A checker for SPARC memory consistency

Matthew Naylor Simon Moore

University of Cambridge

1 Introduction

A *memory consistency model* defines the allowable behaviours of a set of program threads acting on shared memory and is a prerequisite for the development of well-defined concurrent programs.

In this report, we address the problem of verifying a hardware implementation of shared memory against the SPARC memory consistency models.

The hardware that we are verifying is the memory subsystem used in the BERI multi-processor [1], however the verification tool we present could be applied to other memory subsystems too.

We choose to verify the memory subsystem in isolation, independently of the processor pipeline, for the following reasons.

- BERI has a simple in-order pipeline that interacts straightforwardly with the memory subsystem.
- Our memory subsystem is not specific to BERI and could be used in other multi-processor implementations too.
- We can apply greater stress to the memory subsystem directly than would be possible indirectly via the BERI pipeline.
- It is faster to simulate the memory subsystem in isolation, allowing more tests per unit time.

Although BERI largely follows the MIPS ISA, we have chosen to use the SPARC consistency model because the SPARC ISA formally defines a consistency model that is well-known and supported by major software stacks.

1.1 The problem

Given a *trace* consisting of one sequence of memory instructions per thread (including loads, stores, atomic read-modify-writes, and memory barriers), and the value returned by each load, determine whether this trace satisfies one of the SPARC consistency models: *sequential consistency* (SC), *total store order* (TSO), *partial store order* (PSO), or *relaxed memory order* (RMO).

To simplify the problem slightly, we assume that every store operation in a trace writes a *unique* value. This is easily arranged by an automatic test generator, and is justified by the fact that the actual values being stored do not typically affect any interesting hardware behaviour.

1.2 The challenge

Memory consistency models are highly non-determinisic meaning that even a small program can have a very large number of allowed behaviours. In our experience, naive exhaustive enumeration of all behaviours does not usefully scale to checking traces beyond 10-15 instructions in size. We would like to be able to check traces containing many thousands of instructions in seconds.

One possible way to reduce the amount of non-determinism is to modify the hardware to emit extra trace information that can rule out many possible behaviours in the model. In this report however, we treat the memory subsystem as a *black box*: we do not inspect or modify its internals in any way.

1.3 Our solution

We present a checking tool for SPARC memory consistency which includes:

- an executable *operational semantics* and an equivalant *axiomatic semantics* for each consistency model (SC, TSO, PSO and RMO);
- the axiomatic version can generate constraints that are fed to a state-of-the-art SMT solver;
- an automatic test framework that can check the operational and axiomatic versions of the models for equivalence on large numbers of randomly generated traces.

Our checker is inspired by a previous tool called *TSOtool* [2,3], but has the following differences.

- We support checking SC, PSO and RMO models as well as TSO.

- We use an existing general-purpose constraint-solver rather than implementing our own custom solver. This simplifies the checker, making it easier to understand and extend.
- We present operational *and* axiomatic flavours of the models, and check them for equivalance.
- Our checker is freely available and open-source:

  `http://github.com/CTSRD-CHERI/axe`

- Early measurements suggest that the performance of our checker approaches the published performance of TSOtool.

2 Trace format

We introduce the syntax of traces by way of two examples.

Example 1 Here is a simple trace consisting of five instructions running on two threads. The number before the `:` denotes the thread id. It is assumed that the textual order of instructions with the same thread id is the order in which those instructions were submitted to the memory subsystem by that thread. However, no ordering is implied between instructions running on different threads.

```
0: M[1] := 1
0: sync
0: M[0] == 0
1: M[0] := 1
1: M[1] == 0
```

The first line can be read as: thread `0` stores value `1` to memory location `1`. The second line as: thread `0` performs a memory barrier. And the final line as: thread `1` reads value `0` from memory location `1`. In this report, we will assume that the initial value of every memory location is `0`. As we will see shortly, this trace is valid according to the TSO model, but not SC.

Example 2 Here is another trace, this time containing three instructions, this first of which is an atomic read-modify-write instruction.

```
0: <M[0] == 0; M[0] := 1>
1: M[0] := 2
1: M[0] == 1
```

The first line can be read as: thread `0` *atomically* reads value `0` from memory location `0` and updates it to value `1`. (The two memory addresses in an atomic operation must be the same.) It is straightforward to convert a pair of *load-linked* and *store-conditional* operations to this form:

- if the store-conditional fails, then remove it from the trace and convert the load-linked to a standard load;
- otherwise, convert both operations to a single read-modify-write.

As we will see shortly, this trace is not valid according to any of the models presented in this report. The trace implies that the second instruction occured between the read-modify-write, which should have been atomic.

3 Operational semantics

We define the behaviours allowed by each SPARC consistency model using an abstract machine consisting of a state and a set of state-transition rules.

In each case, the state consists of:

- A trace T (a sequence of instructions in the format described above).
- A mapping M from memory addresses to values.
- A mapping B from thread ids to sequences of instructions. We call $B(t)$ the *local buffer* of thread t.

In the *initial state*, T is the trace we wish to check, $M(a) = 0$ for all addresses a, and $B(t) = []$ for all threads t. (Notation: $[]$ denotes the empty sequence.)

Using the state-transition rules, if there is a path from the initial state to a state in which $T = []$ and $B(t) = []$ for all threads t then we say that the machine accepts the initial trace and that the initial trace is allowed by the model. Otherwise, it is disallowed by the model.

We now define the state-transition rules for each model.

3.1 Sequential Consistency (SC)

Although simple to define and understand, SC serves as a good introduction to the presentation style used in the following sections. It uses only one state-transition rule.

Rule 1 Pick a thread t. Remove the first instruction i executed by t from the trace.

1. If $i =$ `M[`a`]:=` v then update $M(a)$ to v.
2. If $i =$ `M[`a`]==` v and $M(a) \neq v$ then **fail**.
3. If $i =$ `<M[`a`]==` v_0`; M[`a`]:=` v_1`>` then:

   i. if $M(a) \neq v_0$ then **fail**;

   ii. else: update $M(a)$ to v_1.

We use the term **fail** to denote that the transition rule cannot be applied.

3.2 Total Store Order (TSO)

In TSO, each thread has a local store buffer. We define it using two rules. The first is similar to Rule 1 of SC, modified to deal with writing to and reading from the store buffers. The second deals with flushing elements of the buffers to memory.

Rule 1 Pick a thread t. Remove the first instruction i executed by t from the trace.

1. If $i =$ `M[`a`]:=` v then append i to $B(t)$.
2. If $i =$ `M[`a`]==` v then let j be the latest instruction of the form `M[`a`]:=` w in $B(t)$ and:

   i. if j exists and $v \neq w$ then **fail**.

   ii. if j does not exist and $M(a) \neq v$ then **fail**;
3. If $i =$ `sync` and $B(t) \neq []$ then **fail**.
4. If $i =$ `<M[`a`]==` v_0`; M[`a`]:=` v_1`>` then:

   i. if $B(t) \neq []$ then **fail**;

   ii. else if $M(a) \neq v_0$ then **fail**;

   iii. else: update $M(a)$ to v_1.

Rule 2 Pick a thread t. Remove the first instruction `M[`a`]:=` v from $B(t)$ and update $M(a)$ to v.

3.3 Partial Store Order (PSO)

PSO is similar to TSO but relaxes the order in which writes can be flushed from the buffer. In particular: writes to different addresses can be reordered.

Rule 1 This is identical to Rule 1 of TSO except that clause 4 becomes:

4. If $i =$ `<M[`a`]==` v_0`; M[`a`]:=` v_1`>` then:
   i. if any instruction in $B(t)$ refers to address a then **fail**;
   ii. else if $M(a) \neq v_0$ then **fail**;
   iii. else: update $M(a)$ to v_1.

Rule 2 Pick a thread t and an address a. Remove the first instruction that refers to address a, `M[`a`]:=` v, from $B(t)$ and update $M(a)$ to v.

3.4 Relaxed Memory Order (RMO)

RMO is a relaxation of PSO in which load instructions, like stores, become non-blocking: thread-local buffers can now contain loads *and* stores.

Rule 1 Pick a thread t. Remove the first instruction i executed by t from the trace.

1. If $i =$ `sync` and $B(t) \neq []$ then **fail**.
2. Otherwise, append i to $B(t)$.

Rule 2 Pick a thread t and an address a. Remove the first store or read-modify-write instruction that refers to address a from $B(t)$.

1. If $i =$ `M[`a`]:=` v then update $M(a)$ to v.
2. If $i =$ `<M[`a`]==` v_0`; M[`a`]:=` v_1`>` then:
   i. if $M(a) \neq v_0$ then **fail**;
   ii. else: update $M(a)$ to v_1.

Rule 3 Pick a thread t. Remove any load instruction `M[`a`]==` v from $B(t)$ and let w be the latest value to be written to a occurring before the load in $B(t)$.

1. If w exists and $v \neq w$ then **fail**.
2. If w does not exist and $v \neq M(a)$ then **fail**.

3.5 Efficiency

A simple but fairly effective performance improvement to the above operational models is to immediately backtrack when reaching a state that has been visited before.

4 Axiomatic semantics

The axiomatic semantics is a function from a trace to a set of constraints. If the constraints are satisfiable then the trace is valid according to the model; otherwise it is invalid. We use the general-purpose solver Yices [6] to decide satisfiability.

We use the term *program order* to refer to the order in which each a thread submits its instructions to the memory subsystem. (Program order only implies an ordering between instructions running on the *same* thread.)

In what follows, a read-modify-write instruction is considered to be both a load and a store.

4.1 Read-consistency

Any trace must be *read-consistent*, regardless of the consistency model being checked. A trace is read-consistent if and only if for each load l of value v from address a on thread t:

- the latest store to address a that preceeds l in program order writes the value v; or
- there exists a store of value v to address a by a thread other than t; or
- the value read b l is the intial value (0).

If a trace is not read-consistent then it does not satisfy any of the memory models presented in this paper.

4.2 Reads-from and write-order constraints

Any trace must also satisfy the following constraints, regardless of the consistency model being checked.

For each load l of value v from address a by thread t:

- if $v = 0$ then for each store s to address a by a thread other than t, add constraint $l < s$;
- if $v \neq 0$ then:
  (a) let s be the store of value v to address a (there must be exactly one such store[1]);
  (b) let $\mathit{tid}(s)$ be the thread id that executes s;
  (c) if $\mathit{tid}(s) \neq t$ then add constraint $s < l$;
  (d) let p be the latest store to address a that preceeds l in program order, if one exists;
  (e) if p exists and $\mathit{tid}(s) \neq t$ then add constraint $p < s$;
  (f) let n be the earliest store to address a that follows s in program order, if one exists;
  (g) if n exists and $\mathit{tid}(n) \neq t$ then add constraint $l < n$;
  (h) for each store $s' \neq s$ by a thread other than t and $\mathit{tid}(s)$ to address a, add constraint $(s' < s) \vee (l < s')$.

4.3 SC local constraints

Let i and j be any two instructions on thread t such that i preceeds j in program order. Add constraint $i < j$.

4.4 TSO local constraints

Let i and j be any two instructions on thread t such that i preceeds j in program order.

- If i is a load then add constraint $i < j$.
- If i and j are stores then add constraint $i < j$.
- If i is a `sync` or j is a `sync` then add constraint $i < j$.

[1]Otherwise the trace is either not read-consistent or stores are not unique.

4.5 PSO local constraints

Let i and j be any two instructions on thread t such that i preceeds j in program order.

- If i is a load then add constraint $i < j$.
- If i and j are stores *to the same address* then add constraint $i < j$.
- If i is a `sync` or j is a `sync` then add constraint $i < j$.

4.6 RMO local constraints

Let i and j be any two instructions on thread t such that i preceeds j in program order.

- If i is a load and j is a store *to the same address* then add constraint $i < j$.
- If i and j are stores *to the same address* then add constraint $i < j$.
- If i is a `sync` or j is a `sync` then add constraint $i < j$.

4.7 Generation of local constraints

We use a linear-time method of generating the local constraints, avoiding many redundant ones by construction. See `LocalOrder.lhs` in the companion repository.

4.8 Efficiency

Let us return to the generation of reads-from and write-order constraints defined above, specifically clause (h) which states:

```
for each store-load pair $(s, l)$ such that $l$ reads from $s$:
    for each store $s' \neq s$ to the same address as $s$:
        if $tid(s') \neq tid(s) \wedge tid(s') \neq tid(l)$:
            introduce constraint $(s' < s) \vee (l < s')$.
```

This clause produces $O(n^2)$ constraints for traces of size n. Such large numbers of constraints are not only expensive to generate, but they are also difficult to solve (conjunctions of disjunctions can easily lead to a combinatorial explosion of choices). Fortunately, many of these constraints are often redundant, implied by the other constraints present, and can simply be thrown away. Specifically, these "other constraints present" are the local order constraints alongside the constraints produced by clauses (a)-(g): we refer to them collectively as the *definite constraints* (they do not contain disjunctions).

One possible pruning approach is to compute the transitive closure of the definite constaints and then to discard any disjunctions of the form $(a < b) \vee (c < d)$ if b is reachable from a or d is reachable from c. However, computing the transitive closure is $O(n^3)$ and iterating over the disjunctions is $O(n^2)$.

Instead, we use an $O(n\ log(n))$ data-flow analysis. Given a set of definite constraints between memory operations, it computes, for each operation x, the latest store on each thread that writes to the same address as x and is known to preceed x. We call this $prec_t(x)$. Similarly, if applied in reverse, it can determine the earliest store on each thread that writes to the same address as x and is known to succeed x. We call this $succ_t(x)$. Once computed, we replace clause (h) with:

```
for each store-load pair $(s, l)$ such that $l$ reads from $s$:
    for each thread $t$ such that $t \neq tid(s) \wedge t \neq tid(l)$:
        for each store between $prec_t(s)$ and $succ_t(l)$ in
        program order:
            introduce constraint $(s' < s) \vee (l < s')$.
```

Now for the data-flow analysis that computes $prec_t(x)$ and $succ_t(x)$ for all t and x in $O(n\ log(n))$ time. The set of definite constraints can be viewed as a graph, with memory operations as nodes. The information that flows along the edges, and which reaches each node x is referred to as $info(x)$. It is a mapping $(t, a) \rightarrow s$ where t is a thread id, a an address and s a store operation; it maps the latest (earliest) store to some address by some thread known to preceed (succeed) the node.

The analysis proceeds as follows:

- remove the node x with the fewest incoming edges;
- let a be the address of operation at node x;
- for each thread t, assign $succ_t(x) \leftarrow info(x)(t, a)$;
- if x is a store, assign $info(x)(t, a) \leftarrow x$ where t is the thread id of x;

- for each successor y of x, assign $\mathit{info}(y) \leftarrow \mathit{merge}(\mathit{info}(x), \mathit{info}(y))$;
- (the *merge* function merges two maps, resolving collisions by perferring the store operation that comes later in program order);
- discard $\mathit{info}(x)$ for space reasons.

The result of the analysis is $\mathit{succ}_t(x)$ for all t and x. To compute $\mathit{prec}_t(x)$ simply apply the same algorithm but this time reversing the direction of the edges in the graph, and reversing program order.

5 Results

We have tested the operational and axiomatic semantics of each model for equivalance on millions of small randomly-generated traces. Each such trace consists of ten instructions running on three threads. This gives us some confidence that our models are indeed defining what we intend them to define.

We have constructed an HDL-level test bench that randomly generates sequences of memory instructions that are applied to the various versions of the BERI memory subsytem. Before generating each test-sequence, a selection of shared memory addresses are picked at random to be used by that test-sequence (the number of addresses used is customisable). Each test-sequence results in a trace that is then checked by our tool. Loads, stores, and read-modify-write instructions are generated with a 31.25% probability, and memory barriers with a 6.25% probability. Figures 1–4 shows the performance of our checker in this role: we can check test-sequences containing a many thousand instructions in seconds. At the time of writing, we have two versions of the memory subsystem: one is TSO and the other RMO. Both pass millions of thousand-element test-sequences, which provides a high degree of confidence in the hardware under test.

6 References

[1] Bluespec Extensible RISC Implementation, *http://www.beri-cpu.org/.*

[2] TSOtool, *http://xenon.stanford.edu/∼hangal/tsotool.html.*

[3] Testing memory consistency of shared-memory multiprocessors, C. Manovit, PhD thesis, Stanford University, 2006.

[4] An executable specification, analyzer and verifier for RMO (relaxed memory order), S. Park and D. L. Dill. In proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), 1995.

[5] The SPARC Architecture Manual Version 9, D. L. Weaver and T. Germond, 2003.

[6] The Yices SMT Solver, Stanford Research Institute, *http://yices.csl.sri.com/*.

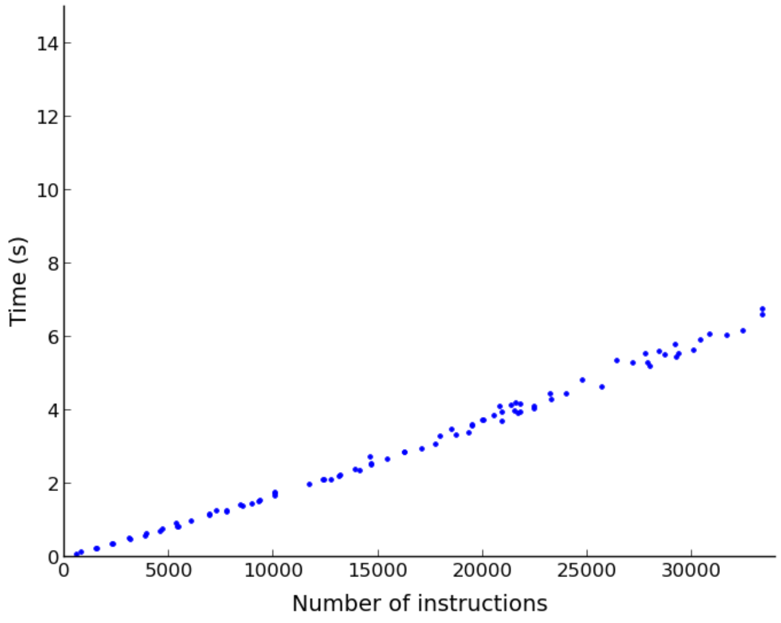

Figure 1: Checking a 3-thread version of the BERI memory subsystem against TSO with 4 shared variables

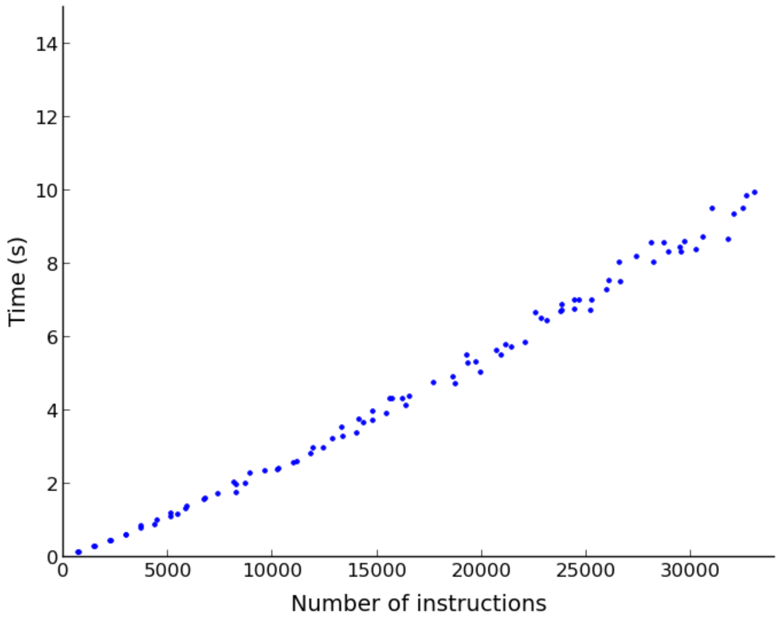

Figure 2: Checking a 3-thread version of the BERI memory subsystem against RMO with 4 shared variables

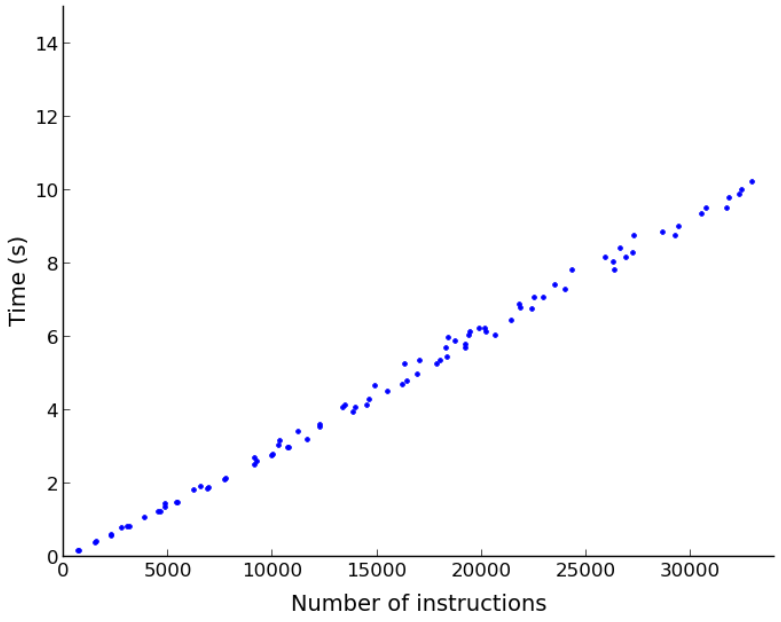

Figure 3: Checking a 3-thread version of the BERI memory subsystem against TSO with 16 shared variables

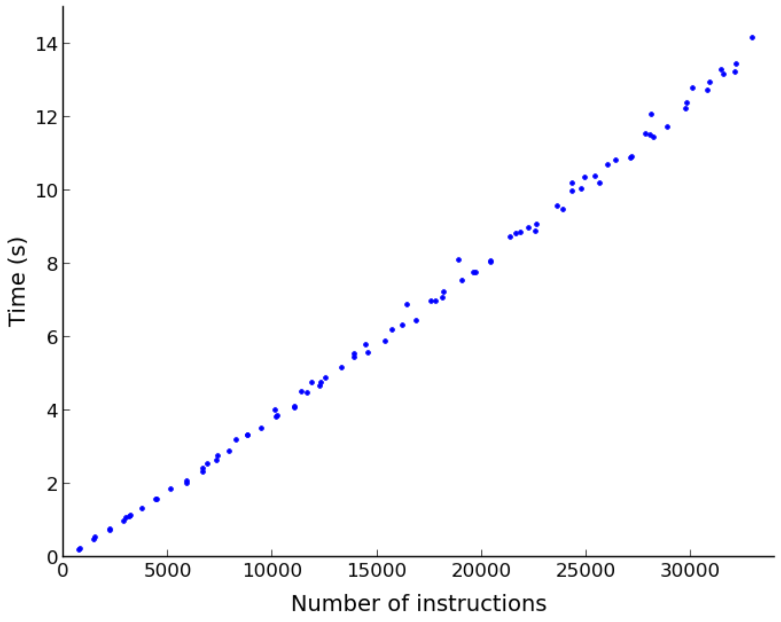

Figure 4: Checking a 3-thread version of the BERI memory subsystem against RMO with 16 shared variables