Generating Litmus Tests for Contrasting Memory Consistency Models*

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 [1]. The emergence of mainstream multi-core processors as well as recent developments in language-level memory models [3, 18], 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 dependencies, and different specification styles (operational and axiomatic). Architecture manuals include litmus tests that can be used to differentiate between memory models [15, 22], but these litmus tests are not complete, and coming up

* 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.

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 [22] 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 specification of TSO [5] with an operational specification of TSO and showed our technique detects a recently discovered specification error [7].

Initially $X = 0$; $Y = 0$

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

Is the outcome $r1 = 0$; $r2 = 0$ allowed?

Fig. 1: Testing write-after-read reordering

2 Specifying memory models

A *memory consistency model* is a specification of the shared memory semantics of a parallel system [1]. The simplest memory model is *Sequential Consistency* (SC) [16]. 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 [22], Intel's x86 [15], Intel's Itanium [24], ARM and PowerPC [2].

Consider for example the program in 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).

2.1 Operational specification

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 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.

For example, we have specified TSO using three different types of components that run concurrently [17]. A processor component produces a sequence of memory operations, a memory location component keeps track of the latest value written to a specific memory location, and a write queue component implements a FIFO buffer for write messages, and supports read forwarding. These

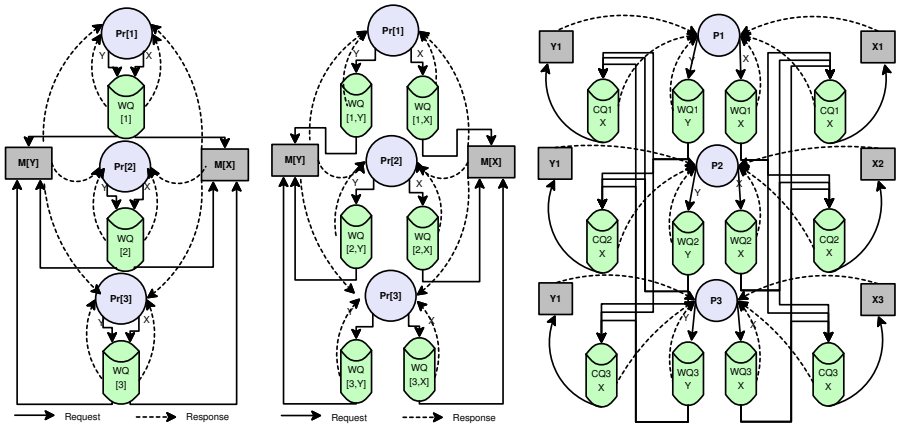

Fig. 2: Component diagram of TSO (left), PSO (middle) and NPSO (right)

components are connected in the configuration described in Fig. 2 (left) to implement TSO. Each processor is connected to a single write queue (WQ) that releases writes from this processor to the main memory.

SPARC's PSO (Partial Store Order [22]), a memory model that relaxes TSO by allowing to reorder writes after writes to different addresses. It can be specified in a similar manner, using the configuration illustrated in Fig. 2 (middle). Instead of one queue per processor, there is a queue per address for each of the processors. Writes to different addresses are stored at different queues, which can send the writes in any order, thus enabling reordering writes after later writes.

The two previous models are store atomic (all threads observe writes from other threads in the same order). In non-store-atomic memory models, different threads do not have to agree on the order of writes from other threads. As an example for a non-store-atomic memory model we present here NPSO, the non-store-atomic version of PSO. The diagram in Fig. 2 (right) presents the operational specification for NPSO. Because each thread may observe stores from different threads in a different order, the NPSO specification does not use one main memory as in the previous models. Instead, each thread has its own local memory. To preserve coherence and ensure all writes to the same address would be observed in a total order. The model maintains coherence by using an additional layer of write queues.

We have defined operational specifications in this style for additional memory models [17] such as RMO and the non-store-atomic versions of each of the store atomic models (NTSO and NPSO). To model these, we add additional component types. The encoding for RMO, for example, requires the ability to read future values to reorder reads after later writes.

2.2 Axiomatic specification

An alternative approach is the axiomatic style of specifications, 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 [4], 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 TSP by allowing reordering writes with other writes to a different location. 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.

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. To ensure transitive causal order between operations, the view includes all operations and not only writes.

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 4, this specification is too weak. In Section 4.3, we use our technique to develop the missing axioms for NPSO.

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

Test A

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

Test B

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

Test C

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

Fig. 3: Address symmetry (A and B); Thread symmetry (B and C)

3.1 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.

3.2 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. 3, the two programs display address symmetry: we obtain Test B from Test A 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 C is the same as Test B with thread T1 switched with T2. By transitivity, any combination of thread and address permutation are equivalent. Hence, Test A and Test C 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

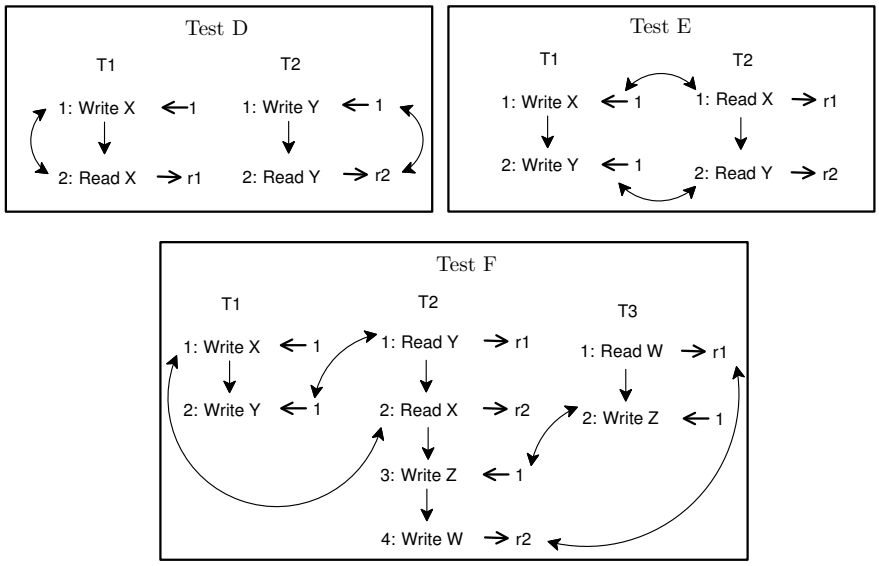

Fig. 4: Redundant tests

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.

3.3 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. 4. 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 4. 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.

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.

3.4 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 [21]. 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 [14] resulted in an inefficient exploration tool. Consequently, we implemented a custom explicit state enumeration model-checker in C++ using sleep-set partial order reduction [12] and state caching.

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

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

conflict clauses from previous runs to make subsequent iterations faster. For the prototype, we used minisat [11] as the SAT solver.

4 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.

4.1 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.

4.2 Test reductions and scalability

The graph in Fig. 5 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. 6 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.

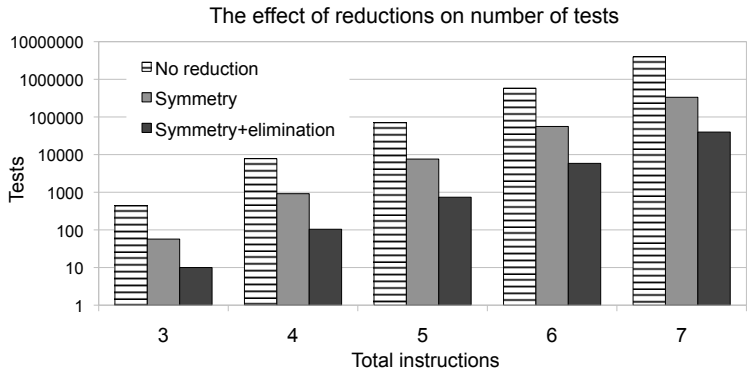

Fig. 5: The effect of reductions on the number of tests.

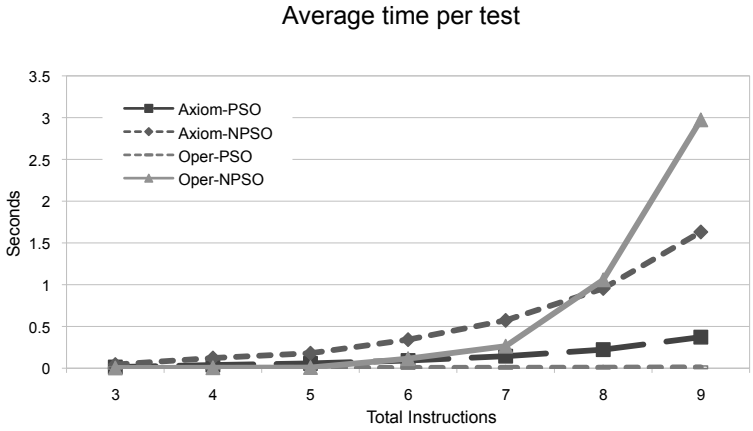

Fig. 6: Average run time per test

4.3 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 2.2, 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. 7 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

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. 7: Litmus tests generated for buggy NPSO specifications

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. 7. 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. 7. 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. 7. 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. 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.

4.4 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 [5]. 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 [6]. However, Burnim et al [7] discovered that SOBER's axiomatic specification and TSO are, in fact, not equivalent. We used SOBER's specification as a case study

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. 8: Litmus tests generated for SOBER

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. 8, which is allowed by TSO but not by SOBER's specification. Such a test is often used to distinguish TSO from IBM 370 [1], which is essentially TSO without forwarding. We then contrasted SOBER with IBM 370 and received Test L in Fig. 8, 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.

5 Related work

Many studies describe tools for testing litmus tests on a formally specified memory model [10, 20, 21, 23, 24]. 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 [10, 20, 23, 24]. Park and Dill [21] 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 [19]. 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 [13] and LCHECK [9]. 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 [8] 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 [10, 20] use theorem proving to prove equivalence between an operational and axiomatic specification of the same model.

6 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. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer **29** (1996) 66–76
2. 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)
3. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI. (2008) 68–78
4. Burckhardt, S., Alur, R., Martin, M.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: PLDI. (2007) 12–21
5. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV. (2008) 107–120

6. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. Technical Report MSR-TR-2008-12, Microsoft Research (2008)
7. 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)
8. Chatterjee, P., Sivaraj, H., Gopalakrishnan, G.: Shared memory consistency protocol verification against weak memory models: Refinement via model-checking. In: CAV. (2002) 123–136
9. Chen, Y., Lv, Y., Hu, W., Chen, T., Shen, H., Wang, P., Pan, H.: Fast complete memory consistency verification. In: HPCA. (2009) 381–392
10. Chong, N., Ishtiaq, S.: Reasoning about the ARM weakly consistent memory model. In: MSPC, ACM (2008) 16–19
11. Een, N., Sorensson, N.: Minisat - a SAT solver with conflict-clause minimization. In: SAT. (2005)
12. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer-Verlag Inc. (1996)
13. 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
14. Holzmann, G.J.: The model checker spin. IEEE Transactions on Software Engineering **23** (1997) 279–295
15. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual. (March 2010)
16. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess program. IEEE Transactions on Computers **28**(9) (1979) 690–691
17. Mador-Haim, S., Alur, R., Martin, M.: Generating litmus tests for contrasting memory consistency models - extended version. Technical report, Dept. of Computer Information Science, U. of Pennsylvania (2010)
18. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL. (2005) 378–391
19. 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
20. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs. (2009) 391–407
21. Park, S., Dill, D.L.: An executable specification and verifier for relaxed memory order. IEEE Transactions on Computers **48** (1999)
22. Weaver, D.L., Germond, T.: The SPARC Architecture Manual Version 9. Prentice Hall PTR (1994)
23. 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
24. 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