# Axe Manual

January 2016

Matthew Naylor and Simon Moore,
University of Cambridge Computer Laboratory

## Contents

| | Model | Expansion | Supports feature |
|---|---|---|---|
| | SC | Sequential Consistency [3] | |
| $\subset$ | TSO | Total Store Order [4] | Store buffering |
| $\subset$ | PSO | Partial Store Order [4] | Out-of-order eviction |
| $\subset$ | WMO[†] | Weak Memory Order [4] | Load buffering and out-of-order responses |
| $\subset$ | POW | POWER model [5] | Nested cache hierarchies and lazy invalidation |

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 randomly-generated sequences of memory operations [1]. It can also assist bug diagnosis by enabling testing strategies that search for small failing cases [1]. Despite the large amount of non-determinism present in memory consistency models, Axe can handle long 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 and supporting a greater number of implementation features.

We validate Axe by various means: (1) by testing it for equivalance against existing models developed by others; (2) by testing equivalance between optimised and non-optimised versions of the same model; and (3) by applying it to traces generated by real and model hardware implementations.

[†]WMO is equivalant 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].

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.)

### 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 give an operational semantics for each of the consistency models 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. This 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 aribtrary 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. The reason for this is that 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 part of our efforts to support testing and debugging of the CHERI processor developed at the University of Cambridge and SRI International [6]. It is heavily inspired by Manovit's *TSOTool* [8, 9] which randomly generates SPARC programs, runs them, and checks the resulting traces for consistency. Apart from operating at the ISA level rather than the HDL

level, this is exactly the approach we had envisaged for testing our own memory subsystem. The performance of TSOTool is impressive, scaling to large traces and many cores. Unfortunatly, "TSOTool is a proprietary program of Sun Microsystems" [9], and only supports the TSO model. Our checking algorithm for the SC, TSO, PSO and WMO models is a fairly minor generalisation of Manovit's algorithm.

Our work is also influenced by the memory model by Sarkar et al. [5] for the IBM POWER architecture, and the associated simulator, PPCMEM [11]. Although PPCMEM is not designed for efficient trace-checking and can take a very long time to terminate even on very small traces (of less than ten instructions on a few threads), this work has greatly helped our understanding of modern relaxed memory models as well as assisting our testing of Axe.

## 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 submitted to the memory subsystem by that thread. Following standard terminology, we refer to this order as *program-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 program-order-before the read by thread `1`.

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

```
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.

**Example 3** Here is a third 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 two reasons:

- some consistency models are unaffected by timestamp information;
- example traces are easier to read if only the interesting or relevant timestamp information is supplied.

In some consistency models however, timestamps do 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 and thus 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* [7]. Other kinds of dependency include *data depedencies* (where the value of a store is dependent on the result of a preceeding load) and *control dependencies* (where an operation is control-flow dependent on the result of preceeding load). All 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*.

*Specifically, we infer an ordering between two `sync` operations running on different threads if the first ends before the second begins.

**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 our models) can be neatly expressed as traces.

# 3 Command-line usage

Axe can be invoked as follows:

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

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

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 straigtforward 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]
```

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.

## 4 SPARC models

This section presents an operational semantics for the SC, TSO, PSO and WMO models supported by Axe. 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 is no interleaving of the operations that results in both reads returning zero. However, under TSO, a write may be buffered locally by a thread, allowing a subsequent load to complete before the write can be seen by another thread.

**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 program-order operations restores SC behaviour.

**Example (SB+RMWs)** Another interesting 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
```

The atomic operations require that update be made globally, not on a local copy, and since TSO prevents reordering of writes, an atomic operation will have the effect of flushing the write 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+RMWs)** 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 revisted)** This example, forbidden by PSO, is allowed by WMO because the 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 forbidded 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.

### Operational Semantics

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

1. If $i =$ `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) $i =$ `sync`; or (b) $i$ accesses address $a$ and no operation that preceeds $i$ in program-order has an end-time that preceeds the begin-time of $i$.

1. If $i =$ `sync` then **fail**.
2. If $i =$ `M[`$a$`]:=` $v$ then append $i$ to $B(t)$.
3. 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**;
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 3** 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 $v$.

# 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 the "non-multi-copy-atomic" property [5]). This supports two important features found in advanced memory-subsystems: (1) cache hierarchies involving more than one shared cache; (2) an optimisation called *lazy invalidation* in which the cache coherence protocol can buffer invalidate messages (or update messages) until a `sync` is issued. 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 reodering 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, or by a coherence protocol that, upon seeing the write of thread 0, does not eagerly update the local cache of thread 2, allowing it to read a stale value.

**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 writes by the issuing thread have propagated to all other threads, but it also ensures that any writes that the issuing thread has observed 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 preceeded by a `sync` which is in turn preceeded 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. To our knowledge, however, this behaviour cannot be explained by the lazy invalidation optimisation.

### Operational Semantics

Once again, we define the allowed behaviours by way of an abstract machine with a state and set of state-transition rules. The state consists of:

- A trace $T$ (a sequence of operations in the format given in §2).
- A *value order* $V(a)$ for each address $a$, represented as a set of edges between values written to address $a$.
- A set $W$ of address-value pairs $(a, v)$, representing writes that have entered the memory-subsystem.
- A mapping $L$ from a thread-address pair $(t, a)$ to the last value seen at address $a$ on thread $t$.

In the *initial state*, $T$ is the trace we wish to check, $V(a) = \{\}$ for each address $a$, $W = \{\}$, and $L(t, a) = 0$ for each thread $t$ and address $a$.

Using the state-transition rules, if there is a path from the initial state to a state in which $T = []$ and $V(a)$ the `final` constraint on each address $a$, 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.

We first present a machine that ignores atomic read-modify-write operations, and then extend the machine to support them.

**Rule 1** 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) $i =$ `sync`; or (b) $i$ accesses address $a$ and no operation that preceeds $i$ in program-order has an end-time that preceeds the begin-time of $i$.

1. If $i =$ `sync` then **fail**.
2. If $i =$ `M[`$a$`]:=` $v$ then:
   (a) add $(a, v)$ to set $W$;
   (b) if $L(t, a) \neq v$, add edge $L(t, a) \rightarrow v$ to $V(a)$ and **fail** if a cycle is created.

(c) update $L(t, a)$ to $v$.

3. If $i =$ `M[`$a$`]==` $v$ then:

   (a) if $(a, v) \notin W$ then **fail**
   (b) if $L(t, a) \neq v$, add edge $L(t, a) \rightarrow v$ to $V(a)$ and **fail** if a cycle is created.
   (c) update $L(t, a)$ to $v$.

**Rule 2** Pick a thread $t$ non-deterministically. Remove the first operation $i$ executed by $t$ from the trace.

1. If $i \neq$ `sync` then **fail**;
2. Otherwise, for each address $a$ and thread $t' \neq t$:

   (a) let $w$ be the next value read from or written to address $a$ by thread $t'$ in the trace;
   (b) if $L(t, a) \neq w$ then add edge $L(t, a) \rightarrow w$ to $V(a)$ and **fail** if a cycle is created.

### Atomic operations

Atomic read-modify-write operations can be treated simply as adjacent program-order 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 $V(a)$ in which $v$ and $w$ are adjacent for each operation `<M[`$a$`]==` $v$`; M[`$a$`]:=` $w$`>`.

## 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

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

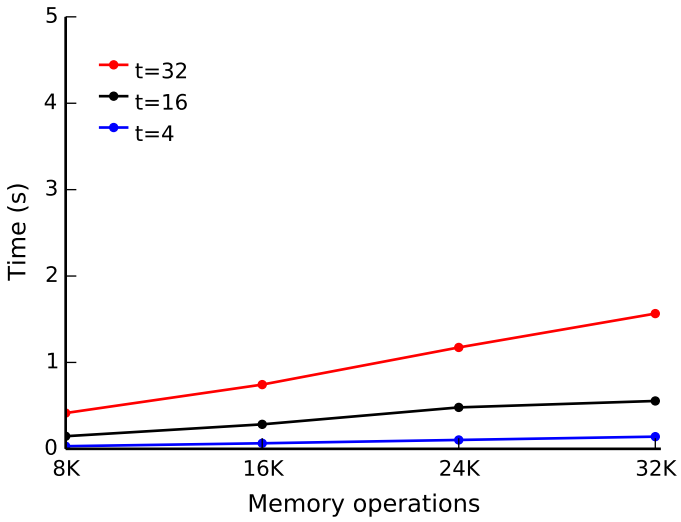


Figure 1: Performance of TSO checker.

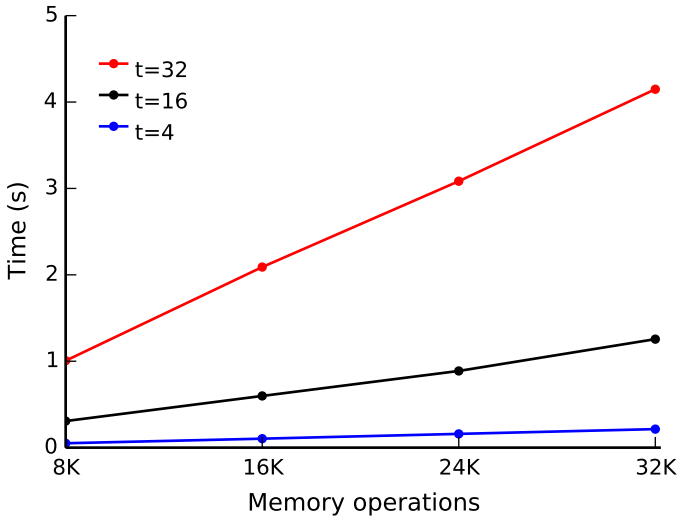


Figure 2: Performance of WMO checker.

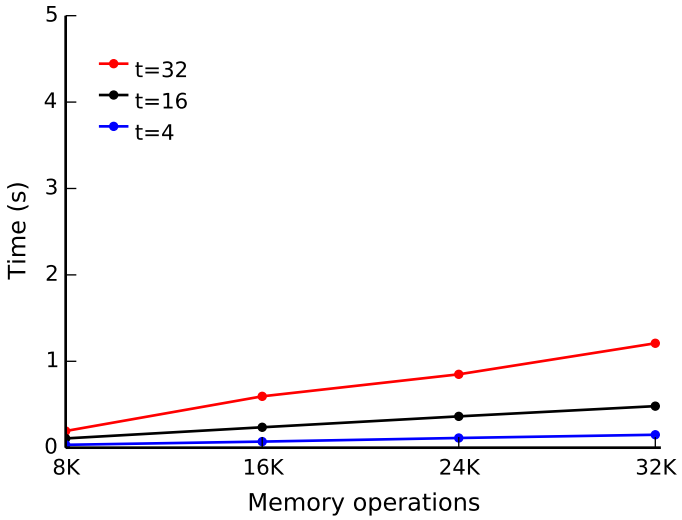


Figure 3: Performance of POW checker with `-g` flag specified.

($t \in \{4, 16, 32\}$), and addresses ($a \in \{4, 16, 32\}$). For each combination of parameters, we generate 16 traces, giving 576 traces for each supported model.

Figures 1, 2, and 3 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%.

# 7 Correctness

### Litmus tests

Axe has been applied to around 200 litmus tests from the PPCMEM distribution [11], and verified against the outcomes of the operational models given in §4 and §5, and also against the output of PPCMEM. The tests are present in the `tests/litmus` subdirectory, and the outcomes are listed in Appendix A.

### Random tests

Axe has been applied to around 200,000 randomly-generated traces present in the `tests/random` subdirectory, and verified against the outcomes of the operational models given in §4 and §5. Each of these traces is fairly short, ranging from around 10 to 50 memory operations in size.

### Cache tests

Axe has been applied to the CHERI cache implementation, and also to the model cache implementation mentioned in §6, giving the expected outcomes. For spaces reasons, we have not included these traces in the Axe distribution.

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

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

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


## Appendix A: 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 | | | | | ✓ |

| Test name | SC | TSO | PSO | WMO | POW |
|---|---|---|---|---|---|
| IRRWIW+sync+po | | | | ✓ | ✓ |
| IRRWIW+syncs | | | | | |
| IRWIW+addr+po | | | | ✓ | ✓ |
| IRWIW+addrs | | | | | ✓ |
| IRWIW | | | | ✓ | ✓ |
| 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 | | ✓ | ✓ | ✓ | ✓ |

| Test name | SC | TSO | PSO | WMO | POW |
|---|---|---|---|---|---|
| RWC+syncs | | | | | |
| S | | | ✓ | ✓ | ✓ |
| SB | | ✓ | ✓ | ✓ | ✓ |
| SB+sync+po | | ✓ | ✓ | ✓ | ✓ |
| SB+syncs | | | | | |
| 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 | | ✓ | ✓ | ✓ | ✓ |

| Test name | SC | TSO | PSO | WMO | POW |
|---|---|---|---|---|---|
| W+RWC+sync+addr+sync | | | | | |
| W+RWC+sync+po+po | | ✓ | ✓ | ✓ | ✓ |
| W+RWC+sync+po+sync | | | | ✓ | ✓ |
| W+RWC+syncs | | | | | |
| W+RWC+sync+sync+po | | ✓ | ✓ | ✓ | ✓ |
| 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 | | | ✓ | ✓ | ✓ |

| Test name | SC | TSO | PSO | WMO | POW |
|---|---|---|---|---|---|
| Z6.1+sync+po+addr | | | ✓ | ✓ | ✓ |
| Z6.1+sync+po+po | | | ✓ | ✓ | ✓ |
| Z6.1+sync+po+sync | | | ✓ | ✓ | ✓ |
| Z6.1+syncs | | | | | |
| Z6.1+sync+sync+addr | | | | | |
| 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 | | ✓ | ✓ | ✓ | ✓ |

| **Test name** | **SC** | **TSO** | **PSO** | **WMO** | **POW** |
|---|---|---|---|---|---|
| Z6.4+po+sync+po | | ✓ | ✓ | ✓ | ✓ |
| Z6.4+po+sync+sync | | | ✓ | ✓ | ✓ |
| Z6.4+sync+po+po | | ✓ | ✓ | ✓ | ✓ |
| Z6.4+sync+po+sync | | ✓ | ✓ | ✓ | ✓ |
| Z6.4+syncs | | | | | |
| 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 |