for example, Test D in Fig. 10. In this case, there are no shared variables between the two threads, and any execution under any memory model would give the same outcome. Similarly, in Test E both variables are shared, but even SC (the strongest model we typically consider) allows all possible outcomes. In both tests, there is no possible conflict in SC and therefore no cases that could be relaxed under a weaker memory model. Furthermore, consider Test F in Fig 10. This test can be decomposed into two separate tests: Test F1 includes T1 and the first two instructions in T2, and test F2 includes the last two instructions in T2 and T3. Test F is not going to exhibit any behaviors that can not be detected by F1 and F2, because the only relation between the two is the program order relation between instruction 2 and 3 in T2.

Algorithm 1 Enumerate(Program p, int remain, bool sym, int nextval, nextaddr, int len)

```
Require: p: the program. Initially empty
  remain: number of remaining instructions. Initially max instructions
  sym: true if the thread is the same as previous thread. Initially false
  nextval: the next available value. Initially 1
  nextaddr: next available address. Initially 0
  len: the maximal length of a thread
  if remain = 0 then
    Test(p)
  else
    if last thread in p=len then
      if remain < len then
        len ← remain
        sym ← False
      else
        sym ← True
      end if
      Add new thread to p
    end if
    th ← the last thread in p
    if th is empty and len > 1 then
      Enumerate(p, remain, False, nextval, nextaddr, len − 1)
    end if
    symi ← the instruction in the previous thread in the same location as the next in
    struction in the current thread
    if not sym or symi type is Write then
      for i = the address of symi to nextaddr do
        Add Write i ← nextVal to th
        newsym ← True if added instruction is the same as symi
        Enumerate(p, remain − 1, newsym, nextval + 1, (i = nextaddr)?nextaddr +
        1 : nextaddr, len)
      end for
    end if
    if not sym or symi type is not Fence then
      for i = the address of symi to nextaddr do
        Add Read i to th
        newsym ← True if added instruction is the same as symi
        Enumerate(p, remain − 1, newsym, nextval + 1, (i = nextaddr)?nextaddr +
        1 : nextaddr, len)
      end for
    end if
    if not first instruction in thread and previous instruction is not Fence then
      and Fence to th
      newsym ← True if this is the same as symi
      Enumerate(p, remain − 1, newsym, nextval, nextaddr, len)
    end if
  end if
```

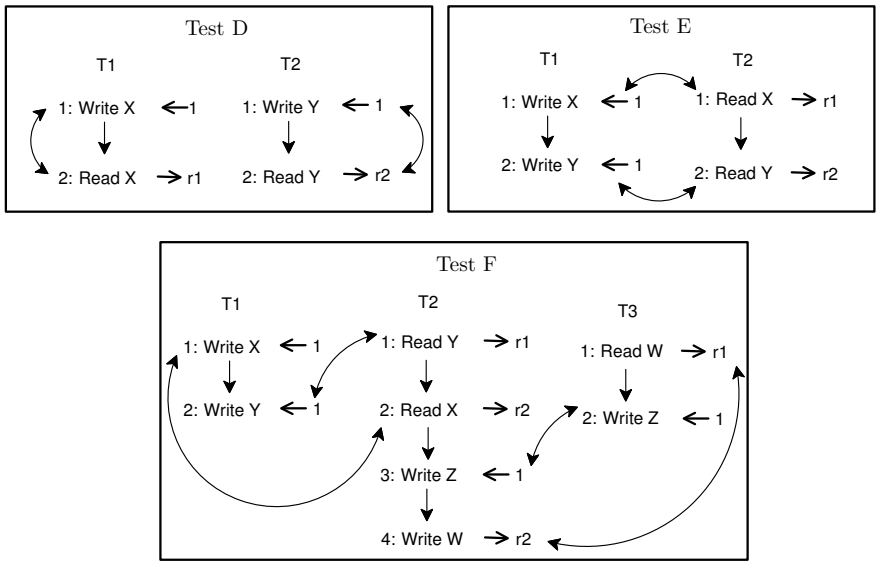

Fig. 10: Redundant tests

We eliminate such redundant test programs by generating a *conflict graph* for the test program. A conflict graph G is a directed graph where each operation is a node and the edges represent potential conflicts between the operations. For every two operations, X and Y, there is an edge in G from X to Y if either: (1) $X <_p Y$, or (2) either of X or Y are write operations and both access the same address. A test is redundant if the conflict graph G for this test is not strongly connected, i.e., there are operations X and Y in the graph such that there is no path from X to Y. For example, in Test C, there is no path from instruction 3 to instruction 2 in T2, and therefore this test is redundant.

Given a program P whose conflict graph in not strongly connected, we partition the instructions in P into two partitions, P_1 and P_2, such that no variables are shared between P_1 and P_2, and if x is an instruction in P_1 and y is an instruction in P_2 and both x and y are in the same thread, then $x <_p y$. We expect that for such a program, no instruction in P_1 would interfere with the execution of P_2 and vice versa, and hence the cross product of the outcomes of the program in partition P_1 and the outcomes of the program in partition P_2 is the set of outcomes of P. Therefore, if P detects a difference between two models, either P_1 or P_2 should detect a difference as well.

4.4 Computing all outcomes of a test program

For each of the test programs we determine if the set of outcomes of P running under a memory model M is the same as for P running on M'. The approach we take is to find all possible outcomes under both models independently and then compare them.

Finding all outcomes for an operational memory model is done in a manner similar to Park and Dill [22]. We use a model checker to find the reachable state space of the model. We extract the outcomes from the set of reachable final states found by the model checker. Our initial experiences in translating the operational models into Promela and running Spin [15] resulted in an inefficient exploration tool. Consequently, we implemented a custom explicit state enumeration model-checker in C++.

We implemented all the components described in Section 3.1 as C++ classes. We can instantiate those classes and connect them by defining the glue logic, as described in Section 3.1. The naive algorithm simply explores all possible execution paths, *including* all non-deterministic choices. Each execution is terminated when all of the components get to their final states.

In order to explore a smaller number of paths, we use the idea of *Sleep Sets*, introduced by Godefroid [13]. Intuitively, Sleep Sets is a partial order reduction that avoids trying out redundant partial orders between *independent* concurrent transitions. Two transitions t_1 and t_2 are independent if by executing t_1 and then t_2 we get to the same state as the state we reach by executing t_2 and then t_1. This reduction is *dynamic* in the sense that we decide if pairs of transitions are independent dynamically during the execution of the program, unlike *static* partial order reductions, such as those implemented in SPIN [15], where all the information used of order reduction is extracted from the syntactic description of the model before its execution.

In order to adapt Sleep Sets to our framework, we had to define when two transitions are considered as independent. One simple way to test for the independence of two transitions, x and y, finding the set of components that are involved in each transition. In each transition, one of the components makes a step and send a message to another component. The receiving component can be a different component at different times when the sending component is being executed. If non of the components that are involved in x are also involved in y, those transitions are independent.

In some cases, however, two transitions are independent even when they involve common components. Suppose, for example, that in transition x, component C_1 sends a message m to C_3, and in y, component C_2 sends the same message m to C_3. In this case, even though both transitions involve C_3, we can switch the order in which they execute and we will still get to the same state because both send the same message. Another optimization, which is specific to one of the component classes in our framework concerns the FIFO buffer in WriteBuf. In a FIFO buffer, we can switch the order between queue and dequeue operations and get to the same states, unless buffer is initially empty. Therefore if in transition x some other component sends a write message to WriteBuf, and in transition y the same WriteBuf component writes from the buffer, those two transitions are also independent. Both optimizations are implemented in our system.

We also implemented state caching, containing states that were already visited. When the model checker visits a state which is already in the cache, it skips

Operational / Axiomatic	SC	TSO	PSO	RMO	NTSO	NPSO
SC	-	$1s/4/2$	$1s/4/2$	$1s/4/2$	$8s/4/2$	$1s/4/2$
TSO	$1s/4/2$	-	$1s/4/2$	$1s/4/2$	$130s/5/3$	$1s/4/2$
PSO	$1s/4/2$	$1s/4/2$	-	$1s/4/2$	$8s/4/2$	$16s/5/3$
RMO	$1s/4/2$	$1s/4/2$	$1s/4/2$	-	$8s/4/2$	$16s/5/3$
NTSO	$2s/4/2$	$39s/5/3$	$2s/4/2$	$2s/4/2$	-	$2s/4/2$
NPSO	$2s/4/2$	$2s/4/2$	$40s/5/3$	$2s/4/2$	$9s/4/2$	-

Table 1: Contrasting axiomatic and operational models: time/instructions/threads

exploring any transitions from this state, since they where already explored. Unlike full state hashing, however, not every visited state have to be in the cache, and the same cache line can be overwritten by newer states.

For memory models specified axiomatically, the model is translated into a propositional formula. The model is specified as a set of first order formulas. In the context of finite programs all the variables have finite domains, so we convert the specification into predicate calculus by unfolding the quantifiers. A satisfying assignment is obtained by a SAT solver, which is one possible outcome of the program. To find all possible outcomes, we add the clause representing the negation of the outcome to the model and run the SAT solver again. As long as there are additional possible outcomes, the SAT solver returns another satisfying assignment. We repeat this process iteratively until the model becomes unsatisfiable. As we only add constraints to the model, the SAT solver uses conflict clauses from previous runs to make subsequent iterations faster. For the prototype, we used minisat [12] as the SAT solver.

5 Experiments

This section describes the experiments we performed to demonstrate the feasibility and usefulness of our approach, including: (1) measuring the execution time for contrasting the operational and axiomatic specifications of six memory models, (2) showing the effectiveness of the reductions targeted at reducing the number of test programs considered, and (3) performing two case studies in which the tool is used to debug memory model specifications.

5.1 Comparing different memory models

We tested our technique by comparing the operational and axiomatic specifications for various memory models: SC, the three SPARC memory models, and the non-store-atomic extensions of TSO and PSO. As seen in Table 1, a counter example is found for most cases within less than a second. The slowest times occur when comparing models to their non-store-atomic extension, which takes over two minutes for TSO versus NTSO. The litmus tests produced by the tool as counter examples were mostly the litmus tests we expected. However, the tool found subtle errors in our initial operational specification for RMO and for NTSO, which we fixed.

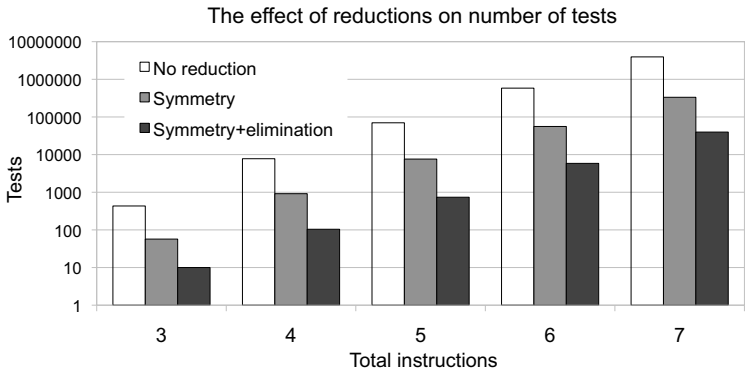

Fig. 11: The effect of reductions on the number of tests.

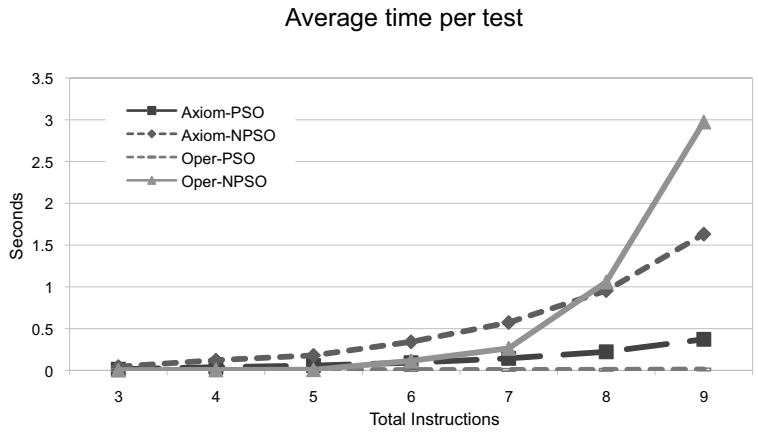

Fig. 12: Average run time per test

5.2 Test reductions and scalability

The graph in Fig. 11 shows the number of tests generated with up to three memory locations, up to three instructions per thread, and a varying number of total instructions. Fences are not counted towards the total number of instructions. Symmetry reductions provide approximately a 10x reduction in the number of tests, and redundant program elimination provides an additional 10x reduction, resulting in an overall reduction by a factor of 100x in the number of generated tests. The graph in Fig. 12 shows the average time per test for both operational and axiomatic memory models. As seen in this graph, the average time per test is no more than several seconds for programs with up to nine instructions, which means a bound of six or seven instructions can be explored in a reasonable time.

Test G

T1	T2
Read $X \rightarrow r1$	Read $Y \rightarrow r2$
Write $Y \leftarrow 1$	Write $X \leftarrow 2$

Outcome: $r1 = 2$; $r2 = 1$
time to find: 2s

Test H

T1	T2	T3
Read $X \rightarrow r1$	Read $Y \rightarrow r2$	Read $Z \rightarrow r3$
Write $Y \leftarrow 1$	Write $Z \leftarrow 2$	Write $X \leftarrow 3$

Outcome: $r1 = 3$; $r2 = 1$; $r3 = 2$
time to find: 824s

Test I

T1	T2
Write $X \leftarrow 1$	Write $Y \leftarrow 2$
Fence	Fence
Read $Y \rightarrow r1$	Read $X \rightarrow r2$

Outcome: $r1 = 0$; $r2 = 0$
time to find: 22s

Test J

T1	T2	T3
Write $X \leftarrow 1$	Read $Y \rightarrow r2$	Write $Y \leftarrow 2$
Fence	Read $X \rightarrow r3$	
Write $Y \leftarrow 2$		

Outcome: $r1 = 0$; $r2 = 2$; $r3 = 0$
time to find: 411s

Fig. 13: Litmus tests generated for buggy NPSO specifications

5.3 Debugging our axiomatic specification for NPSO

As a case study for using our technique for debugging a new memory model specification, we developed an axiomatic specification for NPSO, a non-store-atomic relaxation of PSO. We used an existing operational specification for NPSO as a reference model. We started with the axiomatic specification defined in Section 2, which is an extension of PSO that allows each thread to observe memory operations in a different order with the addition of a coherence axiom. We then ran the prototype with a bound of six instructions.

The prototype reported that Test G in Fig. 13 is allowed in the axiomatic but not in the operational specification. This is a well-known litmus test, which usually illustrates reorderings of reads after later writes. In this specification, however, we explicitly disallow reordering reads after writes. This outcome occurred because threads are not required to agree on the order of writes to different addresses. To correct the specification, we must rule out this kind of behavior and enforce some notion of causal transitivity. Our first attempt to fix it required that if a read sees a write to the same address in some thread, it can be ordered only after this read in the local thread that issued the write. Running the tool again after this modification generated Test H in Fig. 13. The proposed axiom was sufficient to rule out cycles involving two threads, but not cycles involving three threads and three addresses. We fixed this by using an alternative axiom, stating that if a read precedes a write to any address according to the local thread of this write, it will precede this write in any other thread.

After fixing the issue of causal transitivity, we ran the prototype again and received Test I in Fig. 13. This outcome is allowed when fences affect only local order and there is no total order among fences. We fixed it by adding an axiom that requires a total order between fences. In the final iteration, we received Test J in Fig. 13. In this case, the operational model drains both the local and the global queues after a fence, which rules out the outcome listed under Test J. A total order between fences is not sufficient to rule out this outcome.

Test K

T1	T2
Write $X \leftarrow 1$	Write $Y \leftarrow 3$
Write $Y \leftarrow 2$	Read $Y \rightarrow r2$
Read $Y \rightarrow r1$	Read $X \rightarrow r3$

Outcome: $r1 = 3$; $r2 = 3$; $r3 = 0$
time to find: 111s

Test L

T1	T2
Write $X \leftarrow 1$	Write $Y \leftarrow 2$
Fence	Read $Y \rightarrow r2$
Read $Y \rightarrow r1$	Read $X \rightarrow r3$

Outcome: $r1 = 0$; $r2 = 2$; $r3 = 0$
time to find: 43s

Fig. 14: Litmus tests generated for SOBER

We strengthen the total order axiom by requiring all threads to agree about the order between fences and any other operations. After fixing this axiom, we found no new mismatches between the models.

5.4 Debugging the axiomatic specification of TSO used in SOBER

The second case study for our technique was debugging the axiomatic specification of TSO used by SOBER [6]. SOBER is a technique for detecting potential SC violations in software. SOBER uses an axiomatically defined memory model that is intended to be equivalent to SPARC's TSO. The authors stated that their axiomatic definition is equivalent to their operational specification of TSO [7]. However, Burnim et al [8] discovered that SOBER's axiomatic specification and TSO are, in fact, not equivalent. We used SOBER's specification as a case study to see if our technique could detect the discrepancy between the two models without any prior knowledge about the nature of this discrepancy.

We compared SOBER's axiomatic specification with our operational specification for TSO. Our tool took less than two minutes to generate Test K in Fig. 14, which is allowed by TSO but not by SOBER's specification. Such a test is often used to distinguish TSO from IBM 370 [2], which is essentially TSO without forwarding. We then contrasted SOBER with IBM 370 and received Test L in Fig. 14, demonstrating that SOBER allows behaviors that are not allowed by IBM 370. We implemented a fix suggest by Burckhardt (personal communication), and we found no new mismatch between the fixed model our specification of TSO.

6 Related work

Many studies describe tools for testing litmus tests on a formally specified memory model [11, 21, 22, 24, 25]. Given a parallel program and an expected outcome, these tools report whether the specified outcome is feasible on a specified memory model. Most of these tools test for one outcome at a time [11, 21, 24, 25]. Park and Dill [22] presented a tool that enabled exploring all outcomes for a given parallel program using an operational specification for RMO.

Another approach for debugging a memory model is the "test model-checking" methodology [20]. In this approach, a memory model is verified against a state

machine that generates a non-deterministic sequence of writes and test for certain assertions. Each test-generating state machine is designed to detect a certain architectural rule. This approach provides a stronger verification than testing specific litmus tests.

A technique for validating that a system correctly implements a memory model is dynamic testing, which is used by tools such as TSOtool [14] and LCHECK [10]. These tools generate random tests, execute them on a certain hardware, and verify that the execution adheres to a given memory model.

Few studies involve a direct comparison between two memory models. Chatterjee et al [9] shows the equivalence of an operational specification of the Alpha memory model to an implementation of the same model. This work finds a refinement map between the two models via model-checking and uses an intermediate abstraction that exploit structural similarities between the two models to facilitate the proof. Other studies [11, 21] use theorem proving to prove equivalence between an operational and axiomatic specification of the same model.

7 Conclusions

We presented a technique for contrasting memory models and implemented a prototype based on this technique. Our experiments showed that this approach can detect differences between memory models within seconds or minutes, and the case studies showed that by contrasting memory models we can detect subtle differences between memory models that might have gone undetected using a predetermined set of litmus tests. Several key features make this technique a viable tool for debugging memory model specifications: it provides feedback in reasonable time, it generates a minimal-length litmus test as a counter example, which are easy to analyze and understand, it is fully automatic, and it is flexible and general in the sense that it can support different memory models, specification styles, and exploration techniques.

One limitation of our approach is that it does not provide a complete verification for the equivalence of two models. We test programs only up to a certain bound, and we cannot guarantee that there is no longer test that differentiates between the two specifications. Furthermore, redundant program elimination reductions may not be safe when comparing some models. We plan to extend this work to equivalence verification by finding sufficient bounds for a rich but restricted domain of memory models and prove that the reductions we use are safe for this domain of models.

Acknowledgements

We thank Sebastian Burckhardt for suggesting the use of SOBER's TSO specification as a case study for this paper.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theor. Comput. Sci. **82**(2) (1991) 253–284
2. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer **29** (1996) 66–76
3. Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Nardelli, F.Z.: The semantics of power and ARM multiprocessor machine code. In: DAMP. (2009)
4. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI. (2008) 68–78
5. Burckhardt, S., Alur, R., Martin, M.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: PLDI. (2007) 12–21
6. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV. (2008) 107–120
7. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. Technical Report MSR-TR-2008-12, Microsoft Research (2008)
8. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency in relaxed memory models. Technical Report UCB/EECS-2010-31, EECS Department, University of California, Berkeley (Mar 2010)
9. Chatterjee, P., Sivaraj, H., Gopalakrishnan, G.: Shared memory consistency protocol verification against weak memory models: Refinement via model-checking. In: CAV. (2002) 123–136
10. Chen, Y., Lv, Y., Hu, W., Chen, T., Shen, H., Wang, P., Pan, H.: Fast complete memory consistency verification. In: HPCA. (2009) 381–392
11. Chong, N., Ishtiaq, S.: Reasoning about the ARM weakly consistent memory model. In: MSPC, ACM (2008) 16–19
12. Een, N., Sorensson, N.: Minisat - a SAT solver with conflict-clause minimization. In: SAT. (2005)
13. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag Inc. (1996)
14. Hangal, S., Vahia, D., Manovit, C., Lu, J.Y.J.: TSOtool: A program for verifying memory systems using the memory consistency model. ISCA **32**(2) (2004) 114
15. Holzmann, G.J.: The model checker spin. IEEE Transactions on Software Engineering **23** (1997) 279–295
16. Holzmann, G.J.: The SPIN Model Checker : Primer and Reference Manual. Addison-Wesley Professional (September 2003)
17. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual. (March 2010)
18. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess program. IEEE Transactions on Computers **28**(9) (1979) 690–691
19. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL. (2005) 378–391
20. Nalumasu, R., Ghughal, R., Mokkedem, A., Gopalakrishnan, G.: The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In: CAV. (1998) 464–476
21. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs. (2009) 391–407
22. Park, S., Dill, D.L.: An executable specification and verifier for relaxed memory order. IEEE Transactions on Computers **48** (1999)

23. Weaver, D.L., Germond, T.: The SPARC Architecture Manual Version 9. Prentice Hall PTR (1994)
24. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: UMM: an operational memory model specification framework with integrated model checking capability. Concurr. Comput. : Pract. Exper. **17**(5-6) (2005) 465–487
25. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Analyzing the intel itanium memory ordering rules using logic programming and SAT. In: CHARME. (2003) 81–95

A Promela code for components

A.1 The Processor Component

```
proctype Proc(byte PR; Prog P; chan in,out)
{
   byte pc; Set incomp;
   chan observe = [PRLEN] of {DATA}; Msg m; Ins i;
   pc = 0;
   do
   :: in?m;m.type==r ->
      observe!m.val; remove(incomp,m.ins);
   :: in?m;m.type==c ->
      remove(incomp,m.val);
   :: empty(in) ->
      if
      :: P.ins?i; i.type==W; nfind(incomp,i.addr) ->
         m.type = W;m.addr=i.addr;m.val=i.val;
         m.proc=PR;m.ins=pc;out!m;
      :: P.ins?i; i.type==R ->
         m.type= R;m.addr=i.addr;m.val=i.val;
         m.proc=PR;m.ins=pc;out!m;
         insert(incomp,pc);
      :: P.ins?i; i.type==F; empty(incomp.ch) ->
         m.type= F;m.proc=PR;m.ins=pc;out!m
      :: empty(P.ins) & empty(incomp.ch) -> break;
      fi;
      pc = pc+1;
   od
}
```

A.2 The SimpleMem Component

```
proctype SimpleMem(byte ID; chan in,out)
{
   byte value; Msg m;
   value = 0;
   do
   :: in?m; m.type==R ->
      m.type = r; m.val=value;
```

```
      out!m;m.type=c out!m;
   :: in?m; m.type==W ->
      value = m.val; m.type= f; out!m
   od
}
```

A.3 The WriteQueue Component

```
proctype WriteQueue(byte P,Addr; chan in,out)
{
   chan q = [PRLEN] of {byte,byte,Msg};
   Msg m,m1;  byte a,p;
   do
   :: empty(in) -> q?_,_,m; out!m;
   :: in?m; m.type == W -> q!m.addr,m.proc,m;
   :: in?m; m.type == R -> if
      :: q??[eval(m.addr),eval(m.proc),m1] ->
         q??<eval(m.addr),eval(m.proc),m1>;m1.type=r;
         out!m1;m1.type=c;out!m1;
      :: else -> out!m
      fi
   :: in?m; m.type== F -> do
      :: q?a,p,m -> out!m
      :: empty(q) -> break
      od
   od
}
```

A.4 The SetRead Component

```
proctype SetRead(byte P,Addr; chan in,out)
{
   Set  s; byte last; last = 0;
   insert(s,0);
   Msg m;
   do
   :: in?m; m.type == W ->
      insert(s,m.val); out!m;
   :: in?m; m.type == F ->
      clear(s);insert(s,last);out!m;
   :: in?m; m.type == f ->
      last = m.val; insert(s,m.val);
   :: in?m; m.type == R ->
      choose(s,m.val); m.type=r;out!m;
      m.type=c; out!m;
   od
}
```

A.5 The FutureRead Component

```
proctype FutureRead(byte P,Addr; chan in,out)
{
   chan s = [MAXQUEUE] of {byte,byte};
   byte last; last = 0;
   Msg m; byte i,v;
   do
   :: in?m; m.type==f -> last=m.val; do
      :: s??[eval(m.val),i] ->
         s??eval(m.val),i;
         m.type= c; m.ins = i;
      :: else -> break;
      od
   :: in?m; m.type==R ->
      m.type = r; m.val=last; out!m;
   :: in?m; m.type==R ->
      random(NVAL,v); s!v,m.ins;
      m.type=r; m.val=v; out!m
   od
}
```