Axe: a consistency checker for memory-subsystem traces

Version 1.3, April 2016

Matthew Naylor and Simon Moore,
University of Cambridge Computer Laboratory

Contents

1 Introduction **2**
1.1 Problem definition 3
1.2 Background 3

2 Trace format **4**

3 Command-line usage **7**

4 SPARC models **11**
4.1 Sequential Consistency (SC) 11
4.2 Total Store Order (TSO) 12
4.3 Partial Store Order (PSO) 14
4.4 Weak Memory Order (WMO) 15
4.5 Axiomatic definitions 18
4.6 Checking algorithm 19

5 POWER model (POW) **23**
5.1 Checking algorithm 27

6 Performance **27**

7 Correctness **29**

References **30**

Appendix A: Example applications **31**
Berkeley's Rocket Chip 31

Appendix B: Litmus test results **37**

	Model	**Reference**
	SC	Sequential Consistency [3]
$\subset$	TSO	Total Store Order [4]
$\subset$	PSO	Partial Store Order [4]
$\subset$	WMO[†]	Weak Memory Order [4]
$\subset$	POW	POWER model [5]

Table 1: Total order of supported memory consistency models.

1 Introduction

Axe is a tool that aids automatic black-box testing of the memory subsystems found in modern multi-core processors. Given a trace containing a set of top-level memory requests and responses, Axe determines if the trace is valid according to a range of *memory consistency models*. It is designed to be used as the oracle in an automated hardware test framework, quickly checking large memory traces that result from mechanically-generated sequences of memory operations [1]. Despite the large amount of non-determinism present in memory consistency models, Axe can handle large traces involving many cores.

Axe supports a spectrum of five consistency models listed in Table 1, each one permitting a subset of the behaviours allowed by the next. We validate Axe by testing equivalence against simpler, non-optimised trace-checkers for the same models. Axe is available from `http://www.github.com/CTSRD-CHERI/axe`. The name "Axe" comes from the use of axiomatic rules to decide the validity of traces.

Example applications of Axe to open-source memory-subsystem implementations are available in Appendix A.

[†]WMO is equivalent to SPARC RMO [4] except that it forbids reordering of loads to the same address, making it a subset of modern relaxed models such as POWER [5].

1.1 Problem definition

Given a trace containing a set of memory requests and responses (including loads, stores, atomic read-modify-writes, memory barriers, and optional timestamps) initiated by concurrent processor cores (or "hardware threads"), Axe determines if the trace satisfies one of the consistency models listed in Table 1. We define the memory trace format in §2 and present the semantics of each consistency model in §4 and §5.

Following Gibbons [7] and Manovit [8], we assume that the address-value pair of every store in a trace is unique, i.e. the same value is never written to the same address twice. This reduces the amount of non-determinism in a model as it allows the store read by any load to be uniquely identified. The restriction is easily met 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. But it does mean that our tool cannot be used for checking memory traces that arise during execution of arbitrary software applications, which are unlikely to meet this restriction.

Another technique for reducing non-determinism is to modify the hardware to emit extra trace information such as the order in which writes reach a particular internal merge point. However, for now we treat the memory subsystem as a *black box* and do not inspect or modify its internals in any way: we would like our tool to be as easy as possible to use, i.e. not requiring modifications to the system under test.

1.2 Background

Axe is heavily inspired by the work of Manovit et al. [8, 9, 10, 11] and their *TSOtool* [12]. TSOtool generates pseudo-random multi-threaded programs, runs them on a multiprocessor, and compares the results against the *Total Store Order* specification (TSO) to reveal potential discrepancies. A key contribution of the work is a state-of-the-art conformance checking algorithm for TSO that can handle long-running programs – on the order of millions of memory operations and hundreds of cores – *despite* this being an NP-complete problem. TSOtool, and variants of it, have been used with success at Sun Microsystems [8] and Intel [13]. Unfortunately, "TSOTool is a proprietary program of Sun Microsystems" [12], and only supports the

TSO model. Our checking algorithm for the SPARC models (SC, TSO, PSO, WMO) is a generalisation of TSOtool's algorithm; it has a worse time and space complexity but still performs very well in practice.

Our checking algorithm for the POWER model (POW) is inspired by the operational semantics of Sarkar et al. [5] and the associated PPCMEM tool [15], but offers performance sufficient for specification-based testing purposes.

2 Trace format

We introduce the syntax of memory-subsystem traces by way of example.

Example 1 Here is a simple trace consisting of five operations running on two threads.

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

The number before the `:` denotes the thread id. 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`.

The initial value of every memory location is implicitly `0`. For any read of a value other than `0`, there must exist a write of that value to the same address in the trace, otherwise the trace is said to be malformed. As mentioned in §1.1, we also require that the address-value pair of every write is unique.

The textual order of operations with the same thread id is the order in which those operations were issued to the memory subsystem by that thread. We refer to this order as the *thread-order*. No ordering is implied by the textual order of operations with different thread ids. In the above example, the write by thread `1` is not ordered in any way with respect to any of the operations by thread `0`, but it is thread-order-before the read by thread `1`.

Example 2 Here is another trace, illustrating timestamps.

```
0: M[0] := 1
0: sync
0: M[1] := 1
1: M[1] == 1    @ 100 : 110
1: M[0] == 0    @ 115 :
```

The operation on the fourth line contains a begin-time of `100` and an end-time of `110`, denoting the times at which the request was submitted and the response received respectively. Operations that perhaps do not generate a response, such as a store, can simply leave the end-time unspecified*. In fact, all timestamps are completely optional, for a few reasons:

- some consistency models are unaffected by timestamps;
- timestamps may not be available, depending on how the traces are produced; and
- example traces are easier to read if only the interesting or relevant timestamp information is supplied.

In some consistency models however, timestamps can affect whether or not a trace is allowed. In the above example, the timestamps indicate that first load by thread `1` must have finished before the second load by thread `1` begins, implying that the memory subsystem could not have executed the operations out-of-order. In the SPARC and POWER architectures, a programmer can arrange such a dependency by having the address of the second load be dependent on the result of the first load – a so-called *address dependency* [5]. Other kinds of dependency include *data dependencies* (where the

*Axe currently disallows end-times for store operations, making it clear that this information is not used.

value of a store is dependent on the result of a preceding load) and *control dependencies* (where an operation is control-flow dependent on the result of preceding load). These program-level dependencies become observable in the memory trace as end-time-before-begin-time dependencies.

By default, we consider timestamps to be *local to each thread*, i.e. we do not use timestamps to infer ordering between operations that run on different threads. This means we can test hardware in which the threads are running in separate clock domains, for example. However, if the `-g` command line flag is specified then Axe can assume a global clock, and compare timestamps of operations running on different threads. Currently, we only exploit the `-g` flag in our POW model (§5);

There is no explicit support in Axe for *cancelled operations* which often arise in modern CPUs due to speculative execution or exceptions. Traces containing such operations can still be checked using Axe by simply replacing them with no-ops. There is also no support for *mixed-width* accesses at present: Axe abstracts over the width of each memory location and hence the width may vary between traces, but *not within a trace.*

Example 3 Here is third trace, this time containing three operations, the first of which is an atomic read-modify-write operation.

```
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, otherwise the trace is malformed. A common way of expressing atomic operations in RISC instruction sets is via a pair of *load-linked* and *store-conditional* operations. At the trace level, it is straightforward to convert such a pair of operations into a single read-modify-write operation:

- 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 operation.

For read-modify-write operations, the end-time simply denotes the time at which the read-response is received. Currently, Axe has no concept of an end-time on a write operation.

Example 4 The following trace illustrates `final` constraints.

```
0: M[0] := 1
0: M[1] := 1
1: M[1] := 2
1: M[0] == 0
final M[1] == 2
```

The `final` line states that final value at location `1` viewed by all threads after all operations have completed is `2`. These `final` constraints are entirely optional and are primarily supported so that litmus tests (used for testing Axe) can be neatly expressed as traces.

3 Command-line usage

Axe can be invoked as follows:

```
  axe check <MODEL> <FILE> [-g] [-i]
```

where `<MODEL>` is `SC`, `TSO`, `PSO`, `WMO`, or `POW`; `<FILE>` is the name of a file containing a trace or "`-`" to read a trace from standard input. The optional `-g` flag (currently only used in the `POW` model) indicates that a global clock domain may be assumed (see §2 for more details). The optional `-i` flag indicates that timestamps in the trace should be ignored.

The output is either "`OK`", denoting that the trace is allowed by the specified model, or "`NO`" if it is forbidden. If the trace is malformed or does not meet the necessary constraints, an error message will be reported. To illustrate, if the file `trace.axe` contains:

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

then the command

```
axe check TSO trace.axe
```

will output “`OK`”.

Multiple traces per file

Axe allows a single file to contain multiple traces, with each trace terminated by a line containing the text “`check`”. It also allow comments (lines beginning with the character “`#`”) in trace files. To illustrate, if the file `traces.axe` contains:

```
# Trace 1
0: M[1] := 1
0: M[0] == 0
1: M[0] := 1
1: M[1] == 0
check

# Trace 2
0: M[0] := 1
0: M[1] := 1
1: M[1] == 1
1: M[0] == 0
check
```

then the command

```
axe check TSO traces.txt
```

will output:

```
OK
NO
```

That is, one decision per trace, in order.

Interaction

It is straightforward to connect Axe to other tools such as HDL simulators: any program can simply `popen()` Axe, specifying the input file as "`-`", and communicate with it via pipes.

Testing

Axe also supports the invocation pattern:

```
axe test <MODEL> <FILE> <FILE> [-g] [-i]
```

where the arguments are the same as before, except for the introduction of the second `<FILE>` argument which specifies a file of expected outcomes (i.e. "`OK`" or "`NO`"), one for each trace in the trace file. Axe reports an error if any trace does not give the expected outcome. The purpose of this mode is to support testing of Axe itself. There are a large number of tests and expected outcomes in the "`tests`" subdirectory of the Axe distribution.

Shrinking traces

When a trace fails a given consistency model, Axe simply reports back "`NO`". This is not particularly helpful in understanding *why* a trace violates a model, especially when the trace is long. In such cases, the `axe-shrink.py` script (included in the `src` subdirectory) can be used to search for a small subset of the trace that fails the model. To illustrate, suppose the file `failure.axe` contains a 260-line trace that fails the TSO model. Running

```
  axe-shrink.py TSO failure.axe
```

might give:

```
  Pass 0
  Omitted 241 of 260
  Pass 1
  Omitted 256 of 260
  Pass 2
  Omitted 256 of 260
  0: M[2] := 46 @ 497:
  1: M[2] == 46 @ 280:513
  1: M[2] := 61 @ 729:
  1: M[2] == 46 @ 854:979
```

The 260-line trace has been reduced to 4 lines. This simple shrinker tries, in reverse trace order, to drop each operation in turn and this is repeated until a fixed-point is reached.

For large traces, we have developed `axe-big-shrink.py`, which works by applying each of the following rewrite rules for *retry* attempts before moving on to the next rule. Each rule is conditioned on the resulting trace still violating the model.

1. pick an address and drop all accesses to that address;
2. drop a random subset of n loads;
3. drop a random subset of n stores which write a value that is never read;
4. same as (3) except for read-modify-write operations.

After that, it resorts back to the simple shrinking algorithm. For suitable choices of *retry* and n, it is both effective and fast.

4 SPARC models

This section presents an operational semantics for the SC, TSO, PSO and WMO models supported by Axe. The common feature of these models is the existence (or illusion) of a *single shared memory*: if a write by one thread is observed by another then it must be observable to all threads. Sometimes known as *multi-copy atomicity* or *global store atomicity*, this property is typically provided by hardware that implements a single-writer cache coherence protocol.

We define the behaviours allowed by each 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 operations in the format given in §2).
- A mapping M from memory addresses to values.
- A mapping B from thread ids to sequences of operations. 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 each address a, and $B(t) = []$ for each thread 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, where M satisfies all the `final` constraints in this final state, then we say that the machine accepts the initial trace and that the trace T is allowed by the model. Otherwise, it is disallowed by the model.

4.1 Sequential Consistency (SC)

SC has just one state-transition rule.

Rule 1 Pick a thread t non-deterministically. Remove the first operation 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 under the chosen values for the non-deterministic variables. In this case t is the only non-deterministic variable.

4.2 Total Store Order (TSO)

In TSO, each thread has a local store buffer. Before presenting the semantics, we give a few examples of TSO behaviour.

Example (SB) Here is a sample trace that is allowed by TSO but forbidden by SC.

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

There are six possible interleavings of each thread's operations but none result in both reads returning zero. However, under TSO [4], stores may be buffered locally by a thread, allowing subsequent loads to complete before the buffered stores can be observed by other threads.

Example (SB+syncs) Under TSO, the above behaviour can be prevented by inserting `sync` operations that cause the local buffers to be flushed. The following trace is forbidden.

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

In general, placing a `sync` between every pair of thread-ordered operations restores SC behaviour.

Example (SB+RMWs) Another way to prevent the relaxed behaviour in the SB example is to replace each write with an atomic read-modify-write. The following trace is forbidden.

```
0: { M[1] == 0; M[1] := 1 }
0: M[0] == 0
1: { M[0] == 0; M[0] := 1 }
1: M[1] == 0
```

Under TSO, read-modify-write has the side-effect of flushing the store buffer.

Operational Semantics

We define TSO 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 evicting elements from the buffers to memory.

Rule 1 Pick a thread t non-deterministically. Remove the first operation 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 operation 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 non-deterministically. Remove the first operation `M[`a`]:=` v from $B(t)$ and update $M(a)$ to v.

4.3 Partial Store Order (PSO)

PSO is similar to TSO but relaxes the order in which writes can be evicted from the buffer. In particular: writes to different addresses can be evicted out-of-order.

Example (MP) The following trace is allowed by PSO but forbidden by TSO.

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

The writes may be evicted *out-of-order* from the local buffer on thread 0, allowing thread 1 to see the second write before it sees the first.

Example (MP+sync+po) The above relaxed behaviour can be prevented by inserting a `sync` between the writes. The following trace is forbidden by PSO.

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

Example (MP+RMW) Under TSO, atomic operations have the side-effect of flushing the local write buffer. Under PSO, only writes to the same address are flushed, hence the following trace is allowed under PSO.

```
0: M[0] := 1
0: { M[1] == 0; M[1] := 1 }
1: M[1] == 1
1: M[0] == 0
```

Operational Semantics

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 operation 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 Non-deterministically pick a thread t and an address a. Remove the first operation that refers to address a, `M[`a`]:=` v, from $B(t)$ and update $M(a)$ to v.

4.4 Weak Memory Order (WMO)

WMO is a relaxation of PSO in which load operations, like stores, become non-blocking. Unlike SPARC's RMO, it forbids reordering of loads to the same address, making it a subset of modern relaxed models such as POWER. In all other respects, it is equivalent to RMO.

Example (MP+sync+po resisted) This example, forbidden by PSO, is allowed by WMO because while the writes must occur in order, the loads (to different addresses) may happen out-of-order.

Example (MP+syncs) If a sync is also placed between the two loads as follows, then the relaxed behaviour becomes forbidden.

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

Example (MP+sync+dep) Alternatively, a dependency between the two loads, in the form of a "begin-time after an end-time" may also be used to keep the loads in order. The following trace is forbidden by WMO.

```
0: M[0] := 1
0: sync
0: M[1] := 1
1: M[1] == 1   @ 100:110
1: M[0] == 0   @ 115:
```

The fact that the second load begins after the first one completes is enough, under WMO, to imply that the memory subsystem cannot reorder them.

Example (LB) The MP+sync example demonstrates reordering of two loads, but WMO also allows reordering of a load followed by a store (when the addresses are different). The following trace is allowed by WMO.

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

Example (LB+syncs & LB+addrs) As expected, a `sync` after each load will prevent the LB behaviour. So too will a timestamp dependency between each load and store.

Operational Semantics

Rule 1 Pick a thread t non-deterministically. Remove the first operation i executed by t from the trace.

1. If $i = \texttt{sync}$ and $B(t) = []$ then succeed;
2. Otherwise: **fail**.

Rule 2 Non-deterministically pick a thread t and an address a. From the trace, remove the first operation i on thread t that satisfies the condition: (a) $i = \texttt{sync}$; or (b) i accesses address a and no operation that precedes i in thread-order has an end-time that precedes the begin-time of i.

1. If $i = \texttt{sync}$ then **fail**.
2. If $i = \texttt{M[}a\texttt{]:=}\ v$ then append i to $B(t)$.
3. If $i = \texttt{M[}a\texttt{]==}\ v$ then let j be the latest operation of the form $\texttt{M[}a\texttt{]:=}\ w$ 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**;
4. If $i = \texttt{<M[}a\texttt{]==}\ v_0\texttt{;}\ \texttt{M[}a\texttt{]:=}\ v_1\texttt{>}$ 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 3 Non-deterministically pick a thread t and an address a. Remove the first operation that refers to address a, $\texttt{M[}a\texttt{]:=}\ v$, from $B(t)$ and update $M(a)$ to v.

4.5 Axiomatic definitions

This section presents the axiomatic definitions of the SC, TSO, PSO and WMO models upon which the Axe checking algorithm for these models is based. In this section, we consider a read-modify-write operation to be both a "load" and a "store".

To begin, it is helpful to distinguish between two different orderings over operations in the trace:

- *Thread Order*: for any given thread, the textual order of operations in the trace issued by that thread.
- *Memory Order*: a total order over all operations in the trace.

All valid traces under these models must satisfy the following property (**value axiom**): the value returned by a load from address a equals the value of the latest store (in memory order) from the set $\mathit{Local} \cup \mathit{Global}$ where Local is the set of stores to address a that precede the load in *thread order* and Global is the set of stores to address a that precede the load in *memory order*.

Depending on the model, the following **local axioms** on operations i and j from the same thread must also be satisfied.

Sequential Consistency (SC)

If i precedes j in thread-order then i must precede j in memory order.

Total Store Order (TSO)

If i precedes j in thread-order then i must precede j in memory order **when**:

- i is a load; or
- i and j are stores; or
- i is a `sync` or j is a `sync`.

Partial Store Order (PSO)

If i precedes j in thread-order then i must precede j in memory order **when**:

- i is a load; or
- i and j are stores *to the same address*; or
- i is a `sync` or j is a `sync`.

Weak Memory Order (WMO)

If i precedes j in thread-order then i must precede j in memory order **when**:

- i is a load and j accesses the same address; or
- i and j are stores to the same address; or
- i is a `sync` or j is a `sync`; or
- i is a load with end-time t_0 and j has begin-time t_1 and $t_0 < t_1$.

4.6 Checking algorithm

In this section, we generalise an algorithm by Manovit [8] for checking traces against the TSO model to support the SC, TSO, PSO *and* WMO models. The central data structure used by this algorithm is the *analysis graph* in which each node denotes an operation from the trace, and each edge denotes that the source node precedes the destination node in memory order.

Simple algorithm

Starting with an empty analysis graph, a simple checking algorithm is as follows.

1. Add each operation in the trace as a node to the analysis graph.

2. Add the edges implied by the local axioms defined above.

3. Apply the two edge-introduction rules shown in Figure 1 to the graph.

4. Add an edge from each read `M[`x`] == 0` to the first store `M[`x`] :=` a on each thread. This ensures that any read of zero (initial value) from address x must happen before any writes to address x.

5. Apply a standard topological sort procedure to the analysis graph with the following tweak: every time a store operation `M[`x`] :=` a is removed from the graph, add an edge from each load `M[`x`] ==` a to the next unpicked store `M[`x`] :=` b on each thread. This ensures that any read of the current value at address x must happen before any store of another value to address x.

6. If a topological sort can be found, i.e. a total order of operations exists that satisfies the memory order constraints, then the trace is valid, otherwise it is invalid.

The key inefficiency of this algorithm is the non-determinism present in the topological sort. At any stage, there may exist several store operations that can be removed next. If a bad choice is made, the algorithm must backtrack since an alternative choice may lead to success. (The order of stores to each address is not known in advance.)

Reducing non-determinism

Manovit proposes the two rules shown in Figure 2 as a way of inferring new edges in the analysis graph, greatly reducing the amount of non-determinism in the topological sort. Notice that applying these rules can introduce edges which enable the rules to be applied again. Therefore it is desirable to apply the rules repeatedly until a fixed-point is reached, i.e. until no new edges are inferred.

This leads to two modifications of the simple algorithm above: first, add a new step after step (3) that applies the inference rules until a fixed-point; second, every time a store is removed from the graph in step (5), and new edges are added, reapply the inference rules until a fixed-point is reached.

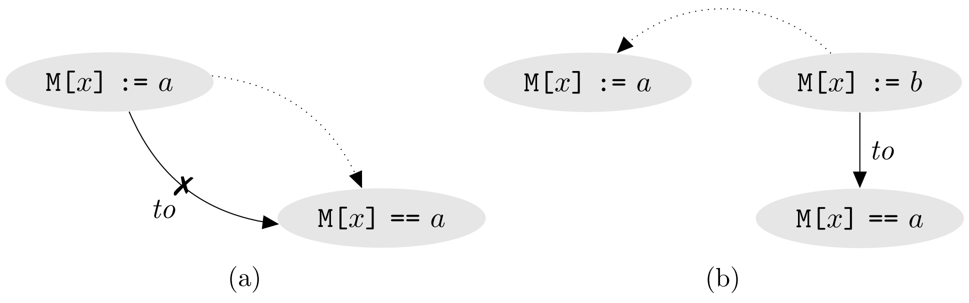

Figure 1: Edge-introduction rules. In (a) the dotted memory-order edge is introduced if the solid thread-order edge, labelled *to*, does not exist. In (b) the dotted memory-order edge is introduced if the thread-order edge, labelled *to*, exists.

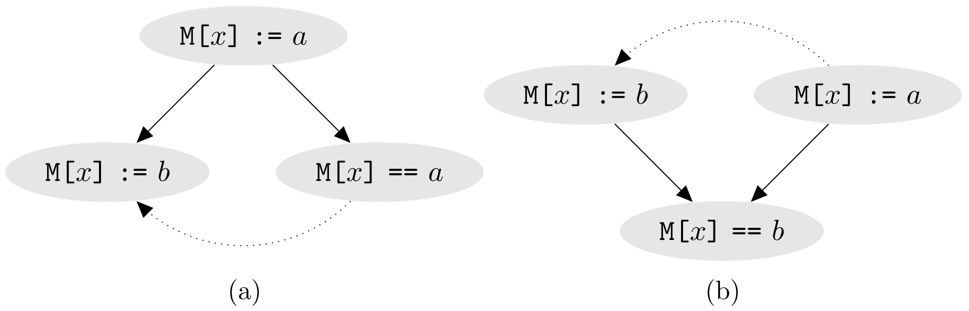

Figure 2: Edge-inference rules proposed by Manovit [8] (our representation). In each case, if the solid memory-order edges are known to exist, either directly or by transitivity, then the dotted memory-order edge can be inferred.

Reducing rule-application sites

Applying the inference rules at all matching sites in the analysis graph would be extremely inefficient and, fortunately, unnecessary. Instead, it is sufficient to apply each rule once for each store s of the form `M[`x`] :=` a with:

- for rule 2a, node `M[`x`] :=` b bound to the earliest store to address x that succeeds s in the analysis graph;
- for rule 2b, node `M[`x`] ==` b bound to the earliest load to address x that succeeds s in the analysis graph.

While there may exist several bindings that satisfy the above constraints (the earliest successor may not be unique in a partial order), the number of application sites to consider is greatly reduced.

Determining the earliest successors

The problem now is: starting from any store operation, how do we efficiently determine the next load and store to the same address in the analysis graph?

To answer this, we maintain two data structures. The first is the mapping $nextLoad(op, t, a)$ that gives the next *load* (in the analysis graph) to address a on thread t from operation op. (Since loads to the same address on a given thread are totally ordered under all models, this mapping is a function, i.e. unambiguous.) Initially, it is computed by a backward analysis, propagating the next load for each (a, t) pair backwards along the edges of the analysis graph, in reverse-topological order. At a fork point, the information at several nodes is merged by taking the minimum load in thread order for each (a, t) pair. When a new edge $i \rightarrow j$ is added to the graph, the $nextLoad$ mapping is updated by applying the same propagation method backwards from node j until no new updates to the mapping are made.

The second data structure we maintain is the mapping $nextStore$, identical to $nextLoad$ but giving the next store instead of the next load. These two data structures have a number of uses:

- the inference rules from Figure 2 can be efficiently applied;

- the existence of a path from a store to any load or store can be determined in constant-time, avoiding the addition of redundant edges to the graph;
- similarly, we can be determine in constant-time whether or not the addition of an edge to the graph will lead to a cycle, allowing immediate failure detection;
- when adding an edge to the graph, the backwards propagation method used to update the *nextLoad* and *nextStore* mappings will naturally visit all the nodes at which the inference rules must be re-applied.

Comparison to Manovit's algorithm

When specialising the algorithm to the TSO model, it is possible to simplify the *nextLoad* and *nextStore* mappings. Instead of mapping each (op, a, t) triple to the next load or next store, it is sufficient to map each (op, t) pair. This is because all loads by the same thread are totally ordered under TSO, and so too are all stores by the same thread. Once the next load on some thread is determined, the next load to a particular address on that thread can be easily found by looking at the static thread order. Consequently, the size of these data structures reduces from $2 \times N \times A \times T$ for N operations, A addresses, and T threads to $2 \times N \times T$. Not only does this save space, but it makes the backwards analysis faster as the amount of information being propagated is smaller. In other words, the efficiency of our checker depends on the number of different address locations used in the trace. This is not the case for Manovit's TSO-only checker.

5 POWER model (POW)

The key relaxation introduced by the POW model is to allow writes to be observed by some threads before they can be observed by others (known as "non-multi-copy-atomicity" [5]). This supports (though is not necessary for) cache hierarchies involving more than one shared cache. Before presenting the model, we look at a few examples demonstrating this relaxed behaviour.

Example (WRC+deps) The following trace is allowed by POW but forbidden by WMO.

```
0: M[0] := 1
1: M[0] == 1    @ 100:110
1: M[1] := 1    @ 115
2: M[1] == 1    @ 200:210
2: M[0] == 0    @ 215
```

Notice the dependencies prevent any local reordering of operations, which is why the trace is forbidden by WMO. This is an example of non-multi-copy-atomic behaviour: the write by thread 0 propagates to thread 1 before it propagates to thread 2. At the hardware level, this could be explained by the presence of a cache shared by threads 0 and 1 but not 2.

Example (WRC+sync+dep) A single `sync` operation, inserted as follows, is enough to forbid the relaxed behaviour.

```
0: M[0] := 1
1: M[0] == 1
1: sync
1: M[1] := 1
2: M[1] == 1    @ 200:210
2: M[0] == 0    @ 215
```

This demonstrates the so-called "cumulative" property of `sync` [5]: not only does `sync` ensure that all preceding writes by the issuing thread have propagated to all other threads, but it also ensures that any writes that the issuing thread has observed before the `sync` have also propagated. In this example, any thread that sees the write by thread 1 must also see the write by thread 0 because thread 1's write is preceded by a `sync` which is in turn preceded by an observation of thread 0's write.

Example (SB+syncs and MP+sync+dep revisited) Under POW, these examples are still forbidden: the `sync` instructions are still are enough to forbid the relaxed behaviour.

Example (WWC+deps) The following trace is allowed by POW but forbidden by WMO.

```
0: M[0] := 1
1: M[0] == 1 @ 100:110
1: M[1] := 1 @ 115:
2: M[1] == 1 @ 200:210
2: M[0] := 2 @ 215:
final M[0] == 1
```

At the hardware level, this example can again be be explained by the presence of a cache, say C, shared by threads 0 and 1 but not 2: (1) the write by thread 0 can reach C and be observed by thread 1; (2) C can evict the write by thread 1 before the write by thread 0; (3) thread 2 can observe the write of thread 1 in the last-level cache; (4) the write by thread 2 can reach the last-level cache; and (5) C can evict the write by thread 0 and overwrite thread 2's write in the last-level cache.

Axiomatic definition

We first present a semantics that ignores atomic read-modify-write operations, and then extend the semantics to support them. The semantics is defined by the combination of two kinds of order:

- For each address a, a value order $<_a$ over values written to address a.
- An operation order $\prec$ over operations in the trace.

Let $val(op)$, where op is a load or store, denote the value read or written by op.

Let $rf(op)$, where op is a load, denote the store operation in the trace with the same (address,value) pair as op.

If op is the first operation to address a on some thread and $val(op) \neq 0$ then add constraint $0 <_a val(op)$.

If op_1 and op_2 are thread-ordered operations that access address a and $val(op_1) \neq val(op_2)$ then add constraint $val(op_1) <_a val(op_2)$.

Let op_1 and op_2 be thread-ordered operations. Add constraint $op_1 \prec op_2$ if:

- op_1 is a load and op_2 accesses the same address; or
- op_1 and op_2 are stores to the same address; or
- op_1 is a `sync` or op_2 is a `sync`; or
- op_1 is a load with end-time t_1 and op_2 has begin-time t_2 and $t_1 < t_2$.

(Note: these are the same as the local axioms for the WMO model.)

For each load op, add constraint $rf(op) \prec op$.

Let op_1 and op_2 be any two distinct `sync` operations from the trace. Add constraint $x \prec y \vee y \prec x$, i.e. a total-order over syncs.

Let op_1 and op_2 be two distinct `sync` operations from the trace. For each address a, add constraint $op_1 \prec op_2 \Rightarrow v <_a w$ if $v \neq w$ where:

- v is the latest value seen at address a before op_1 in thread-order; and
- w is the earliest value seen at address a after op_2 in thread-order.

Let op_1 be a `sync` and op_2 be a load with response-time t. Let op_3 be the first operation with a request-time larger then t that follows op_2 in thread order. For each address a, add constraint $op_1 \prec op_2 \Rightarrow v <_a w$ if $v \neq w$ where:

- v is the latest value seen at address a before op_1 in thread-order; and
- w is the earliest value seen at address a at or after after op_3 in thread-order.

If there is a solution to all the above constraints, then the trace conforms to the POW model, otherwise it doesn't.

Atomic operations

Atomic read-modify-write operations are treated simply as adjacent thread-ordered read and write operations in the above semantics, provided the following condition is met: for each address a there must exist a topological sort of $<_a$ in which v and w are adjacent for each read-modify-write operation `<M[`a`]==` v`; M[`a`]:=` w`>`.

5.1 Checking algorithm

Our checking algorithm for POW is essentially a solver for the above constraint set. When the `-g` option is specified (indicating a global clock domain), we infer an additional ordering between any two `sync` operations running on different threads if one has a response time that precedes the request time of the other. This can significantly reduce the amount of non-determinism in the solver.

6 Performance

For performance evaluation, we have generated a range of traces[†] with various numbers of memory operations ($n \in \{8K, 16K, 24K, 32K\}$), threads ($t \in \{4, 16, 32\}$), and addresses ($a \in \{4, 16, 32\}$). For each combination of parameters, we generate 16 traces, giving 576 traces for each supported model.

Figures 3, 4, and 5 show how the performances of the TSO, WMO and POW checkers vary with the number of operations and threads present, averaged over the number of addresses present. In the case of the POW checker, we use the `-g` flag, specifying a global clock domain and enabling the use of begin and end times to infer a partial global ordering of `sync` operations. Without the `-g` flag, the POW checker has a limited completion rate: for 4, 16 and 32 threads respectively, it has a completion rate of 100%, 96%, and 54%.

[†]Using a model cache implementation with load and store buffering, out-of-order eviction, out-of-order responses, prefetching and invalidation-based coherence.

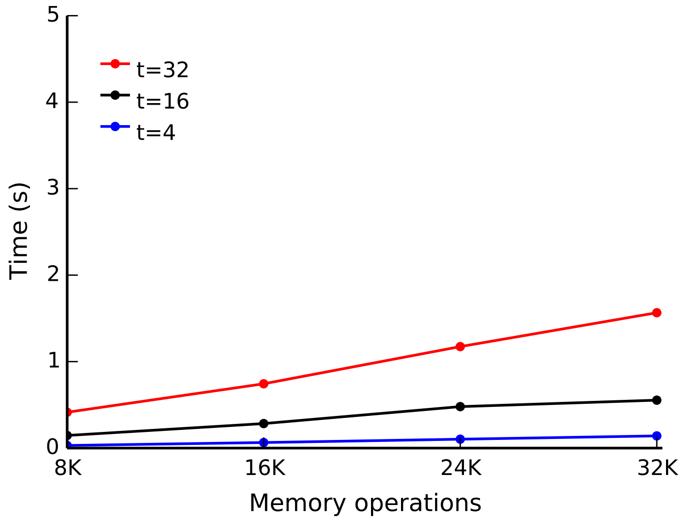

Figure 3: Performance of the TSO checker.

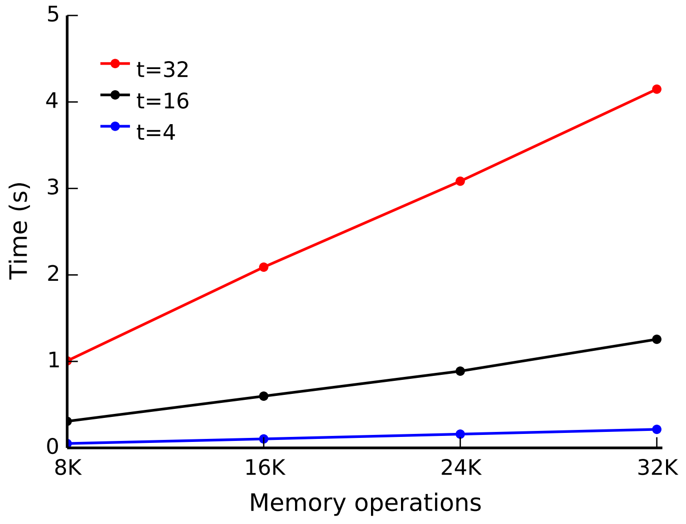

Figure 4: Performance of the WMO checker.

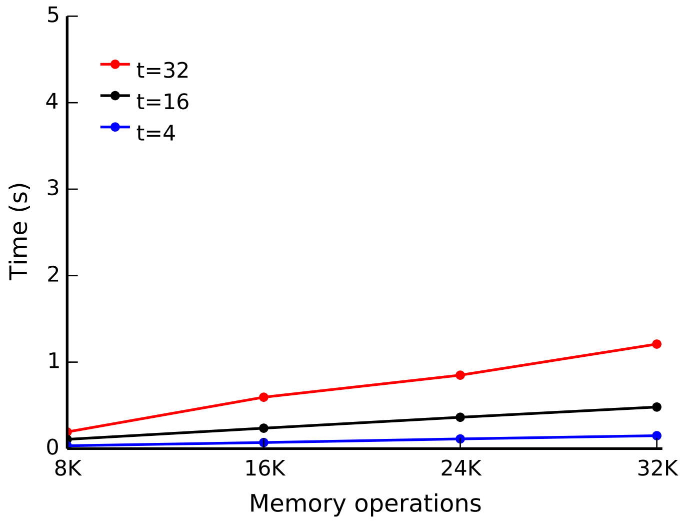

Figure 5: Performance of the POW checker with the `-g` flag specified.

7 Correctness

Axe has been tested for equivalence against an operational semantics for each model and also an axiomatic semantics for each model (§4 and §5). The test traces include: (1) 199 litmus tests from the PPCMEM distribution [15]; and (2) 200K randomly-generated traces ranging from around 10 to 50 operations in size. For the litmus tests, Axe's POW checker gives the same outcome as PPCMEM. Axe also gives the expected outcomes for all the traces used in our performance evaluation (§6).

The 199 litmus test traces and the 200,000 randomly generated traces are both distributed with the Axe tool (in the `tests` sub-directory), along with the expected outcomes of each trace on each model. In addition, the outcomes of the litmus test traces are listed in Appendix B.

Acknowledgements

Thanks to members of the Semantics group at the University of Cambridge Computer Laboratory for numerous clarifications about relaxed memory models. This work was supported by DARPA/AFRL contracts FA8750-10-C-0237 (CTSRD) and FA8750-11-C-0249 (MRC2), and EPSRC grant EP/K008528/1 (REMS). The views, opinions, and/or findings contained in this manual are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Department of Defense or the U.S. Government.

References

[1] M. Naylor and S. W. Moore, *A generic synthesisable test bench*, in MEMOCODE 2015, pp. 128–137.

[2] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, *Litmus: Running Tests Against Hardware*, in TACAS 2011, pp. 41–44.

[3] L. Lamport. *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs*, IEEE Transactions on Computers, volume 28, number 9, pp. 690–691, 1979.

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

[5] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. *Understanding POWER Multiprocessors*, SIGPLAN Notices, vol. 46, num. 6, pp. 175–186, June 2011.

[6] *Homepage of the CHERI processor (Capability Hardware Enhanced RISC Instructions)*, `http://chericpu.org`.

[7] P. B. Gibbons and E. Korach. *On testing cache-coherent shared memories*, in SPAA 1994, pp. 177-188.

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

[9] S. Hangal, D. Vahia, C. Manovit, and JY. J. Lu, *TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model*, in ISCA 2004, pp. 114.

[10] C. Manovit and S. Hangal, *Efficient algorithms for verifying memory consistency*, in SPAA 2005, pp. 245–252.

[11] C. Manovit and S. Hangal, *Completely verifying memory consistency of test program executions*, in HPCA 2006, pp. 166–175.

[12] *Homepage of TSOTool*, a program for verifying memory systems using the memory consistency model, `http://xenon.stanford.edu/~hangal/tsotool.html`.

[13] A. Roy, S. Zeisset, C. J. Fleckenstein, J. C. Huang, *Fast and Generalized Polynomial Time Memory Consistency Verification*, CAV 2006, pp. 503.

[14] S. Park and D. L. Dill. *An executable specification, analyzer and verifier for RMO (relaxed memory order)*, in SPAA 1995.

[15] *Homepage of PPCMEM/ARMMEM*, a tool for exploring the POWER and ARM memory models, `https://www.cl.cam.ac.uk/~pes20/ppcmem/`.

[16] K. Asanovic et al., *The Rocket Chip Generator*, Technical Report UCB/EECS-2016-17, University of California, Berkeley, 2016.

Appendix A: Example applications

Berkeley's Rocket Chip

Rocket Chip is an open-source system-on-chip generator developed at UC Berkeley including support for multiple processor cores and a cache-coherent shared memory subsystem. Available processor cores include the in-order Rocket, the out-of-order BOOM, the Z-scale microcontroller, and (pending release) the Hwacha vector-thread accelerator. Having been taped out 11 times between 2011 and 2015, Rocket Chip is fairly mature but faces constant change through extensions, redesigns, and refactorings. Rocket Chip is written using the Chisel HDL, also from UC Berkeley.

The Rocket Chip developers have recognised the importance of making HDL-level test benches for the memory subsystem: *"In order to test behaviors in our memory hierarchy which are not easy or efficient to test in software, we have designed a set of test circuits called GroundTest"* [16].

GroundTest plugs into the socket given to CPU tiles and generates various kinds of memory traffic directly to the memory subsystem, either via the L1 caches, or directly to the L2, or via DMA.

Rocket Chip is highly parameterised, and this includes the choice of coherence protocol which by default (at the time of writing) is MESI. Since MESI guarantees either a single writer or multiple readers to a cache line at any time, it gives the illusion of a single shared memory – despite the reality of multiple local caches – and is thus expected to conform to one of the SPARC consistency models.

Extending GroundTest We developed a *trace generator* that plugs into the GroundTest framework. Given a random seed, it generates random memory requests from each tile, and emits a trace of events. To illustrate, here is an example of generated trace.

```
1: load-req     0x0000000008 #0 @64
1: store-req  5 0x0000100008 #1 @65
1: store-req  7 0x0000000010 #2 @66
0: store-req  2 0x0000000008 #0 @303
0: load-req     0x0000000008 #1 @304
0: store-req  6 0x0000100008 #2 @305
1: resp       0              #0 @96
0: resp       0              #0 @350
0: resp       2              #1 @351
0: load-req     0x0000000010 #3 @353
1: resp       0              #1 @149
1: load-req     0x0000000108 #3 @152
1: resp       0              #3 @184
0: resp       5              #2 @422
0: resp       0              #3 @424
1: resp       0              #2 @226
```

Main syntactical points:

- the first number on each line of the trace is the **thread-id**;
- #n denotes a **request-id** n;

- `@`t denotes a **time** t in clock cycles;
- hex numbers denote **addresses**;
- remaining decimal numbers denote **values** being loaded or stored;
- this trace contains only **loads**, **stores** and **responses**, but the generator also supports **LR/SC** pairs, **atomics**, and **fences**.

Notice that the timestamps are not monotonically increasing. In simulation, the Rocket Chip tiles are brought out of reset sequentially – each tile starts roughly 250 cycles apart. Hence the timestamps are local to each thread, which is not a problem for Axe but may be confusing when reading the trace.

The number of tiles, requests, and addresses used when generating a trace can all be controlled using *compile-time* parameters. Ideally though, the number of requests would be taken as a *simulation-time* parameter, allowing an iterative-deepening strategy in which the trace generator is repeatedly invoked to emit gradually longer traces over time, in the hope of finding simple failures first. Unfortunately, we are not aware of a convenient way to do this in the Chisel HDL; the equivalent of Verilog's `$value$plusargs` or `$fscanf` would have been very helpful here.

Another compile-time parameter that would ideally be a simulation-time parameter is the set of memory addresses used by each tile: varying the address set by simply rerunning the trace generator would make it easier to explore a wide range of address combinations. The problem is that each tile needs to know at least a subset of the addresses being used by other tiles (otherwise the chances of generating shared variables would be very low) but we are not aware of a convenient way for tiles to share simulation-time data in GroundTest. Again, a Verilog-style `$fscanf` function would suffice to overcome this problem: the common address set could be read from a file by each tile.

Finding a clean way to terminate the Rocket Chip simulator on completion of the trace generator also proved to be a challenge. The standard exit strategy is to raise a flag in one of Rocket Chip's CSR registers but that causes the entire simulation to stop even though some tiles may not yet have finished – and there is no convenient way for tiles to inform each other that they have finished. We settled for the following solution: when a tile finishes it emits a special “finished” message in the trace. A wrapper script runs the trace generator, waits until all tiles have finished, and then sends a `SIGTERM` to the simulator (which gracefully handles this signal).

Converting traces to Axe format We made a simple script to convert traces emitted by the trace generator into Axe format. For example, given the above sample trace, this conversion script yields:

```
# &M[2] == 0x0000000010
# &M[0] == 0x0000000008
# &M[3] == 0x0000000108
# &M[1] == 0x0000100008
1: M[0] == 0 @ 64:96
1: M[1] := 5 @ 65:
1: M[2] := 7 @ 66:
0: M[0] := 2 @ 303:
0: M[0] == 2 @ 304:351
0: M[1] := 6 @ 305:
0: M[2] == 0 @ 353:424
1: M[3] == 0 @ 152:184
```

Notice that lines beginning with `#` are treated as comments by Axe: we use these comments to record the mapping between physical addresses and addresses used by Axe.

Testing against the SC model We made a script that repeatedly: (1) generates a trace with a random seed; (2) converts the trace to Axe format; and (3) checks the trace against the chosen consistency model. Running this script, we found a 260-element trace that fails to satisfy sequential consistency. Passing this trace through `axe-shrink.py` (see §3), we get:

```
Pass 0
Omitted 245 of 260
Pass 1
Omitted 255 of 260
Pass 2
```

```
Omitted 255 of 260
1: M[1] := 185 @ 1921:
1: M[0] := 193 @ 1966:
0: M[0] == 193 @ 2207:2245
0: M[1] := 204 @ 2208:
0: M[1] == 185 @ 2209:2269
```

This trace (similar to the MP example) can be explained either by thread 1's stores being performed out-of-order (PSO) or thread 0's loads being performed out-of-order (WMO).

Testing against the PSO model We also found a 261-element trace that violates PSO.

```
Pass 0
Omitted 252 of 261
Pass 1
Omitted 257 of 261
Pass 2
Omitted 257 of 261
0: M[2] == 137 @ 1825:1948
0: M[0] := 154 @ 1886:
1: M[0] == 154 @ 1689:1725
1: M[2] := 137 @ 1690:
```

This trace (similar to the LB example) can be explained by the load and store on thread 0 (or thread 1) being reordered (WMO).

Coherence bug We observed a large number of traces that satisfy the WMO model, but eventually hit this counter-example:

```
Pass 0
Omitted 241 of 260
Pass 1
Omitted 256 of 260
```

```
Pass 2
Omitted 256 of 260
0: M[2] := 46 @ 497:
1: M[2] == 46 @ 280:513
1: M[2] := 61 @ 729:
1: M[2] == 46 @ 854:979
```

Note that the write of `M[2] := 46` by core 0 is the only write of 46 in the entire trace (the trace generator ensures that all write values are unique). Also, the initial value of each location is 0. Therefore, the write `M[2] := 61` by core 1 has seemingly been dropped. This is a coherence violation and undesirable: if the write of 46 to `M[2]` is interpreted as "core 1, a message is available" then core 1 might end up receiving two messages as it effectively sees the write twice. (It sees 46 once on line 2, then it clears that value with a store on line 3, and finally it sees 46 again on line 4).

We reported this issue to the Rocket Chip developers who identified a race condition in the coherence protocol and fixed it within a few days.

Livelock bug For the above testing we only enabled loads and stores in the trace generator. When we enabled generation of LR/SC pairs, we found a lock-up issue in which a store-conditional would never return under some circumstances. We reported this to the Rocket Chip developers who diagnosed the problem as a livelock issue in the coherence protocol.

Store-conditional bug With the livelock issue fixed, we found the following counter-example to WMO:

```
Pass 0
Omitted 217 of 228
Pass 1
Omitted 224 of 228
Pass 2
Omitted 224 of 228
1: M[3] := 31 @ 340:
0: { M[3] == 31; M[3] := 178} @ 745:812
0: { M[3] == 178; M[3] := 198} @ 926:955
```

```
  1: { M[3] == 178; M[3] := 59} @ 759:761
```

Notice that the read-modify-write by thread 1 atomically changes `M[0]` from 178 to 59. Furthermore, the second read-modify-write on thread 0 atomically changes `M[0]` from 178 to 198. Of course, if these operations really were atomic, this behaviour would be impossible. After investigating the raw trace emitted by the trace generator, we noticed this issue arises when a store-conditional is issued before a load-reserve response is received.

We reported this issue to the Rocket Chip developers who identified it as a bug in which a cache line is not marked as dirty when it should.

Testing against the WMO model At the time of writing – with loads, stores, LR/SC pairs, atomics, and fences all being generated – Rocket Chip satisfies the WMO model on thousands of large traces (32k operations per trace, 16 addresses, 8 threads).

Liveness A key limitation of the above specification-based testing approach is that it does not check for liveness, e.g. that a store-conditional operation succeeds when it should. In response, we added a mode to the trace generator in which it will only generate LR/SC pairs that are expected to succeed. This is possible in Rocket Chip because of the way it implements LR/SC: the L1 cache will hold on to a cache line for a maximum of n cycles after an LR response. So provided the LR and SC are within n cycles of each other, the SC should succeed. In this mode, we observed an LR/SC success rate of 94%. The 6% of failures remain unexplained and we plan to explore this in future work.

Appendix B: Litmus test results

The following table gives the output of Axe on a large number of litmus tests from the PPCMEM distribution. These tests can also be found in the Axe distribution in the `tests/litmus` subdirectory. We use a tick mark (✓) to denote that the behaviour described by the test is allowed, and an empty cell to denote that it is forbidden. In each case, our POW model gives the same outcome as PPCMEM.

Test name	**SC**	**TSO**	**PSO**	**WMO**	**POW**
`2+2W+sync+po`			✓	✓	✓
`3.2W`			✓	✓	✓
`3.2W+sync+po+po`			✓	✓	✓
`3.2W+syncs`					
`3.2W+sync+sync+po`			✓	✓	✓
`3.LB+addr+addr+po`				✓	✓
`3.LB+addr+po+po`				✓	✓
`3.LB+addrs`					
`3.LB+addr+sync+po`				✓	✓
`3.LB`				✓	✓
`3.LB+sync+addr+addr`					
`3.LB+sync+addr+po`				✓	✓
`3.LB+sync+po+po`				✓	✓
`3.LB+syncs`					
`3.LB+sync+sync+addr`					
`3.LB+sync+sync+po`				✓	✓
`3.SB`		✓	✓	✓	✓
`3.SB+sync+po+po`		✓	✓	✓	✓
`3.SB+syncs`					
`3.SB+sync+sync+po`		✓	✓	✓	✓
`IRIW+addr+po`				✓	✓
`IRIW+addrs`					✓
`IRIW`				✓	✓
`IRIW+sync+addr`					✓
`IRIW+sync+po`				✓	✓
`IRIW+syncs`					
`IRRWIW+addr+po`				✓	✓
`IRRWIW+addrs`					✓
`IRRWIW+addr+sync`					✓
`IRRWIW`				✓	✓
`IRRWIW+po+addr`				✓	✓
`IRRWIW+po+sync`				✓	✓
`IRRWIW+sync+addr`					✓
`IRRWIW+sync+po`				✓	✓
`IRRWIW+syncs`					
`IRWIW+addr+po`				✓	✓
`IRWIW+addrs`					✓
`IRWIW`				✓	✓

Test name	**SC**	**TSO**	**PSO**	**WMO**	**POW**
`IRWIW+sync+addr`					✓
`IRWIW+sync+po`				✓	✓
`IRWIW+syncs`					
`ISA2+sync+addr+addr`					
`ISA2+sync+addr+po`				✓	✓
`ISA2+sync+addr+sync`					
`ISA2+sync+po+addr`				✓	✓
`ISA2+sync+po+po`				✓	✓
`ISA2+sync+po+sync`				✓	✓
`ISA2+syncs`					
`ISA2+sync+sync+addr`					
`ISA2+sync+sync+po`				✓	✓
`LB+addr+po`				✓	✓
`LB+addrs`					
`LB`				✓	✓
`LB+sync+addr`					
`LB+sync+po`				✓	✓
`LB+syncs`					
`MP`			✓	✓	✓
`MP+po+addr`			✓	✓	✓
`MP+po+sync`			✓	✓	✓
`MP+sync+addr`					
`MP+sync+po`				✓	✓
`MP+syncs`					
`R`		✓	✓	✓	✓
`R+po+sync`			✓	✓	✓
`R+sync+po`		✓	✓	✓	✓
`R+syncs`					
`RWC+addr+po`		✓	✓	✓	✓
`RWC+addr+sync`					✓
`RWC`		✓	✓	✓	✓
`RWC+po+sync`				✓	✓
`RWC+sync+po`		✓	✓	✓	✓
`RWC+syncs`					
`S`			✓	✓	✓
`SB`		✓	✓	✓	✓
`SB+sync+po`		✓	✓	✓	✓
`SB+syncs`					

Test name	**SC**	**TSO**	**PSO**	**WMO**	**POW**
`S+po+addr`			✓	✓	✓
`S+po+sync`			✓	✓	✓
`S+sync+addr`					
`S+sync+po`				✓	✓
`S+syncs`					
`WRC+addr+po`				✓	✓
`WRC+addrs`					✓
`WRC+addr+sync`					✓
`WRC`				✓	✓
`WRC+po+addr`				✓	✓
`WRC+po+sync`				✓	✓
`WRC+sync+addr`					
`WRC+sync+po`				✓	✓
`WRC+syncs`					
`WRR+2W+addr+po`			✓	✓	✓
`WRR+2W+addr+sync`					✓
`WRR+2W`			✓	✓	✓
`WRR+2W+po+sync`				✓	✓
`WRR+2W+sync+po`			✓	✓	✓
`WRR+2W+syncs`					
`WRW+2W+addr+po`			✓	✓	✓
`WRW+2W+addr+sync`					✓
`WRW+2W`			✓	✓	✓
`WRW+2W+po+sync`				✓	✓
`WRW+2W+sync+po`			✓	✓	✓
`WRW+2W+syncs`					
`W+RWC`		✓	✓	✓	✓
`W+RWC+po+addr+po`		✓	✓	✓	✓
`W+RWC+po+addr+sync`			✓	✓	✓
`W+RWC+po+po+sync`			✓	✓	✓
`W+RWC+po+sync+po`		✓	✓	✓	✓
`W+RWC+po+sync+sync`			✓	✓	✓
`W+RWC+sync+addr+po`		✓	✓	✓	✓
`W+RWC+sync+addr+sync`					
`W+RWC+sync+po+po`		✓	✓	✓	✓
`W+RWC+sync+po+sync`				✓	✓
`W+RWC+syncs`					
`W+RWC+sync+sync+po`		✓	✓	✓	✓

Test name	**SC**	**TSO**	**PSO**	**WMO**	**POW**
`WRW+WR+addr+po`		✓	✓	✓	✓
`WRW+WR+addr+sync`					✓
`WRW+WR`		✓	✓	✓	✓
`WRW+WR+po+sync`				✓	✓
`WRW+WR+sync+po`		✓	✓	✓	✓
`WRW+WR+syncs`					
`WWC+addr+po`				✓	✓
`WWC+addrs`					✓
`WWC+addr+sync`					✓
`WWC`				✓	✓
`WWC+po+addr`				✓	✓
`WWC+po+sync`				✓	✓
`WWC+sync+addr`					
`WWC+sync+po`				✓	✓
`WWC+syncs`					
`Z6.0`		✓	✓	✓	✓
`Z6.0+po+addr+po`		✓	✓	✓	✓
`Z6.0+po+addr+sync`			✓	✓	✓
`Z6.0+po+po+sync`			✓	✓	✓
`Z6.0+po+sync+po`		✓	✓	✓	✓
`Z6.0+po+sync+sync`			✓	✓	✓
`Z6.0+sync+addr+po`		✓	✓	✓	✓
`Z6.0+sync+addr+sync`					
`Z6.0+sync+po+po`		✓	✓	✓	✓
`Z6.0+sync+po+sync`				✓	✓
`Z6.0+syncs`					
`Z6.0+sync+sync+po`		✓	✓	✓	✓
`Z6.1`			✓	✓	✓
`Z6.1+po+po+addr`			✓	✓	✓
`Z6.1+po+po+sync`			✓	✓	✓
`Z6.1+po+sync+addr`			✓	✓	✓
`Z6.1+po+sync+po`			✓	✓	✓
`Z6.1+po+sync+sync`			✓	✓	✓
`Z6.1+sync+po+addr`			✓	✓	✓
`Z6.1+sync+po+po`			✓	✓	✓
`Z6.1+sync+po+sync`			✓	✓	✓
`Z6.1+syncs`					
`Z6.1+sync+sync+addr`					

Test name	SC	TSO	PSO	WMO	POW
Z6.1+sync+sync+po				✓	✓
Z6.2			✓	✓	✓
Z6.2+po+addr+addr			✓	✓	✓
Z6.2+po+addr+po			✓	✓	✓
Z6.2+po+addr+sync			✓	✓	✓
Z6.2+po+po+addr			✓	✓	✓
Z6.2+po+po+sync			✓	✓	✓
Z6.2+po+sync+addr			✓	✓	✓
Z6.2+po+sync+po			✓	✓	✓
Z6.2+po+sync+sync			✓	✓	✓
Z6.2+sync+addr+addr					
Z6.2+sync+addr+po				✓	✓
Z6.2+sync+addr+sync					
Z6.2+sync+po+addr				✓	✓
Z6.2+sync+po+po				✓	✓
Z6.2+sync+po+sync				✓	✓
Z6.2+syncs					
Z6.2+sync+sync+addr					
Z6.2+sync+sync+po				✓	✓
Z6.3			✓	✓	✓
Z6.3+po+po+addr			✓	✓	✓
Z6.3+po+po+sync			✓	✓	✓
Z6.3+po+sync+addr			✓	✓	✓
Z6.3+po+sync+po			✓	✓	✓
Z6.3+po+sync+sync			✓	✓	✓
Z6.3+sync+po+addr			✓	✓	✓
Z6.3+sync+po+po			✓	✓	✓
Z6.3+sync+po+sync			✓	✓	✓
Z6.3+syncs					
Z6.3+sync+sync+addr					
Z6.3+sync+sync+po				✓	✓
Z6.4		✓	✓	✓	✓
Z6.4+po+po+sync		✓	✓	✓	✓
Z6.4+po+sync+po		✓	✓	✓	✓
Z6.4+po+sync+sync			✓	✓	✓
Z6.4+sync+po+po		✓	✓	✓	✓
Z6.4+sync+po+sync		✓	✓	✓	✓
Z6.4+syncs					

Test name	SC	TSO	PSO	WMO	POW
`Z6.4+sync+sync+po`		✓	✓	✓	✓
`Z6.5`		✓	✓	✓	✓
`Z6.5+po+po+sync`			✓	✓	✓
`Z6.5+po+sync+po`		✓	✓	✓	✓
`Z6.5+po+sync+sync`			✓	✓	✓
`Z6.5+sync+po+po`		✓	✓	✓	✓
`Z6.5+sync+po+sync`			✓	✓	✓
`Z6.5+syncs`					
`Z6.5+sync+sync+po`		✓	✓	✓	✓
Count	0	35	89	140	155