A better x86 memory model: x86-TSO (extended version)

Scott Owens Susmit Sarkar Peter Sewell

University of Cambridge

http://www.cl.cam.ac.uk/users/pes20/weakmemory

March 25, 2009 *Revision* : 1746

Abstract

Real multiprocessors do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, typically described in ambiguous prose, which lead to widespread confusion. These are prime targets for mechanized formalization. In previous work we produced a rigorous *x86-CC* model, formalizing the Intel and AMD architecture specifications of the time, but those turned out to be unsound with respect to actual hardware, as well as arguably too weak to program above. We discuss these issues and present a new *x86-TSO* model that suffers from neither problem, formalized in HOL4. We believe it is sound with respect to real processors, reflects better the vendor's intentions, and is also better suited for programming. We give two equivalent definitions of x86-TSO: an intuitive operational model based on local write buffers, and an axiomatic total store ordering model, similar to that of the SPARCv8. Both are adapted to handle x86-specific features. We have implemented the axiomatic model in our `memevents` tool, which calculates the set of all valid executions of test programs, and, for greater confidence, verify the witnesses of such executions directly, with code extracted from a third, more algorithmic, equivalent version of the definition.

Contents

1 Introduction

Most previous research on the semantics and verification of concurrent programs assumes sequential consistency: that accesses by multiple threads to a shared memory occur in a global-time linear order. Real multiprocessors, however, incorporate many performance optimisations. These are typically unobservable by single-threaded programs, but some have observable consequences for the behaviour of concurrent code. For example, on standard Intel or AMD x86 processors, given two memory locations x and y (initially holding 0), if two processors proc:0 and proc:1 respectively write 1 to x and y and then read from y and x, as in the program below, it is possible for both to read 0 *in the same execution.*

iwp2.3.a/amd4	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$1
poi:1	MOV EAX←[y]	MOV EBX←[x]
Allow: 0:EAX=0 ∧ 1:EBX=0		

One can view this as a visible consequence of *write buffering*: each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), so the reads from y and x can occur before the writes have propagated from the buffers to main memory. Such optimisations destroy the illusion of sequential consistency, making it impossible (at this level of abstraction) to reason in terms of an intuitive notion of global time.

To describe what programmers can rely on, processor vendors document *architectures*. These are loose specifications, claimed to cover a range of past and future actual processors, which should reveal enough for effective programming, but without unduly constraining future processor designs. In practice, however, they are informal prose documents, e.g. the Intel 64 and IA-32 Architectures SDM [2] and AMD64 Architecture Programmer's Manual [1]. Informal prose is a poor medium for loose specification of subtle properties, and, as we shall see in §2, such documents are often ambiguous, are sometimes incomplete (too weak to program above), and are sometimes unsound (with respect to the actual processors). Moreover, one cannot test programs above such a vague specification (one can only run programs on particular actual processors), and one cannot use them as criteria for testing processor implementations.

Architecture specifications are, therefore, prime targets for rigorous mechanised formalisation. In previous work [20] we introduced a rigorous x86-CC model, formalised in HOL4 [11], based on the

informal prose causal-consistency descriptions of the then-current Intel and AMD documentation. Unfortunately those, and hence also x86-CC, turned out to be unsound, forbidding some behaviour which actual processors exhibit.

In this paper we describe a new model, x86-TSO, also formalised in HOL4. To the best of our knowledge, x86-TSO is sound, is strong enough to program above, and is broadly in line with the vendors' intentions. We present two equivalent definitions of the model: an abstract machine, in §3.1, and an axiomatic version, in §3.2. We compensate for the main disadvantage of formalisation, that it can make specifications less widely accessible, by extensively annotating the mathematical definitions. To explore the consequences of the model, we have a hand-coded implementation in our `memevents` tool, which can explore all possible executions of litmus-test examples such as that above, and for greater confidence we have a verified execution checker extracted from the HOL4 axiomatic definition, in §4. We discuss related work in §5 and conclude in §6.

Further details can be found in the appendices. In Appendix A we discuss a number of litmus tests, comparing their behaviour in the model (as calculated by `memevents`) and on actual processors (as observed with our `litmus` tool). In Appendix B we reproduce the key "principles" from the informal-prose vendor documentation, for reference. Appendix C makes the notion of well-formed event structure precise, Appendix D collects a number of routine auxiliary definitions used in the definition of our models, and Appendix E gives an overview of the proofs of our results.

2 Many Memory Models

We begin by reviewing the informal-prose specifications of recent Intel and AMD documentation. There have been several versions, some differing radically; we contrast them with each other, and with what we know of the behaviour of actual processors.

2.1 pre-IWP (before Aug. 2007)

Early revisions of the Intel SDM (e.g. rev-22, Nov. 2006) gave an informal-prose model called 'processor ordering', unsupported by any litmus-test examples. It is hard to give a precise interpretation of this description.

2.2 IWP/AMD64-3.14/x86-CC

In August 2007, an Intel White Paper [12] (IWP) gave a somewhat more precise model, with 8 informal-prose principles supported by 10 litmus tests. This was incorporated, essentially unchanged, into later revisions of the Intel SDM (including rev.26–28), and AMD gave similar, though not identical, prose and tests [1]. These are essentially causal-consistency models. They allow independent readers to see independent writes (by different processors to different addresses) in different orders, as below (IRIW, see also [5]),

amd6	proc:0	proc:1	proc:2	proc:3
poi:0	MOV [x]←$1	MOV [y]←$1	MOV EAX←[x]	MOV ECX←[y]
poi:1			MOV EBX←[y]	MOV EDX←[x]
Final: 2:EAX=1 ∧ 2:EBX=0 ∧ 3:ECX=1 ∧ 3:EDX=0				
cc : Allow; tso : Forbid				

but require that, in some sense, causality is respected: *"P5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility)"*. These were the basis for our x86-CC model, for which a key issue was giving a reasonable interpretation to this "causality". Apart from that, the informal specifications were reasonably unambiguous — but they turned out to have two serious flaws.

First, they are arguably rather weak for programmers. In particular, they admit the IRIW behaviour above but, under reasonable assumptions on the strongest x86 memory barrier, MFENCE, adding MFENCEs would not suffice to recover sequential consistency [20, §2.12]. Here the specifications seem to be much looser than the behaviour of implemented processors: to the best of our knowledge, and following some testing, IRIW is not observable in practice. It appears that some JVM implementations depend on this fact, and would not be correct if one assumed only the IWP/AMD64-3.14/x86-CC architecture [9].

Second, more seriously, they are unsound with respect to current processors. The following n6 example, due to Paul Loewenstein [14], shows a behaviour that is observable (e.g. on an Intel Core 2 duo), but that is disallowed by x86-CC, and by any interpretation we can make of IWP and AMD64-3.14.

n6	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$2
poi:1	MOV EAX←[x]	MOV [x]←$2
poi:2	MOV EBX←[y]	
Final: 0:EAX=1 ∧ 0:EBX=0 ∧ [x]=1		
cc : Forbid; tso : Allow		

To see why this may be allowed by multiprocessors with FIFO write buffers, suppose that first the proc:1 write of [y]=2 is buffered, then proc:0 buffers its write of [x]=1, reads [x]=1 from its own write buffer, and reads [y]=0 from main memory, then proc:1 buffers its [x]=2 write and flushes its buffered [y]=2 and [x]=2 writes to memory, then finally proc:0 flushes its [x]=1 write to memory.

2.3 Intel SDM rev-29 (Nov. 2008)

The most recent x86 vendor specification, at the time of writing, is revision 29 of the Intel SDM (we are told that there will be a future revision of the AMD specification on similar lines). Key extracts are summarised in Appendix B. This is in a similar informal-prose style to previous versions, again supported by litmus tests, but is significantly different to IWP/AMD64-3.14/x86-CC. First, the IRIW final state above is forbidden [Example 7-7, rev-29], and the previous coherence condition: *"P6. In a multiprocessor system, stores to the same location have a total order"* has been replaced by: *"P9. Any two stores are seen in a consistent order by processors other than those performing the stores"*.

Second, the memory barrier instructions are now included, with *"P11. Reads cannot pass LFENCE and MFENCE instructions"* and *"P12. Writes cannot pass SFENCE and MFENCE instructions"*.

Third, same-processor writes are now explicitly ordered (we regarded this as implicit in the IWP *"P2. Stores are not reordered with other stores"*): *"P10. Writes by a single processor are observed in the same order by all processors"*.

This specification appears to deal with the unsoundness, admitting the n6 behaviour above, but, unfortunately, it is still problematic. The first issue is, again, how to interpret "causality" as used in P5. The second issue is one of weakness: the new P9 says nothing about observations of two stores by those two processors themselves (or by one of those processors and one other). This is illustrated by the following n5 and n4 examples. These final states were not allowed in x86-CC, and we would be surprised if they were allowed by any reasonable implementation (they are not allowed in a pure write-buffer implementation). We have not observed them on actual processors, and programming above a model that permitted them would be problematic. However, rev-29 appears to allow them.

n5	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [x]←$2
poi:1	MOV EAX←[x]	MOV EBX←[x]
Forbid: 0:EAX=2 ∧ 1:EBX=1		

n4	proc:0	proc:1
poi:0	MOV EAX←[x]	MOV ECX←[x]
poi:1	MOV [x]←$1	MOV [x]←$2
poi:2	MOV EBX←[x]	MOV EDX←[x]
Forbid: 0:EAX=2 ∧ 0:EBX=1∧ 1:ECX=1 ∧ 1:EDX=2		

Summarising the key litmus-test differences, we have:

	IWP/AMD64-3.14/x86-CC	rev-29	actual processors
IRIW	allowed	forbidden	not observed
n6	forbidden	allowed	observed
n4/n5	forbidden	allowed	not observed

There are also many non-differences: tests for which the behaviours coincide in all three cases. The test details can be found in Appendix A. They include the 9 other IWP tests, illustrating that various load and store reorderings other than that shown in iwp2.3.a/amd4 (§1) are not possible; the AMD MFENCE tests amd5 and amd10, and several other tests.

3 The x86-TSO Model

Given these problems with the informal specifications, we cannot produce a useful rigorous model by formalising the "principles" they contain (as we attempted with x86-CC [20]). Instead, we have to build a reasonable model that is consistent with the given litmus tests, observable processor behaviour, and with what we know of the needs of programmers and of the vendors intentions.

The fact that write buffering is observable (iwp2.3.a/amd4 and n6) but IRIW is not, together with the other tests that prohibit many other reorderings, strongly suggests that, apart from write buffering, all processors share the same view of memory (in contrast to x86-CC, where each processor had a separate view order). This is broadly similar to the SPARC Total Store Ordering (TSO) memory model [21, 23], which is essentially an axiomatic description of the behaviour of write-buffer multiprocessors. Moreover, while the term "TSO" is not used, informal discussions suggest this matches the intention behind the rev.29 informal specification. Accordingly, we define here a rigorous x86-TSO model.

After some preliminaries, we give two equivalent definitions of x86-TSO. The first, in §3.1, is an abstract machine, with explicit write buffers. The second, in §3.2, is an axiomatic model, defining valid executions in terms of memory orders and reads-from maps. In both, we deal with the x86 CISC instructions with multiple memory accesses, with x86 LOCK'd instructions (CMPXCHG, LOCK;INC, etc.), with potentially non-terminating computations, and with dependencies through registers. Together with our earlier instruction semantics, x86-TSO thus defines a complete semantics of programs.

The intended scope of x86-TSO, as for the x86-CC model, covers typical user code and most kernel code: programs using coherent write-back memory, without exceptions, misaligned or mixed-size accesses, 'non-temporal' operations (e.g. MOVNTI), self-modifying code, or page-table changes.

Basic Types: Actions, Events, and Event Structures As in our earlier work, the action of (any particular execution of) a program is abstracted into a set of *events* (with additional data) called an *event structure*. An event represents a read or write of a particular value to a memory address, or to a register, or the execution of a fence. Our earlier work includes a definition of the set of event structures generated by an assembly language program. For any such event structure, the memory model (there x86-CC, here x86-TSO) defines what a *valid execution* is.

In more detail, each machine-code instruction may have multiple events associated with it: events are indexed by an instruction ID *iiid* that identifies which processor the event occurred on and the position in the instruction stream of the instruction it comes from (the *program order index*, or *poi*). Events also have an event ID *eiid* to identify them within an instruction (to permit multiple, otherwise identical, events). An event structure indicates when one of an instruction's events has a dependency on another event of the same instruction with an *intra_causality* relation, a partial order over the events of each instruction. An event structure also records which events occur together in a locked instruction with *atomicity* data, a set of (disjoint, non-empty) sets of events which must occur atomically together.

Expressing this in HOL, we index processors by a type proc = num, take types address and value to both be the 32-bit words, and take a location to be either a memory address or a register of a particular processor:

location = LOCATION_REG **of** proc *'reg*
| LOCATION_MEM **of** address

The model is parameterised by a type *'reg* of x86 registers, which one should think of as an enumeration of the names of ordinary registers EAX, EBX, etc., the instruction pointer EIP, and the status flags. To identify an instance of an instruction in an execution, we specify its processor and its program order index.

iiid =⟨[*proc* : proc; *poi* : num]⟩

This introduces a type of records with two fields, a *proc* of type proc and a program order index *poi* of type num. An action is either a read or write of a value at some location, or a barrier:

dirn = *R* | W
barrier = LFENCE | SFENCE | MFENCE
action = ACCESS **of** dirn (*'reg* location) value | BARRIER **of** barrier

Finally, an event has an instruction instance id, an event id (of type eiid = num, unique per iiid), and an action:

event =⟪ *eiid* : eiid; *iiid* : iiid; *action* : action⟫

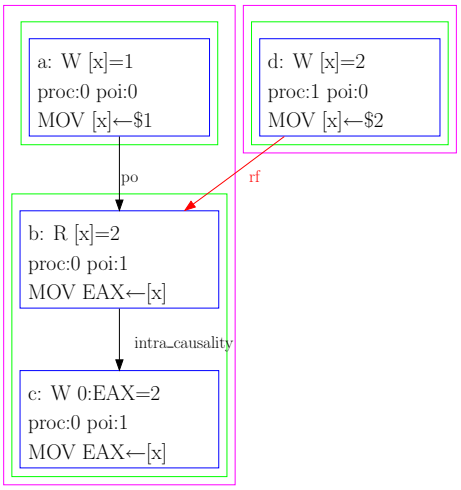
and an event structure E comprises a set of processors, a set of events, an intra-instruction causality relation, and a partial equivalence relation (PER) capturing sets of events which must occur atomically, all subject to some well-formedness conditions which we omit here.

event_structure =⟪ *procs* : proc set;
events : (′*reg* event)set;
intra_causality : (′*reg* event)reln;
atomicity : (′*reg* event)set set⟫

Example We show a very simple event structure below, for the program:

tso1	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [x]←$2
poi:1	MOV EAX←[x]	

There are four events — the inner (blue) boxes. The event ids are pretty-printed alphabetically, as a,b,c,d, etc. We also show the assembly instruction that gave rise to each event, e.g. MOV [x]←$1, though that is not formally part of the event structure.



tso1 rfmap 0 (of ess 0)

Note that events contain concrete values: in this particular event structure, there are two writes of x, with values 1 and 2, a read of [x] with value 2, and a write of proc:0's EAX register with value 2. Later we show two valid executions for this program, one for this event structure and one for another (note also that some event structures may not have any valid executions).
In the diagram, the instructions of each processor are clustered together, into the outermost (magenta) boxes, with program order (po) edges between them, and the events of each instruction are clustered together into the intermediate (green) boxes, with intra-causality edges as appropriate — here, in the MOV EAX←[x], the write of EAX is dependent on the read of x.

3.1 The x86-TSO Abstract Machine Memory Model

To understand our x86-TSO machine model, consider an idealised x86 multiprocessor system partitioned into two components: its memory and register state (of all its processors combined), and the rest of the system (the other parts of all the processor cores). Our abstract machine is a labelled transition system: a set of states, ranged over by s, and a transition relation $s \xrightarrow{l} s'$. An abstract machine state s models the state of the first component, the memory and register state of a multiprocessor system, and the machine interacts with the rest of the system by synchronising on labels l (the interface of the abstract machine), which include register and memory reads and writes. One should think of the machine as operating in parallel with the processor cores (absent their register/memory subsystems), executing their instruction streams in program order; the latter data is provided by an event structure. This partitioning does not correspond directly to the microarchitecture of any realistic x86 implementation, in which memory and registers would be implemented by separate and intricate mechanisms, but it is useful and sufficient for describing the programming model, which is the proper business of an architecture description. It also supports a precise correspondence with our axiomatic memory model. In more detail, the labels l are the values of the HOL type:

label = TAU | EVT **of** proc (′*reg action*) | LOCK **of** proc | UNLOCK **of** proc

- TAU, for an internal action by the machine;

Read from memory

$$\frac{\text{not\_blocked}\ \ s\ p \wedge (s.M\ \ a = \textsc{Some}\ \ v) \wedge \text{no\_pending}\ (s.B\ p)a}{s\ \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ R\ (\textsc{Location\_mem}\ a)v)}\ s}$$

Read from write buffer

$$\frac{\text{not\_blocked}\ \ s\ p \wedge (\exists b_1\ b_2.(s.B\ p = b_1 +\!\!+[(a, v)] +\!\!+ b_2) \wedge \text{no\_pending}\ \ b_1\ a)}{s\ \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ R\ (\textsc{Location\_mem}\ a)v)}\ s}$$

Read from register

$$\frac{(s.R\ p\ r = \textsc{Some}\ \ v)}{s\ \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ R\ (\textsc{Location\_reg}\ p\ r)v)}\ s}$$

Write to write buffer

$$\frac{\mathbf{T}}{s\ \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ \text{W}\ (\textsc{Location\_mem}\ a)v)}\ s\ \oplus \langle\!|B := s.B \oplus (p \mapsto [(a, v)] +\!\!+ (s.B\ p))|\!\rangle}$$

Write from write buffer to memory

$$\frac{\text{not\_blocked}\ \ s\ p \wedge (s.B\ p = b +\!\!+ [(a, v)])}{s\ \xrightarrow{\textsc{Tau}}\ s\ \oplus \langle\!|M := s.M \oplus (a \mapsto \textsc{Some}\ \ v); B := s.B \oplus (p \mapsto b)|\!\rangle}$$

Write to register

$$\frac{\mathbf{T}}{s\ \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ \text{W}\ (\textsc{Location\_reg}\ p\ r)v)}\ s\ \oplus \langle\!|R := s.R \oplus (p \mapsto ((s.R\ p) \oplus (r \mapsto \textsc{Some}\ \ v)))|\!\rangle}$$

Barrier

$$\frac{(b = \textsc{Mfence}) \implies (s.B\ p = [\,])}{s\ \xrightarrow{\textsc{Evt}\ p\ (\textsc{Barrier}\ b)}\ s}$$

Lock

$$\frac{(s.L = \textsc{None}) \wedge (s.B\ p = [\,])}{s\ \xrightarrow{\textsc{Lock}\ p}\ s\ \oplus \langle\!|L := \textsc{Some}\ \ p|\!\rangle}$$

Unlock

$$\frac{(s.L = \textsc{Some}\ \ p) \wedge (s.B\ p = [\,])}{s\ \xrightarrow{\textsc{Unlock}\ p}\ s\ \oplus \langle\!|L := \textsc{None}|\!\rangle}$$

Figure 1: The x86-TSO Machine Behaviour

- EVT p a, where a is an action, as defined above (a memory or register read or write, with its value, or a barrier), by processor p;
- LOCK p, indicating the start of a LOCK'd instruction by processor p; or
- UNLOCK p, for the end of a LOCK'd instruction by p.

(Note that there is nothing specific to any particular memory model in this interface.) The states of the x86-TSO machine are records, with fields R, giving a value for each register on each processor; M, giving a value for each shared memory location; B, modelling a write buffer for each processor, as a list of address/value pairs; and L, which is a global lock, either SOME p, if p holds the lock, or NONE. The HOL type is below.

```
machine_state =⟨[ R : proc → 'reg → value option; (* per-processor registers *)
                  M : address → value option; (* main memory *)
                  B : proc → (address#value)list; (* per-processor write buffers *)
                  L : proc option(* which processor holds the lock *)]⟩
```

The behaviour of the x86-TSO machine, the transition relation $s \xrightarrow{l} s'$, is defined by the rules in Fig. 1. The rules use two auxiliary definitions: processor p is *not blocked* in machine state s if either it holds the lock or no processor does; and there are *no pending* writes in a buffer b for address a if there are no (a, v) pairs in b. Restating the rules informally:

1. p can read v from memory at address a if p is not blocked, has no buffered writes to a, and the memory does contain v at a;
2. p can read v from its write buffer for address a if p is not blocked and has v as the newest write to a in its buffer;
3. p can read the stored value v from its register r at any time;
4. p can write v to its write buffer for address a at any time;
5. if p is not blocked, it can silently dequeue the oldest write from its write buffer to memory;
6. p can write value v to one of its registers r at any time;
7. if p's write buffer is empty, it can execute an MFENCE (so an MFENCE cannot proceed until all writes have been dequeued, modelling buffer flushing); LFENCE and SFENCE can occur at any time, making them no-ops;
8. if the lock is not held, and p's write buffer is empty, it can begin a LOCK'd instruction; and
9. if p holds the lock, and its write buffer is empty, it can end a LOCK'd instruction.

We emphasise that this is an *abstract* machine: we are concerned with its extensional behaviour, the (completed, finite or infinite) traces of labelled transitions it can perform (which should include the behaviour of real implementations), not with its internal states and the transition rules. The machine should provide a good model for programmers, but may bear little resemblance to the internal structure of implementations. Indeed, a realistic design would certainly not implement LOCK'd instructions with a global lock, and would have many other optimisations — the force of the x86-TSO model is that none of those have *programmer-visible* effects, except perhaps via performance observations.

One can imagine several variants of the machine with different degrees of locking. We conjecture that one could additionally make the read-register, write-register, write-buffer, and barrier transitions dependent on a not_blocked premise to arrive at a stricter, but observationally equivalent, machine, that might be simpler for programmers to reason with. However, the additions would make the proof of equivalence to the axiomatic memory model more difficult. We also conjecture that removing the not_blocked premise from the read-memory transition, and removing the requirement that the buffer is empty on a Lock label, would give a more liberal, but equivalent, machine. Again the equivalence proof would be more difficult. Should either of these variant machines become useful, we anticipate being able to prove them equivalent by working solely on the machines.

We relate the machine to event structures in two steps, which we summarise here (the HOL details can be found on-line [22] and are summarised in Appendix E). First, we define a more intensional event-machine: we annotate each memory and register location with an event option, recording the most recent write event (if any) to that location, refine write buffers to record lists of events rather than of plain location/value pairs, and annotate labels with the relevant events.

Theorem 1 *The annotation-erasure of the event-machine is exactly the machine presented above.*
[HOL proof]

Let *path* range over finite or infinite sequences of states and labels $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_1} s_1 \cdots$. We define (okMpath *path*) to hold if *path* is a path through the event-machine, the first state is an initial machine state, with empty write buffers, etc., and a progress condition holds: for each memory write in *path*, the corresponding TAU is eventually performed. Finally, given an event structure E, we say such a *path* is a machine execution corresponding to that event structure if (okEpath E *path*), i.e., if there is a 1:1 correspondence between non-TAU/LOCK/UNLOCK labels of *path* and the events of E, consistent with program order and intra-causality, and atomic sets are properly bracketed by LOCK/UNLOCK pairs.

3.2 The x86-TSO Axiomatic Memory Model

Our x86-TSO axiomatic memory model is based on the SPARCv8 memory model specification [21, 23], but adapted to x86 and in the same terms as our earlier x86-CC model. Compared with the SPARCv8 TSO specification, we omit instruction fetches (*IF*), instruction loads (*IL*), flushes (*F*), and stbars (~~*S*~~). The first three deal exclusively with instruction memory, which we do not model, and the last is useful only under the SPARC PSO memory model. To adapt it to x86 programs, we add register and fence events, generalize to support instructions that give rise to many events (partially ordered by an intra-instruction causality relation), and generalize atomic load/store pairs to locked instructions.

An execution is permitted by our memory model if there exists an *execution witness* X for its event structure E that is a *valid execution*. An execution witness contains a *memory_order*, an *rfmap*, and an *initial_state*; the rest of this section defines when these are valid.

execution_witness =
  ⟪ *memory_order* : ('*reg* event)reln;
    rfmap : ('*reg* event)reln;
    initial_state : ('*reg* location → value option)⟫

The memory order is a partial order that records the global ordering of memory events. It must be a total order on memory writes, and corresponds to the $\leq$ relation in SPARCv8, as constrained by the SPARCv8 **Order** condition (in figures, we use the label mo_non-po_write_write for the otherwise-unforced part of this order).

partial_order $(<_{X.memory\_order})$(mem_accesses E)

linear_order $((<_{X.memory\_order})|_{(\text{mem\_writes } E)})$(mem_writes E)

The initial state is a partial function from locations to values. Each read event's value must come either from the initial state or from a write event: the rfmap ('reads-from map') records which, containing (ew, er) pairs where the read er reads from the write ew. The *reads_from_map_candidates* predicate below ensures that the rfmap only relates such pairs with the same address and value. (The SPARCv8 model does not have an explicit representation of reads-from maps, and does not deal with initial states.)

reads_from_map_candidates E *rfmap* =
  $\forall (ew, er) \in rfmap.(er \in \text{reads } E) \wedge (ew \in \text{writes } E) \wedge$
    $(\text{loc } ew = \text{loc } er) \wedge (\text{value\_of } ew = \text{value\_of } er)$

We lift program order from instructions to a relation po_iico E over events, taking the union of program order of instructions and intra-instruction causality. This corresponds roughly to the ; in SPARCv8. However, *intra_causality* might not relate some pairs of events in an instruction, so our po_iico E will not generally be a total order for the events of a processor.

po_strict E =

$$\{(e_1, e_2) \mid (e_1.iiid.proc = e_2.iiid.proc) \wedge e_1.iiid.poi < e_2.iiid.poi \wedge e_1 \in E.events \wedge e_2 \in E.events\}$$

$$<_{(\text{po\_iico } E)} = \text{po\_strict } E \cup E.intra\_causality$$

The *check_rfmap_written* below ensures that the rfmap relates a read to the most recent preceding write. For a register read, this is the most recent write in program order. For a memory read, this is the most recent write in memory order among those that precede the read in either memory order or program order (intuitively, the first case is a read of a committed write and the second is a read from the local write buffer). The check_rfmap_written and reads_from_map_candidates predicates implement the SPARCv8 **Value** axiom above the rfmap witness data. The *check_rfmap_initial* predicate extends this to handle initial state, ensuring that any read not in the rfmap takes its value from the initial state, and that that read is not preceded by a write in memory order or program order.

previous_writes E er $<_{order}$ =
  $\{ew' \mid ew' \in \text{writes } E \wedge ew' <_{order} er \wedge (\text{loc } ew' = \text{loc } er)\}$

check_rfmap_written E X =
  $\forall (ew, er) \in (X.rfmap)$.
    if $ew \in$ mem_accesses E **then**
      $ew \in$ maximal_elements (previous_writes E er $(<_{X.memory\_order})$ $\cup$
                                 previous_writes E er $(<_{(\text{po\_iico } E)})$)
                               $(<_{X.memory\_order})$
    else (* ew IN reg_accesses E *)
      $ew \in$ maximal_elements (previous_writes E er $(<_{(\text{po\_iico } E)})$)$(<_{(\text{po\_iico } E)})$

check_rfmap_initial E X =
  $\forall er \in (\text{reads } E \setminus \text{range } X.rfmap)$.
    $(\exists l.(\text{loc } er = \textsc{Some } l) \wedge (\text{value\_of } er = X.initial\_state\ l)) \wedge$
    (previous_writes E er $(<_{X.memory\_order})$ $\cup$
      previous_writes E er $(<_{(\text{po\_iico } E)}) = \{\})$

We now further constrain the memory order, to ensure that it respects the relevant parts of program order, and that the memory accesses of a LOCK'd instruction do occur atomically.

- Program order is included in memory order, for a memory read before a memory access (labelled mo_po_read_access in figures) (SPARCv8's **LoadOp**):

  $\forall er \in (\text{mem\_reads } E).\forall e \in (\text{mem\_accesses } E).$
    $er <_{(\text{po\_iico } E)} e \implies er <_{X.memory\_order} e$

- Program order is included in memory order, for a memory write before a memory write (mo_po_write_write) (the SPARCv8 **StoreStore**):

  $\forall ew_1\ ew_2 \in (\text{mem\_writes } E).$
    $ew_1 <_{(\text{po\_iico } E)} ew_2 \implies ew_1 <_{X.memory\_order} ew_2$

- Program order is included in memory order, for a memory write before a memory read, *if* there is an MFENCE between (mo_po_mfence). (There is no need to include fence events themselves in the memory ordering.)

  $\forall ew \in (\text{mem\_writes } E).\forall er \in (\text{mem\_reads } E).\forall ef \in (\text{mfences } E).$
    $(ew <_{(\text{po\_iico } E)} ef \wedge ef <_{(\text{po\_iico } E)} er) \implies ew <_{X.memory\_order} er$

- Program order is included in memory order, for any two memory accesses where at least one is from a LOCK'd instruction (mo_po_access/lock):

  $\forall e_1\ e_2 \in (\text{mem\_accesses } E).\forall es \in (E.atomicity).$
    $((e_1 \in es \vee e_2 \in es) \wedge e_1 <_{(\text{po\_iico } E)} e_2) \implies e_1 <_{X.memory\_order} e_2$

- The memory accesses of a LOCK'd instruction occur atomically in memory order (mo_atomicity), i.e., there must be no intervening memory events. Further, all program order relationships between the locked memory accesses and other memory accesses are included in the memory order (this is a generalization of the SPARCv8 **Atomicity** axiom):

$$\begin{array}{l}\forall es \in (E.atomicity).\forall e \in (\text{mem\_accesses } E \setminus es).\\ \quad (\forall e' \in (es \cap \text{mem\_accesses } E).e <_{X.memory\_order} e') \vee\\ \quad (\forall e' \in (es \cap \text{mem\_accesses } E).e' <_{X.memory\_order} e)\end{array}$$

To deal properly with infinite executions, we also require that the prefixes of the memory order are all finite, ensuring that there are no limit points, and, to ensure that each write eventually takes effect globally, there must not be an infinite set of reads unrelated to any particular write, all on the same memory location (this formalizes the SPARCv8 **Termination** axiom).

$$\text{finite\_prefixes } (<_{X.memory\_order})(\text{mem\_accesses } E)$$

$$\begin{array}{l}\forall ew \in (\text{mem\_writes } E).\\ \quad \text{finite}\{er \mid er \in E.events \wedge (\text{loc } er = \text{loc } ew) \wedge\\ \qquad\qquad er \not<_{X.memory\_order} ew \wedge ew \not<_{X.memory\_order} er\}\end{array}$$

A final state of a valid execution takes the last write in memory order for each memory location, together with a maximal write in program order for each register (or the initial state, if there is no such write). This is uniquely defined assuming that no instruction has multiple unrelated writes to the same register — a reasonable property for x86 instructions.

The definition of valid_execution E X comprising the above conditions is equivalent to one in which $<_{X.memory\_order}$ is required to be a linear order, not just a partial order (again, the full details are on-line):

Theorem 2

1. *If* linear_valid_execution E X *then* valid_execution E X.
2. *If* valid_execution E X *then there exists an* $\hat{X}$ *with a linearisation of* X*'s memory order such that* linear_valid_execution E $\hat{X}$. [HOL proof]

Interpreting "not reordered with" Perhaps surprisingly, the above definition does not require that program order is included in memory order for a memory write followed by a read from the same address. The definition does imply that any such read cannot be speculated before the write (by check_rfmap_written, as that takes both $<_{(\text{po\_iico } E)}$ and $<_{X.memory\_order}$ into account). However, if one included a memory order edge, perhaps following a naive interpretation of the rev-29 *"P4. Reads may be reordered with older writes to different locations but not with older writes to the same location"*, then the model would be strictly stronger: the n7 example below would become forbidden, whereas it is allowed on x86-TSO. We conjecture that this would correspond to the (rather strange) machine with the Fig. 1 rules but without the read-from-write-buffer rule, in which any processor would have to flush its write buffer up to (and including) a local write before it can read from it.

n7	proc:0	proc:1	proc:2
poi:0	MOV [x]←$1	MOV [y]←$1	MOV ECX←[y]
poi:1	MOV EAX←[x]		MOV EDX←[x]
poi:2	MOV EBX←[y]		
Allow: 0:EAX=1 ∧ 0:EBX=0 ∧ 2:ECX=1 ∧ 2:EDX=0			

Examples We show two valid executions of the previous example program in Fig. 2. In both executions, the proc:0 W x=1 event is before the proc:1 W x=2 event in memory order (the bold mo_non-po _write_write edge). In the first execution, on the left, the proc:0 read of x reads from the most recent write in memory order (the combination of the bold mo_non-po_write_write edge and the mo_rf edge), which is the proc:1 W x=2. In the second execution, on the right, the proc:0 read of x reads from the most recent write in program order, which is the proc:0 W x=1. This example also illustrates some register events: the MOV EAX←[x] instruction gives rise to a memory read of x, followed by (in the intra-instruction causality relation) a register write of EAX.

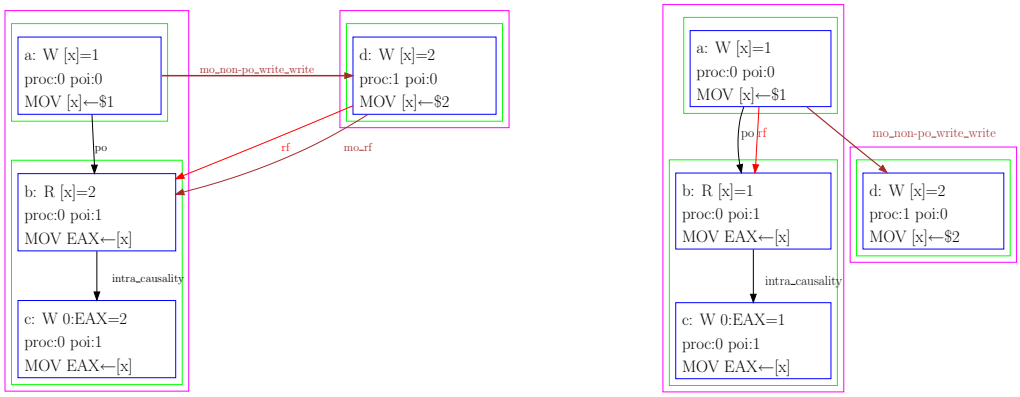


Figure 2: Example valid execution witnesses (for two different event structures)

3.3 The Machine and Axiomatic x86-TSO Models are Equivalent

To show these two definitions equivalent, given an event-machine *path*, we first build an execution witness path_to_X *path*, putting the memory reads and write-buffer flushes (TAUs) in their (linear) order from *path*. Now:

Theorem 3 *For any well-formed event structure E and event-machine path path, if* okEpath *E path and* okMpath *path, then* path_to_X *path is a valid execution for E.* [HOL proof]

Theorem 4 *For any well-formed event structure E, and valid execution X for E, there exists some event-machine path, such that* okEpath *E path and* okMpath *path, in which the memory reads and write-buffer flushes both respect* $<_{X.memory\_order}$. [hand proof, with some parts in HOL]

4 Verified Checker and Results

To explore the consequences of x86-TSO, we implemented the axiomatic model in our `memevents` tool, which exhaustively explores candidate execution witnesses. For greater confidence, we added to this a verified witness checker: we defined variants of event structures and execution witnesses, using lists instead of sets, wrote algorithmic versions of well_formed_event_structure and valid_execution, proved these equivalent (in the finite case) to our other definitions, extracted OCaml code from the HOL, and integrated that into `memevents`. (Obviously, this only provides assurance for positive tests, those with allowed final states.)

The `memevents` results coincide with our observations on real processors and the vendor specifications, for the 10 IWP tests, the (negated) IRIW test, the two MFENCE tests amd5 and amd10, our n2–n6 and n8, and rwc-fenced. The remaining tests (amd3, n1, n7, and rwc-unfenced) are "allow" tests for which we have not observed the specified final state in practice.

5 Related Work

There is an extensive literature on relaxed memory models, but most of it does not address x86, and we are not aware of any previous model that addresses the concerns of §2. We touch here on some of the most closely related work.

There are several surveys of weak memory models, including those by Adve and Gharachorloo [3], by Luchango [16], and by Higham, Kawash, and Verwaal [13]. The latter, in particular,formalises a range of models, including a TSO model, in both operational and axiomatic styles, and proves equivalence results. Their axiomatic TSO model is in a different style to ours, rather closer to the machine behaviour, and idealised rather than x86-specific. Burckhardt and Musuvathi [7, Appendix A] also give operational and axiomatic definitions of a TSO model and prove equivalence, but only for finite executions. Their models treat memory read, memory write and barrier events, but lack register events and locked instructions that force multiple events to happen atomically. Hangel et al. [10] describe the Sun TSOtool, checking

the observed behaviour of pseudo-randomly generated programs against a TSO model. Roy et al. [17] describe an efficient algorithm for checking whether an execution lies within an approximation to a TSO model, used in Intel's Random Instruction Test (RIT) generator. Boudol and Petri [6] give an operational model with hierarchical write buffers (thereby permitting IRIW behaviours), and prove sequential consistency for data-race-free (DRF) programs. Burckhardt et al. [8] define an x86 memory model based on IWP [12] (together with two MS CLR models). The mathematical form of their definitions is rather different to ours, using rewrite rules to re-order or eliminate memory accesses in sets of traces. Their model validates the 10 IWP tests and also some instances of IRIW (depending on how parallel compositions are associated), so it will not coincide with x86-TSO or x86-CC. Loewenstein et al. [15] describe a "golden memory model" for SPARC TSO, somewhat closer to a particular implementation microarchitecture than the abstract machine we give in §3.1, that they use for testing implementations. They argue that the additional intensional detail increases the effectiveness of simulation-based verification. Roychoudhury [18] describes a system for exhaustive search of executions in a logic programming system for versions of SPARC TSO and the Java Memory Model. Saraswat et al. [19] also define memory models in terms of local reordering, and prove a DRF theorem, but focus on high-level languages rather than processors. Several groups have used proof tools to tame the intricacies of these models, including Yang et al.[24], using Prolog and SAT solvers to explore an axiomatic Itanium model, and, turning briefly to high-level languages, Aspinall and Sevcik [4], who formalised and identified problems with the Java Memory Model using Isabelle/HOL.

6 Conclusion

We have described x86-TSO, a memory model for x86 processors that does not suffer from the ambiguities, weaknesses, or unsoundnesses of earlier models. Its abstract-machine definition should be intuitive for programmers, whereas its equivalent axiomatic definition supports the `memevents` exhaustive search and permits an easy comparison with related models; the similarity with SPARCv8 suggests x86-TSO is strong enough to program above. Mechanisation in HOL4 revealed a number of subtle points of detail, including some of the well-formed event structure conditions that we depend on (e.g. that instructions have no *internal* data races). We hope this will clarify the semantics of x86 architectures.

Acknowledgements We thank Luc Maranget for his work on `memevents`, and David Christie, Dave Dice, Doug Lea, Paul Loewenstein, Gil Neiger, and Francesco Zappa Nardelli for their helpful remarks. We acknowledge funding from EPSRC grant EP/F036345.

A Litmus Tests and Discussion

In this appendix we go through a number of litmus tests, including all those from the Intel IWP [12], the Intel SDM (Rev-29) [2], the AMD64 APM [1], and some others.

Tests are named iwpXXX, for the test numbered XXX from IWP, amdNNN, for the NNN'th test from the AMD64 APM, or nNNN, for other tests. We let x and y range over distinct aligned memory locations. Unless otherwise stated, all tests are considered with respect to an initial state in which those locations and all registers contain 0. We write assembly instructions in Intel syntax (as opposed to the AT&T syntax used by `gcc` and `gas`) except that we write an arrow ← instead of a comma, to clarify the direction of data flow.

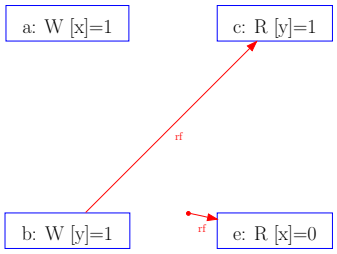
The results of the tests, in x86-CC, x86-TSO, and our observations on actual processors, are summarised in §A.9. In all cases the x86-TSO behaviours are consistent with our observations, though it does permit some behaviours that we have not observed. In other words, this data does not contradict the claim that x86-TSO is sound.

A.1 Load/Store Reordering

Rev-29 Example 7-1. Stores Are Not Reordered with Other Stores.

iwp2.1/amd1	proc:0	proc:1
poi:0	MOV [x]←$1	MOV EAX←[y]
poi:1	MOV [y]←$1	MOV EBX←[x]
Forbid: 1:EAX=1 ∧ 1:EBX=0		

This is Test 2.1 in the Intel White Paper (IWP), and also the first test in the AMD64-3.14. There are two stores on one processor to two different locations, and the other processor loads from those locations in the opposite order. Executions in which the first load reads from the second store, but the second load reads from the initial state, not from the first store, are forbidden. We illustrate such executions below, in a diagram showing the reads-from map over the memory events (eliding register events and the *eiid* and *iiid* data of each memory event). This test also shows that loads are not reordered with other loads.



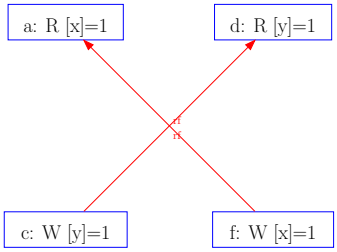
iwp2.1/amd1 rfmap 1 (of ess 0)

Rev-29 Example 7-2. Stores Are Not Reordered with Older Loads.

iwp2.2/amd2	proc:0	proc:1
poi:0	MOV EAX←[x]	MOV EBX←[y]
poi:1	MOV [y]←$1	MOV [x]←$1
Forbid: 0:EAX=1 ∧ 1:EBX=1		

Here the two processors read a value, from two different locations respectively, and then each stores a value to the other location. Executions as shown below, in which each reads from the write of the other

processor, are forbidden.

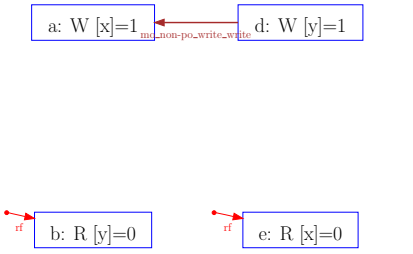


iwp2.2/amd2 rfmap 0 (of ess 0)

Rev-29 Example 7-3. Loads May be Reordered with Older Stores.

iwp2.3.a/amd4	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$1
poi:1	MOV EAX←[y]	MOV EBX←[x]
Allow: 0:EAX=0 ∧ 1:EBX=0		

This test illustrates non-sequentially-consistent behaviour allowed by store buffering, as discussed in §1. Each processor stores to one location and loads from the other location. Both loads are allowed to read from the initial state. We illustrate one such allowed execution below.

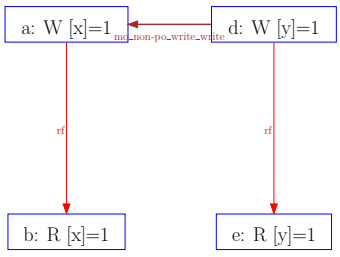


iwp2.3.a/amd4 vos 1 (of productive ess 3) showing Allow

Rev-29 Example 7-4. Loads Are not Reordered with Older Stores to the Same Location.

iwp2.3.b	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$1
poi:1	MOV EAX←[x]	MOV EBX←[y]
Require: 0:EAX=1 ∧ 1:EBX=1		

This is variant of the previous test in which each load reads from the same location as that processor stored to. Here they are forced to read from the local store. Operationally, loads must read from the local store buffer if it contains a write to the address in question. All executions must satisfy the 'Require' condition; one such is shown below. (The Rev-29 example is identical to the proc:0 part of iwp2.3.b; the proc:1 part adds no more force).

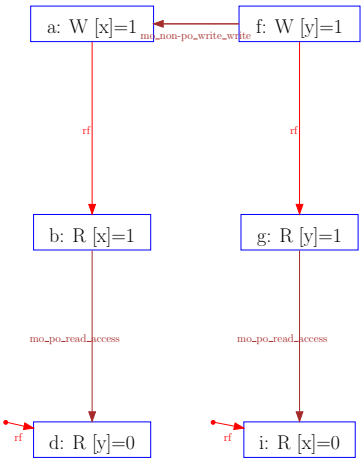


iwp2.3.b vos 1 (of productive ess 0) showing Require

Rev-29 Example 7-5. Intra-Processor Forwarding is Allowed.

iwp2.4/amd9	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$1
poi:1	MOV EAX←[x]	MOV ECX←[y]
poi:2	MOV EBX←[y]	MOV EDX←[x]
Allow: 0:EBX=0 ∧ 1:EDX=0		

This test is a variant of Example 7-3 (iwp2.3.a/amd4) in which each processor stores to a location *then reads from that location* before reading from the other location. Again, store buffers allow the specified final state, and one such execution is shown below.

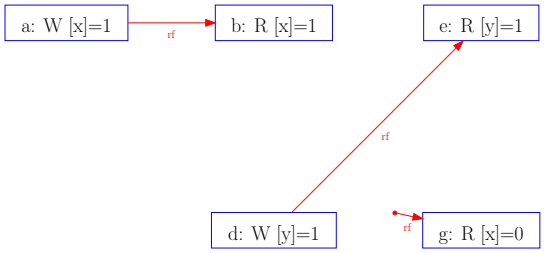


iwp2.4/amd9 vos 1 (of productive ess 3) showing Allow

Rev-29 Example 7-6. Stores Are Transitively Visible.

iwp2.5/amd8	proc:0	proc:1	proc:2
poi:0	MOV [x]←$1	MOV EAX←[x]	MOV EBX←[y]
poi:1		MOV [y]←$1	MOV ECX←[x]
Forbid: 1:EAX=1 ∧ 2:EBX=1 ∧ 2:ECX=0			

This test shows that a particular transitive chain of causality must be respected. Here proc:1 reads proc:0's write to x, then (in program order) proc:1 writing to y; proc:2 reads from that write of y, then (in program order) proc:1 reads x. That final read is not allowed to be from the initial state. The forbidden reads-from relationship is shown below. The Write-to-Read Causality (WRC) test of Boehm and Adve [5, Fig. 5] is similar to this test, but with fences between the proc:1 and proc:2 pairs of instructions. In x86-TSO the final state is forbidden without requiring such fences.

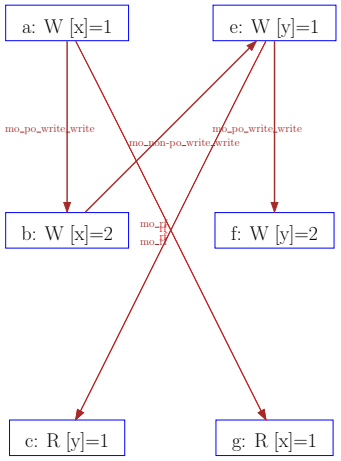


iwp2.5/amd8 rfmap 1 (of ess 0)

AMD3.14 Test 3

amd3	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$1
poi:1	MOV [x]←$2	MOV [y]←$2
poi:2	MOV EAX←[y]	MOV EBX←[x]
Allow: 0:EAX=1 ∧ 1:EBX=1		

This test is another variant of Example 7-3 (iwp2.3.a/amd4) in which each processor stores to a location *then stores again to that location* before reading from the other location. Again, store buffers allow each processor to read from the first write of the other processor, and one such execution is shown below.

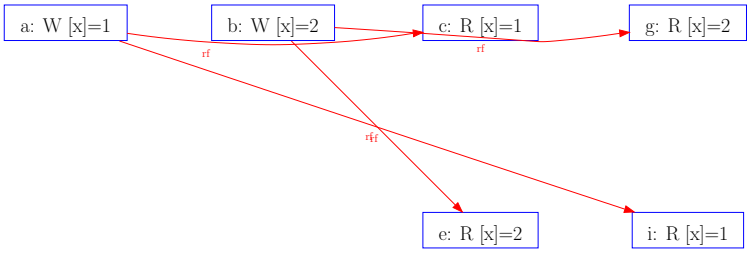


amd3 vos 0 (of productive ess 4) showing Allow

IWP Test 2.6. Total Order on Stores to the Same Location

iwp2.6	proc:0	proc:1	proc:2	proc:3
poi:0	MOV [x]←$1	MOV [x]←$2	MOV EAX←[x]	MOV ECX←[x]
poi:1			MOV EBX←[x]	MOV EDX←[x]
Forbid: 2:EAX=1 ∧ 2:EBX=2 ∧ 3:ECX=2 ∧ 3:EDX=1				

This test requires stores to the same location by two processors to be observed in the same order by two other processors. The forbidden reads-from relation is shown below. The test was in IWP, but is no longer present in rev-29.

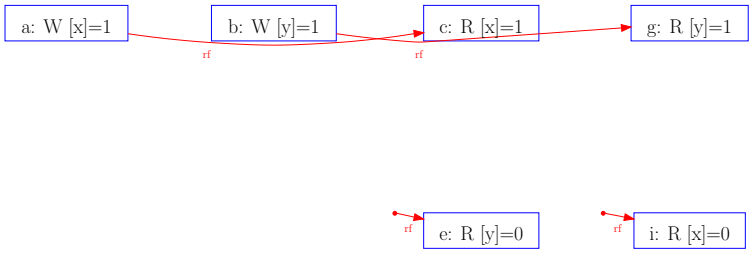


iwp2.6 rfmap 56 (of ess 0)

A.2 Independent Reads of Independent Writes

The following IRIW example, discussed here in §2.2 and by Boehm and Adve [5], has a final state permitted in AMD64-3.14 (explicitly), in IWP (implicitly), and in x86-CC. Rev-29, however, contains the same example but with the given final state forbidden (rev-29 Example 7-7. Stores Are Seen in a Consistent Order by Other Processors). We have not observed the final state in practice, and it is forbidden in x86-TSO. The reads-from relation required for the given final state is below; in such executions, proc:0 and proc:1 write to two different locations, and proc:2 and proc:3 read from those locations, seeing the writes in opposite orders (proc:2 sees the write to x but not the write to y, reading y from the initial state, whereas proc:3 sees the write to x but reads x from the initial state).

amd6	proc:0	proc:1	proc:2	proc:3
poi:0	MOV [x]←$1	MOV [y]←$1	MOV EAX←[x]	MOV ECX←[y]
poi:1			MOV EBX←[y]	MOV EDX←[x]
Final: 2:EAX=1 ∧ 2:EBX=0 ∧ 3:ECX=1 ∧ 3:EDX=0				
cc : Allow; tso : Forbid				



amd6 rfmap 5 (of ess 0)

A.3 Locked Instructions: Tests iwp2.7/amd7, iwp2.8.a, iwp2.8.b, n8, and n3

Only the following three IWP litmus tests involve locked instructions, and none of the three forbidden results are allowed in x86-TSO. However, only iwp2.8.a is forbidden by the parts of x86-TSO dealing with locking; the other two are forbidden by the core TSO properties directly (i.e., even if hypothetical non-locked versions of the XCHG instructions were used).

Rev-29 Example 7-8. Locked Instructions Have a Total Order.

iwp2.7/amd7	proc:0	proc:1	proc:2	proc:3
poi:0	XCHG [x]←EAX	XCHG [y]←EBX	MOV ECX←[x]	MOV ESI←[y]
poi:1			MOV EDX←[y]	MOV EDI←[x]
Initial state: 0:EAX = 1; 1:EBX = 1 (elsewhere 0)				
Forbid: 2:ECX=1 ∧ 2:EDX=0 ∧ 3:ESI=1 ∧ 3:EDI=0				

This is a variant of the IRIW example amd6 but using XCHG instructions (which are implicitly LOCK'd) instead of MOV instructions on proc:0 and proc:1.

Rev-29 Example 7-9. Loads Are not Reordered with Locks.

iwp2.8.a	proc:0	proc:1
poi:0	XCHG [x]←EAX	XCHG [y]←ECX
poi:1	MOV EBX←[y]	MOV EDX←[x]
Initial state: 0:EAX = 1; 1:ECX = 1 (elsewhere 0)		
Forbid: 0:EBX=0 ∧ 1:EDX=0		

Rev-29 Example 7-10. Stores Are not Reordered with Locks.

iwp2.8.b	proc:0	proc:1
poi:0	XCHG [x]←EAX	MOV EBX←[y]
poi:1	MOV [y]←$1	MOV ECX←[x]
Initial state: 0:EAX = 1 (elsewhere 0)		
Forbid: 1:EBX=1 ∧ 1:ECX=0		

Test n8. Loads Are not Reordered with Locks — Single-XCHG Variant.

n8	proc:0	proc:1
poi:0	XCHG [x]←EAX	MOV [y]←$1
poi:1	MOV EBX←[y]	MOV EDX←[x]
Initial state: 0:EAX = 1; [y] = 1 (elsewhere 0)		
Allow: 0:EAX=0 ∧ 1:EDX=0		

This test is a variant of iwp2.8.a, but with an unlocked write in place of one of the XCHG instructions. The x86-TSO model allows the result where EBX and EDX are both 0.

Test n3: Independent Locked Instructions and Stores

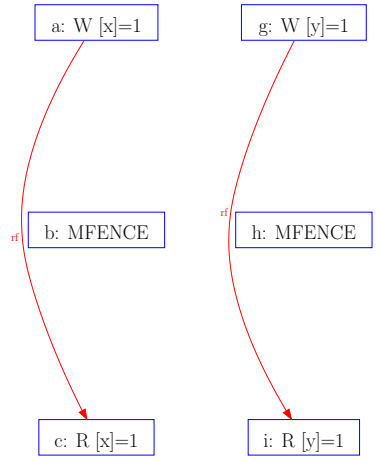
n3	proc:0	proc:1	proc:2	proc:3
poi:0	XCHG EAX←[x]	MOV [y]←$1	MOV EBX←[y]	MOV ESI←[x]
poi:1			MOV ECX←[x]	MOV EDI←[y]
poi:2			MOV EDX←[x]	MOV EBP←[y]
Initial state: 0:EAX = 1 (elsewhere 0)				
Final: 2:EBX=1 ∧ 2:ECX=0 ∧ 2:EDX=1 ∧ 3:ESI=1 ∧ 3:EDI=0 ∧ 3:EBP=1				
cc : Allow; tso : Forbid				

IWP left open the question of whether a locked instruction and a store to a different location could be seen in different orders by two other processors. We introduced the n3 test above to cover this case [20]. The given final state was allowed by x86-CC, but is not allowed by x86-TSO (again without the LOCK having anything to do with the reasoning). We have not observed the final state to occur in practice.

A.4 Fence Instructions: Tests amd5 and amd10

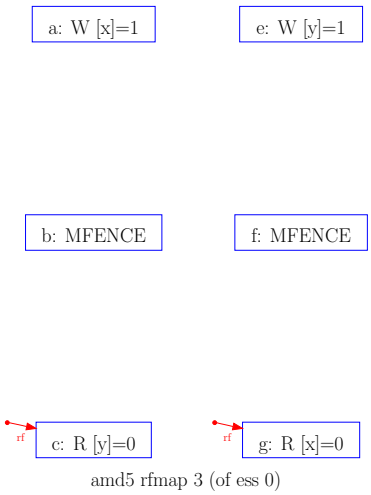
We handle fences by forcing write-to-read program order dependencies into the memory order. The following amd10 example is very similar to iwp2.8.a, and the forbidden behavior is prohibited by x86-TSO by essentially the same mechanism. Test amd5 below is similar. Our interpretation seems consistent with rev-29 principles P11 and P12.

amd10	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$1
poi:1	MFENCE	MFENCE
poi:2	MOV EAX←[x]	MOV ECX←[y]
poi:3	MOV EBX←[y]	MOV EDX←[x]
Final: 0:EBX=0 ∧ 1:EDX=0		
cc : Allow; tso : Forbid		



amd10 rfmap 3 (of ess 0)

amd5	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$1
poi:1	MFENCE	MFENCE
poi:2	MOV EAX←[y]	MOV EBX←[x]
Final: 0:EAX=0 ∧ 1:EBX=0		
cc : Allow; tso : Forbid		



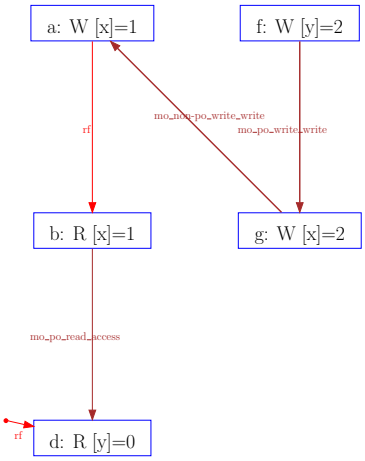
amd5 rfmap 3 (of ess 0)

A.5 The Unsoundness of IWP/AMD3.14/x86-CC: Test n6

The n6 test below, by Loewenstein, was discussed in §2.2, where we explained how the final state is allowed by write-buffer implementations.

n6	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [y]←$2
poi:1	MOV EAX←[x]	MOV [x]←$2
poi:2	MOV EBX←[y]	
Final: 0:EAX=1 ∧ 0:EBX=0 ∧ [x]=1		
cc : Forbid; tso : Allow		

The final state is also allowed in x86-TSO, as shown in the execution below.



n6 vos 0 (of productive ess 7) showing Allow

The final state is allowed (as far as we can interpret the principles) in rev-29. The final state is observable on Intel processors: we find witnesses on an Intel Core 2, with our `litmus` tool. However, the given final state is forbidden in our x86-CC model, and by any interpretation we can make of the IWP or AMD3.14 principles. We have:

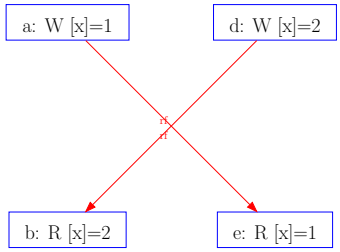
(a, b) in preserved_program_order (by IWP P4)
(b, d) in preserved_program_order (by IWP P1)
(d, f) in the view order of proc:0 (otherwise d could not read value 0)
(f, g) in preserved_program_order (by IWP P2)
So the view order of proc:0 must be a, b, d, f, g.
So (a, g) in the write serialization for location x (by IWP P6)
So the final state must have $[x] = 2$, not $[x] = 1$.

A.6 The Weakness of Rev-29: Tests n4 and n5

As discussed in §2.3, the following examples were not allowed in x86-CC, nor are they allowed by x86-TSO. We would be surprised if they were allowed by any reasonable implementation, and have not observed them on Intel processors. Programming above a model that permitted them would be problematic. However, the rev-29 principles seem to allow them. In particular, P9 has no force because

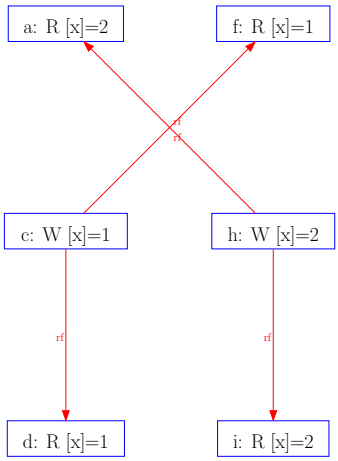
it only involves processors other than those performing the two stores.

n5	proc:0	proc:1
poi:0	MOV [x]←$1	MOV [x]←$2
poi:1	MOV EAX←[x]	MOV EBX←[x]
Forbid: 0:EAX=2 ∧ 1:EBX=1		



n5 rfmap 2 (of ess 0)

n4	proc:0	proc:1
poi:0	MOV EAX←[x]	MOV ECX←[x]
poi:1	MOV [x]←$1	MOV [x]←$2
poi:2	MOV EBX←[x]	MOV EDX←[x]
Forbid: 0:EAX=2 ∧ 0:EBX=1 ∧ 1:ECX=1 ∧ 1:EDX=2		



n4 rfmap 8 (of ess 0)

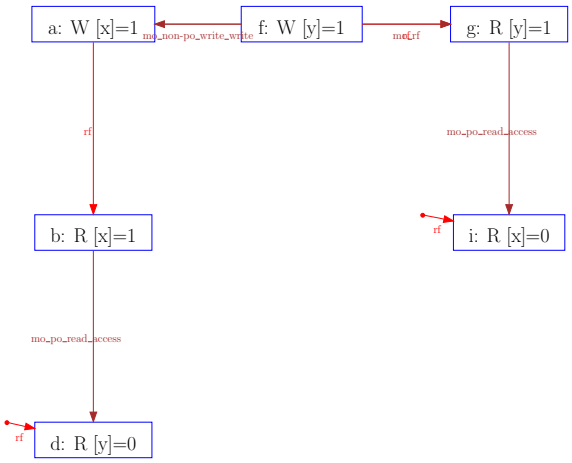
A.7 Interpreting the rev-29 "not reordered with": Test n7

Key phrases in the vendor documentation (Intel IWP, AMD64-3.14, and Intel SDM rev-29) are not given definite meanings, making it hard use them to justify x86-TSO. Principles P1, P2, P3, P4, and P8 of rev-29 (reproduced in Appendix B) all refer to some events being "not reordered with" others. These principles were also present in IWP/AMD64-3.14, and in x86-CC we interpreted them as giving rise to a preserved-program-order relation, which was included in a transitive happens-before relation, with which each processor's view order had to be consistent. In a TSO-based model, however, one might expect to interpret two events being "not reordered" by requiring that any program order relation between them must also appear in the transitive memory order (as suggested by P5). This seems reasonable, and is what we do, for P1, P2, P3, and P8. However, P4 states READS MAY BE REORDERED WITH OLDER WRITES TO DIFFERENT LOCATIONS BUT NOT WITH OLDER WRITES TO THE SAME LOCATION. There is no problem with the first part, but for the second part, that interpretation would forbid the final state of litmus test n7 below (discussed in §3.2), whereas it is allowed on x86-TSO.

n7	proc:0	proc:1	proc:2
poi:0	MOV [x]←$1	MOV [y]←$1	MOV ECX←[y]
poi:1	MOV EAX←[x]		MOV EDX←[x]
poi:2	MOV EBX←[y]		
Allow: 0:EAX=1 ∧ 0:EBX=0 ∧ 2:ECX=1 ∧ 2:EDX=0			

We illustrate this for the example x86-TSO valid execution below. If the second part of P4 were interpreted to give rise to preserved program order edges appearing in memory order, then there would be a

memory order edge from a to b. Read d would then have to read from f, not from the initial state.



n7 vos 0 (of productive ess 5) showing Allow

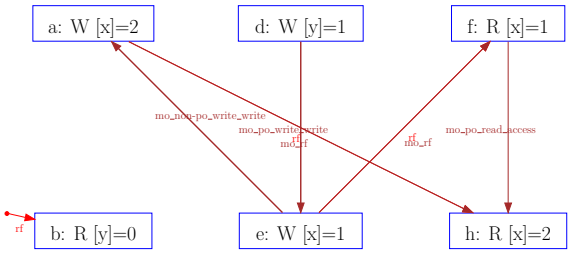
There are other valid executions, but not with the given final state.

A.8 Other Tests

Test n1: Reordering of Loads with Older Stores to Different Locations

n1	proc:0	proc:1	proc:2
poi:0	MOV [x]←$2	MOV [y]←$1	MOV EBX←[x]
poi:1	MOV EAX←[y]	MOV [x]←$1	MOV ECX←[x]
Allow: 0:EAX=0 ∧ 2:EBX=1 ∧ 2:ECX=2			

In x86-CC, the allowed final result of iwp2.3.a/amd4 did not require the reordering of loads with older stores to different locations that the test description spoke of. We introduced Test n1 [20] as an example where such reordering was (in x86-CC) essential. The final behaviour is still allowed in x86-TSO, as shown below.



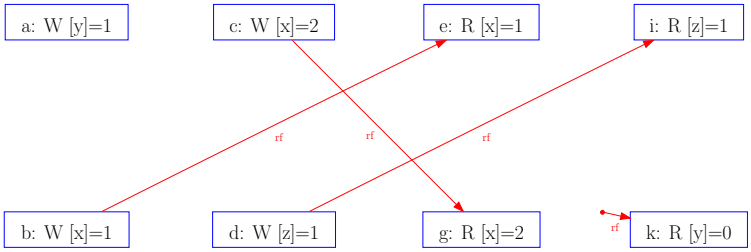
n1 vos 0 (of productive ess 21) showing Allow

Test n2: Transitive Causality via Preserved Program Order and the Write Serialisation

n2	proc:0	proc:1	proc:2	proc:3
poi:0	MOV [y]←$1	MOV [x]←$2	MOV EAX←[x]	MOV ECX←[z]
poi:1	MOV [x]←$1	MOV [z]←$1	MOV EBX←[x]	MOV EDX←[y]
Forbid: 2:EAX=1 ∧ 2:EBX=2 ∧ 3:ECX=1 ∧ 3:EDX=0				

We introduced n2 [20] to show transitivity through preserved program order (of the proc:0 and proc:1 events) and the write serialisation for x (which has W [x]=1 before W [x]=2 by the proc:2 observations).

There are no x86-TSO executions with the reads-from map below.

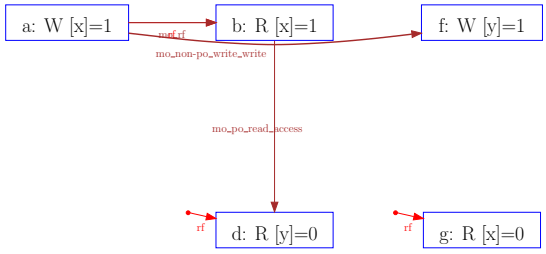


n2 rfmap 26 (of ess 0)

Test RWC (unfenced): Read-to-Write Causality

rwc-unfenced	proc:0	proc:1	proc:2
poi:0	MOV [x]←$1	MOV EAX←[x]	MOV [y]←$1
poi:1		MOV EBX←[y]	MOV ECX←[x]
Allow: 1:EAX=1 ∧ 1:EBX=0 ∧ 2:ECX=0			

This test is based on the Read-to-Write Causality (RWC) test of Boehm and Adve [5, Fig. 6]. With no fences, the non-SC behaviour below is allowed in x86-TSO.

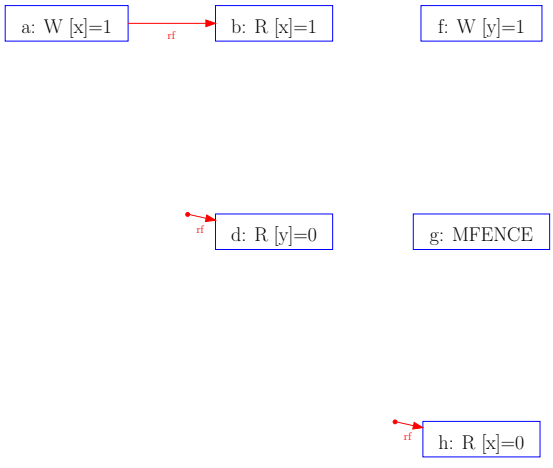


rwc-unfenced vos 0 (of productive ess 3) showing Allow

Test RWC (singly fenced): Read-to-Write Causality

rwc-fenced	proc:0	proc:1	proc:2
poi:0	MOV [x]←$1	MOV EAX←[x]	MOV [y]←$1
poi:1		MOV EBX←[y]	MFENCE
poi:2			MOV ECX←[x]
Final: 1:EAX=1 ∧ 1:EBX=0 ∧ 2:ECX=0			
cc : Allow; tso : Forbid			

Adding an MFENCE to proc:2 rules out that behaviour:



rwc-fenced rfmap 3 (of ess 0)

A.9 Summary of Test Results

The table below summarises the results for all the previous tests in the x86-CC model, the x86-TSO model, and our observations on actual processors. The memevents-cc.txt and memevents-tso.txt columns give the behaviour in our x86-CC and x86-TSO models respectively[1]. These results are from our `memevents` tool, which includes hand-written executable implementations of the formal models. For x86-TSO, "Allow" results are validated with the verified checker described in §4. The "actual processors" data is from observations with our `litmus` tool on an Intel Core 2 Duo, and indicates the number of positive (for "Allow" or "Require" tests) or negative (for "Forbid" tests) observations out of the number of runs.

	memevents-cc	memevents-tso	Actual hardware	
amd10	Allow	Forbid	Forbid validated	0/200000
amd3	Allow	Allow	Allow not validated	0/200000
amd5	Allow	Forbid	Forbid validated	0/200000
amd6	Allow	Forbid	Forbid validated	0/200000
iwp2.1/amd1	Forbid	Forbid	Forbid validated	0/200000
iwp2.2/amd2	Forbid	Forbid	Forbid validated	0/200000
iwp2.3.a/amd4	Allow	Allow	Allow validated	46/200000
iwp2.3.b	Require	Require	Require validated	200000/200000
iwp2.4/amd9	Allow	Allow	Allow validated	193/200000
iwp2.5/amd8	Forbid	Forbid	Forbid validated	0/200000
iwp2.6	Forbid	Forbid	Forbid validated	0/200000
iwp2.7/amd7	Forbid	Forbid	Forbid validated	0/200000
iwp2.8.a	Forbid	Forbid	Forbid validated	0/200000
iwp2.8.b	Forbid	Forbid	Forbid validated	0/200000
n1	Allow	Allow	Allow not validated	0/200000
n2	Forbid	Forbid	Forbid validated	0/200000
n3	Allow	Forbid	Forbid validated	0/200000
n4	Forbid	Forbid	Forbid validated	0/200000
n5	Forbid	Forbid	Forbid validated	0/200000
n6	Forbid	Allow	Allow validated	9/200000
n7	Allow	Allow	Allow not validated	0/200000
n8	Allow	Allow	Allow validated	130832/200000
rwc-fenced	Allow	Forbid	Forbid validated	0/200000
rwc-unfenced	Allow	Allow	Allow not validated	0/200000

Discrepancies between x86-TSO and the observed behaviour could be of two kinds: For a "Forbid" or "Require" test, an observation of the forbidden result, which we would indicate with "Forbid NOT validated" or "Require NOT validated" respectively, would indicate that the formal x86-TSO model is unsound with respect to actual hardware. We see no such observations.

For an "Allow" test, the absence of an observation of the allowed result, which we indicate with "Allow not validated" (in blue), may suggest that x86-TSO is weaker (allowing more behaviours) than actual processors, or may merely mean that the test has not been run sufficiently often or with the processor in a state to exhibit the behaviour.

Note also that this experimental data is from just a few tests run on just one machine of a specific implementation, whereas many x86 implementations, with radically different microarchitectures, are in widespread use. Further testing would be desirable to increase confidence in the soundness of x86-TSO.

It would be very surprising if x86-TSO were not more liberal in some ways than actual processors. For example, actual processors will have size-bounded write buffers, whereas x86-TSO does not, and probably should not (otherwise reasoning above the model would be needlessly complex and implementation-specific).

[1]For the fence tests (amd10, amd5, and rwc-fenced) the x86-CC column indicates the behaviour in the extension of our formal x86-CC model (which did not cover fences) with MFENCE enforcing local write-read ordering in happens-before.

B The IWP and Rev-29 Principles and Litmus Tests

B.1 Principles

The Intel documentation, both IWP and the SDM Vol.3A rev-29 §7.2.2, states a number of "principles". We reproduce them here for reference.

Eight of the rev-29 principles correspond to principles from IWP, in the sense that they have the same force for the fragment we are considering, but may have additional conditions, here greyed out, relevant to other instructions. For consistency with our previous work we tag these P1–P8, following the IWP numbering. In addition there are five new clauses, which we tag P9–P14. We list them in the same order as Rev-29.

P1 Reads are not reordered with other reads.

P3 Writes are not reordered with older reads.

P2 Writes to memory are not reordered with other writes[, with the exception of writes executed with the CLFLUSH instruction,streaming stores (writes) executed with the non-temporal move instructions (MOVNTI, MOVNTQ, MOVNTDQ, MOVNTPS, and MOVNTPD),string operations (see Section 7.2.4.1)].

P4 Reads may be reordered with older writes to different locations but not with older writes to the same location.

P8 Reads or writes cannot be reordered with [I/O instructions,] locked instructions[, or serializing instructions].

P11 Reads cannot pass LFENCE and MFENCE instructions.

P12 Writes cannot pass SFENCE and MFENCE instructions.

In a multiple-processor system, the following ordering principles apply:

P13 Individual processors use the same ordering principles as in a single-processor system.

P10 Writes by a single processor are observed in the same order by all processors.

P14 Writes from an individual processor are NOT ordered with respect to the writes from other processors.

P5 Memory ordering obeys causality (memory ordering respects transitive visibility).

P9 Any two stores are seen in a consistent order by processors other than those performing the stores.

P7 Locked instructions have a total order.

Summarising the differences of rev-29 with respect to IWP, the IWP P6

P6 In a multiprocessor system, stores to the same location have a total order.

has been replaced by P9; P11 and P12, concerning the fence instructions, have been added, and P10, P13 and P14 have been added (though they might have been considered implicit in IWP).

B.2 Litmus Tests

The rev-29 litmus tests, Examples 7-1 to 7-10, are essentially identical to the IWP litmus tests, except that iwp2.6 is removed and the IRIW test, with forbidden final state, is added. Rev-29 also adds 6 Litmus tests for string operations, which are not in the fragment of the ISA we consider in this paper.

The vendor litmus tests appear to be a more reliable source than the prose principles: the litmus tests are almost completely unambiguous (the only exception is that in some cases it is not explicitly stated that certain locations are not aliased), and, as far as we can observe, are consistent with the behaviour of actual processors, whereas the principles are hard to interpret, as we have discussed. Of course, the litmus tests do not by themselves *determine* a useful memory model, as they specify behaviour only for a small number of very specific programs.

C Well-Formed Event Structures

In this appendix we define our well-formedness condition for event structures, comprising a number of sanity conditions (that there are only a finite number of processors, that the intra-instruction causality relation is indeed a partial order, etc.). This follows our x86-CC work, in which we proved that all event structures arising from the semantics for assembly programs are indeed well-formed.

well_formed_event_structure $E =$
(* The set of events is at most countable *)
countable $E.events \wedge$

(* there are only a finite number of processors *)
(finite $E.procs$) $\wedge$

(* all events are from one of those processors *)
$(\forall e \in (E.events).\mathsf{proc}\ e\ \in\ E.procs) \wedge$

(* the eiid and iiid of an event (together) identify it uniquely *)
$(\forall e_1\ e_2 \in (E.events).(e_1.\mathsf{eiid} = e_2.\mathsf{eiid}) \wedge (e_1.iiid = e_2.iiid)\ \Longrightarrow\ (e_1 = e_2)) \wedge$

(* intra-instruction causality is a partial order over the events *)
partial_order $(E.intra\_causality) E.events \wedge$

(* ...and moreover, is *intra*-instruction *)
$(\forall (e_1, e_2) \in (E.intra\_causality).(e_1.iiid = e_2.iiid)) \wedge$

(* the atomicity data is a partial equivalence relation: the atomic sets of events are disjoint *)
per $E.events\ E.atomicity \wedge$

(* atomic sets are *intra* instruction *)
$(\forall es \in (E.atomicity).\forall e_1\ e_2 \in es.(e_1.iiid = e_2.iiid)) \wedge$

(* accesses to a register on a processor can only be by that processor *)
$(\forall e \in (E.events).\forall p\ r.(\text{loc}\ e = \textsc{Some}\ (\textsc{Location\_reg}\ p\ r))\ \Longrightarrow\ (p = \mathsf{proc}\ e)) \wedge$

(* An event never comes after an infinite number of other events in program order *)
finite_prefixes (po_iico E)$E.events \wedge$

(* The additional properties below hold, for the ISA fragment dealt with in [SSFN+09], and were useful for the metatheory there, but seem less essential than those above. *)

(* there is no intra-causality edge *from* a memory write *)
$(\forall (e_1, e_2) \in (E.intra\_causality).e_1 \neq e_2\ \Longrightarrow\ e_1\ \notin\ \text{mem\_writes}\ E) \wedge$

(* if an instruction two events on a location and one is a write, then there must be an intra-causality edge between them. In other words, there cannot be a local race within an instruction *)
$(\forall (e_1 \in \text{writes}\ E) e_2.$
$(e_1 \neq e_2) \wedge$
$(e_2\ \in\ \text{writes}\ E \vee e_2\ \in\ \text{reads}\ E) \wedge$
$(e_1.iiid = e_2.iiid) \wedge$
$(\text{loc}\ e_1 = \text{loc}\ e_2)$
$\Longrightarrow$
$(e_1, e_2) \in\ E.intra\_causality \vee$
$(e_2, e_1) \in\ E.intra\_causality) \wedge$

(* each atomic set includes all the events of its instruction *)
$(\forall es \in (E.atomicity).\forall e_1 \in es.\forall e_2 \in (E.events).(e_1.iiid = e_2.iiid)\ \Longrightarrow\ e_2\ \in\ es) \wedge$

(* all locked instructions include at least one memory read *)
$(\forall es \in (E.atomicity).\exists e \in es.e \in \text{mem\_reads } E)$

D Auxiliary Definitions

This appendix collects the remaining auxiliary definitions used in the x86-TSO axiomatic and abstract machine memory model definitions. These are mostly routine, but presented here for reference.

D.1 Axiomatic Memory Model

type_abbrev proc : num

type_abbrev Ximm : word32

type_abbrev address : Ximm

type_abbrev value : Ximm

type_abbrev eiid : num

type_abbrev reln : $'a\#'a \rightarrow$ bool

is_mem_access $e = \exists d\ a\ v.e.action = \text{Access}\ d\ (\text{Location\_mem}\ a)v$

writes $E = \{e \mid e \in E.events \wedge \exists l\ v.e.action = \text{Access W}\ l\ v\}$

reads $E = \{e \mid e \in E.events \wedge \exists l\ v.e.action = \text{Access}\ R\ l\ v\}$

fences $E = \{e \mid e \in E.events \wedge (\exists f.e.action = \text{Barrier}\ f)\}$

mfences $E = \{e \mid e \in E.events \wedge (e.action = \text{Barrier Mfence})\}$

mem_writes $E = \{e \mid e \in E.events \wedge \exists a\ v.e.action = \text{Access W (Location\_mem}\ a)v\}$

mem_reads $E = \{e \mid e \in E.events \wedge \exists a\ v.e.action = \text{Access}\ R\ (\text{Location\_mem}\ a)v\}$

reg_writes $E = \{e \mid e \in E.events \wedge \exists p\ r\ v.e.action = \text{Access W (Location\_reg}\ p\ r)v\}$

reg_reads $E = \{e \mid e \in E.events \wedge \exists p\ r\ v.e.action = \text{Access}\ R\ (\text{Location\_reg}\ p\ r)v\}$

mem_accesses $E = \{e \mid e \in E.events \wedge (\exists d\ a\ v.e.action = \text{Access}\ d\ (\text{Location\_mem}\ a)v)\}$

reg_accesses $E = \{e \mid e \in E.events \wedge \exists d\ p\ r\ v.e.action = \text{Access}\ d\ (\text{Location\_reg}\ p\ r)v\}$

loc $e =$
  case $e.action$ **of**
    Access $d\ l\ v \rightarrow$ Some l
  $\|$ Barrier $f \rightarrow$ None

```
value_of e =
  case e.action of
    ACCESS d l v → SOME v
  ‖ BARRIER f → NONE

proc e = e.iiid.proc
```

The following definition of valid executions collects together the conditions in the text of §3.2.

```
valid_execution E X =
  partial_order X.memory_order (mem_accesses E) ∧
  linear_order (X.memory_order|_(mem_writes E))(mem_writes E) ∧
  finite_prefixes X.memory_order (mem_accesses E) ∧
  (∀ew ∈ (mem_writes E).
    finite{er | er ∈ E.events ∧ (loc er = loc ew) ∧
                  (er, ew) ∉ X.memory_order ∧ (ew, er) ∉ X.memory_order}) ∧
  (∀er ∈ (mem_reads E).∀e ∈ (mem_accesses E).(er, e) ∈ po_iico E ⟹ (er, e) ∈ X.memory_order) ∧
  (∀ew1 ew2 ∈ (mem_writes E).(ew1, ew2) ∈ po_iico E ⟹ (ew1, ew2) ∈ X.memory_order) ∧
  (∀ew ∈ (mem_writes E).∀er ∈ (mem_reads E).∀ef ∈ (mfences E).
    (ew, ef) ∈ po_iico E ∧ (ef, er) ∈ po_iico E ⟹ (ew, er) ∈ X.memory_order) ∧
  (∀e1 e2 ∈ (mem_accesses E).∀es ∈ (E.atomicity).
    (e1 ∈ es ∨ e2 ∈ es) ∧ (e1, e2) ∈ po_iico E
    ⟹
    (e1, e2) ∈ X.memory_order) ∧
  (∀es ∈ (E.atomicity).∀e ∈ (mem_accesses E \ es).
    (∀e' ∈ (es ∩ mem_accesses E).(e, e') ∈ X.memory_order) ∨
    (∀e' ∈ (es ∩ mem_accesses E).(e', e) ∈ X.memory_order)) ∧
  X.rfmap ∈ reads_from_map_candidates E ∧
  check_rfmap_written E X ∧
  check_rfmap_initial E X


max_state_updates E X l =
  {value_of ew | ew ∈ maximal_elements
                        {ew' | ew' ∈ writes E ∧ (loc ew' = SOME l)}
                        (case l of
                              LOCATION_MEM a → X.memory_order
                            ‖ LOCATION_REG p r → po_iico E)}


(check_final_state E X NONE =
  ¬(finite E.events)) ∧
(check_final_state E X (SOME final_state) =
  finite E.events ∧
  (∀l.
    if (max_state_updates E X l) = {} then
      final_state l = X.initial_state l
    else
      final_state l ∈ max_state_updates E X l))
```

D.2 Abstract Machine Memory Model

```
clause_name x = T


not_blocked s p = (s.L = NONE) ∨ (s.L = SOME p)


no_pending b a = ¬(∃v'. MEM (a, v')b)
```

The following is the primary HOL definition of the abstract machine transitions. The rules in Fig. 1 are typeset from an equivalent HOL definition in a slightly different style.

```
(∀s a v p.
 clause_name "read-mem" ∧
 not_blocked s p ∧
 (s.M a = SOME v) ∧
 no_pending (s.B p) a
  ⟹
 machine_trans s
               (EVT p (ACCESS R (LOCATION_MEM a) v))
               s) ∧

(∀s a v p.
 clause_name "read-buffer" ∧
 not_blocked s p ∧
 (∃b1 b2. (s.B p = b1 ++ [(a, v)] ++ b2) ∧ no_pending b1 a)
  ⟹
 machine_trans s
               (EVT p (ACCESS R (LOCATION_MEM a) v))
               s) ∧

(∀s r v p.
 clause_name "read-reg" ∧
 (*not_blocked s p /\*)
 (s.R p r = SOME v)
  ⟹
 machine_trans s
               (EVT p (ACCESS R (LOCATION_REG p r) v))
               s) ∧

(∀s a v p s'.
 clause_name "write-buffer" ∧
 (*not_blocked s p /\*)
 (s' = ⟦ R := s.R;
         M := s.M;
         B := (p ↦ (a, v) ∈ s.B p) s.B;
         L := s.L ⟧)
  ⟹
 machine_trans s
               (EVT p (ACCESS W (LOCATION_MEM a) v))
               s') ∧

(∀s a v p b s'.
 clause_name "write-mem" ∧
 not_blocked s p ∧
 (s.B p = b ++ [(a, v)]) ∧
 (s' = ⟦ R := s.R;
         M := (a ↦ SOME v) s.M;
         B := (p ↦ b) s.B;
         L := s.L ⟧)
  ⟹
 machine_trans s TAU s') ∧

(∀s r v p s'.
 clause_name "write-reg" ∧
 (*not_blocked s p*)
 (s' = ⟦ R := (p ↦ (r ↦ SOME v) (s.R p)) s.R;
```

```
        M := s.M;
        B := s.B;
        L := s.L⟫)
 ⟹
machine_trans s
              (Evt p (Access W (Location_reg p r)v))
              s') ∧

(∀s p.
clause_name "barrier" ∧
(*not_blocked s p /\*)
(s.B p = [ ])
 ⟹
machine_trans s (Evt p (Barrier Mfence))s) ∧

(∀s p b.
clause_name "nop" ∧
(*not_blocked s p /\*)
b ≠ Mfence
 ⟹
machine_trans s (Evt p (Barrier b))s) ∧

(∀s p s'.
clause_name "lock" ∧
(s.L = None) ∧
(s.B p = [ ]) ∧
(s' =⟪ R := s.R;
        M := s.M;
        B := s.B;
        L := Some p⟫)
 ⟹
machine_trans s (Lock p)s') ∧

(∀s p s'.
clause_name "unlock" ∧
(s.L = Some p) ∧
(s.B p = [ ]) ∧
(s' =⟪ R := s.R;
        M := s.M;
        B := s.B;
        L := None⟫)
 ⟹
machine_trans s (Unlock p)s')

machine_state_to_state_constraint s =
λl.
case l of
     Location_mem a → s.M a
 ‖ Location_reg p r → s.R p r

machine_final_state path =
if finite path then
Some (machine_state_to_state_constraint (last path))
 else
None

machine_init_state sc =
```

```
⦇ R :=(λp r.sc (LOCATION_REG p r));
  M :=(λa.sc (LOCATION_MEM a));
  B :=(λp.[]);
  L := NONE⦈
```

is_init $s = \exists sc.s =$ machine_init_state sc

The definitions of okEpath and okMpath are given over event machines, in Appendix E.

E Proof Outlines

E.1 Event-annotated machine

We annotate the abstract machine with events to support the equivalence proof between the abstract machine and axiomatic presentations of x86-TSO.

The event abstract machine stores with each memory and register location the event that last wrote there. It also keeps events in the write buffer.

```
evt_machine_state =⦇
                     (* Per processor registers, annotated with the event that last wrote it *)
                     eR : proc → 'reg → (value#'reg event option) option;
                     (* main memory, annotated with the event that last wrote it *)
                     eM : address → (value#'reg event option) option;
                     (* Per processor FIFO write buffers *)
                     eB : proc → 'reg event list;
                     (* Which processor holds the lock *)
                     eL : proc option
                   ⦈
```

evt_no_pending $b\ a = \neg(\exists e.\,\text{MEM}\ e\ b \wedge (\text{loc}\ e = \text{SOME}\ (\text{LOCATION\_MEM}\ a)))$

evt_not_blocked $s\ p = (s.eL = \text{NONE}) \vee (s.eL = \text{SOME}\ p)$

A TAUEVT ew label reports that ew is the event moved from the write buffer to the store. A REVT er ew_opt label does read event er and reports that ew_opt was the last write event on that location. WEVT and BEVT labels both simply perform the given event. The event set argument to UNLOCKE and LOCKE labels is ignored by the machine; it is used only for bookkeeping during a later proof.

```
evt_machine_label =
TAUEVT of 'reg event
| REVT of 'reg event 'reg event option
| WEVT of 'reg event
| BEVT of 'reg event
| LOCKE of proc 'reg event set
| UNLOCKE of proc 'reg event set
```

The transition relation of the event machine is defined by the following rules. Read each clause as "the event-annotated machine has a transition from state s to state s' labelled l if each conjunct preceding $\Longrightarrow$ is satisfied".

```
(∀s a v p er ew_opt.
 clause_name "evt-read-mem" ∧
 evt_not_blocked s p ∧
```

$(\mathsf{proc}\ er = p) \wedge$
$(er.action = \textsc{Access}\ R\ (\textsc{Location\_mem}\ a)v) \wedge$
$(s.eM\ a = \textsc{Some}\ \ (v, ew\_opt)) \wedge$
$\text{evt\_no\_pending}\ (s.eB\ p)a$
$\implies$
$\text{evt\_machine\_trans}\ s\ (\text{REvt}\ er\ ew\_opt)s) \wedge$

$(\forall s\ a\ v\ p\ er\ ew.$
$\text{clause\_name}\ \text{“evt-read-buffer”} \wedge$
$\text{evt\_not\_blocked}\ s\ p \wedge$
$(\mathsf{proc}\ er = p) \wedge$
$(er.action = \textsc{Access}\ R\ (\textsc{Location\_mem}\ a)v) \wedge$
$(ew.action = \textsc{Access}\ \text{W}\ (\textsc{Location\_mem}\ a)v) \wedge$
$(\exists b_1\ b_2.(s.eB\ p = b_1 +\!\!+ [ew] +\!\!+ b_2) \wedge \text{evt\_no\_pending}\ b_1\ a)$
$\implies$
$\text{evt\_machine\_trans}\ s\ (\text{REvt}\ er\ (\textsc{Some}\ \ ew))s) \wedge$

$(\forall s\ r\ v\ p\ er\ ew\_opt.$
$\text{clause\_name}\ \text{“evt-read-reg”} \wedge$
(*evt_not_blocked s p /*)
$(\mathsf{proc}\ er = p) \wedge$
$(er.action = \textsc{Access}\ R\ (\textsc{Location\_reg}\ p\ r)v) \wedge$
$(s.eR\ p\ r = \textsc{Some}\ \ (v, ew\_opt))$
$\implies$
$\text{evt\_machine\_trans}\ s\ (\text{REvt}\ er\ ew\_opt)s) \wedge$

$(\forall s\ a\ v\ p\ ew\ s'.$
$\text{clause\_name}\ \text{“evt-write-buffer”} \wedge$
(*evt_not_blocked s p /*)
$(\mathsf{proc}\ ew = p) \wedge$
$(ew.action = \textsc{Access}\ \text{W}\ (\textsc{Location\_mem}\ a)v) \wedge$
$(s' = \langle\!|\ eR := s.eR;$
$\quad eM := s.eM;$
$\quad eB := (p \mapsto ew \in s.eB\ p)s.eB;$
$\quad eL := s.eL\ |\!\rangle)$
$\implies$
$\text{evt\_machine\_trans}\ s\ (\text{WEvt}\ ew)s') \wedge$

$(\forall s\ a\ v\ p\ ew\ b\ s'.$
$\text{clause\_name}\ \text{“evt-write-mem”} \wedge$
$\text{evt\_not\_blocked}\ s\ p \wedge$
$(\mathsf{proc}\ ew = p) \wedge$
$(ew.action = \textsc{Access}\ \text{W}\ (\textsc{Location\_mem}\ a)v) \wedge$
$(s.eB\ p = b +\!\!+ [ew]) \wedge$
$(s' = \langle\!|\ eR := s.eR;$
$\quad eM := (a \mapsto \textsc{Some}\ \ (v, \textsc{Some}\ \ ew))s.eM;$
$\quad eB := (p \mapsto b)s.eB;$
$\quad eL := s.eL\ |\!\rangle)$
$\implies$
$\text{evt\_machine\_trans}\ s\ (\text{TauEvt}\ ew)s') \wedge$

$(\forall s\ r\ v\ p\ ew\ s'.$
$\text{clause\_name}\ \text{“evt-write-reg”} \wedge$
(*evt_not_blocked s p /*)
$(\mathsf{proc}\ ew = p) \wedge$
$(ew.action = \textsc{Access}\ \text{W}\ (\textsc{Location\_reg}\ p\ r)v) \wedge$
$(s' = \langle\!|\ eR := (p \mapsto (r \mapsto \textsc{Some}\ \ (v, \textsc{Some}\ \ ew))(s.eR\ p))s.eR;$

```
        eM := s.eM;
        eB := s.eB;
        eL := s.eL⟫)
  ⟹
evt_machine_trans s (WEvt ew)s') ∧

(∀s p eb.
clause_name "evt-barrier" ∧
(*evt_not_blocked s p /\*)
(proc eb = p) ∧
(eb.action = Barrier Mfence) ∧
(s.eB p = [ ])
  ⟹
evt_machine_trans s (BEvt eb)s) ∧

(∀s eb.
clause_name "evt-nop" ∧
(*not_blocked s p /\*)
((eb.action = Barrier Sfence) ∨ (eb.action = Barrier Lfence))
  ⟹
evt_machine_trans s (BEvt eb)s) ∧

(∀s p s' es.
clause_name "evt-lock" ∧
(s.eL = None) ∧
(s.eB p = [ ]) ∧
(s' =⟪ eR := s.eR;
        eM := s.eM;
        eB := s.eB;
        eL := Some p⟫)
  ⟹
evt_machine_trans s (LockE p es)s') ∧

(∀s p s' es.
clause_name "evt-unlock" ∧
(s.eL = Some p) ∧
(s.eB p = [ ]) ∧
(s' =⟪ eR := s.eR;
        eM := s.eM;
        eB := s.eB;
        eL := None⟫)
  ⟹
evt_machine_trans s (UnlockE p es)s')


evt_machine_state_to_state_constraint s =
λl.
case l of
    Location_mem a → OPTION_MAP FST (s.eM a)
 || Location_reg p r → OPTION_MAP FST (s.eR p r)
```

A path through the machine should only be considered if it reasonably corresponds to an event structure. The okEpath predicate expresses the necessary conditions. (nth_label and PL are from HOL's library for paths through a LTS. nth_label gets the nth label from the head of the path, and PL returns a (downward closed) set of all numbers that can be used to index into the path, which can be finite or infinite. first returns the first state in a path.)

```
(get_orig_event (REvt e _) = Some e) ∧
```

$(\text{get\_orig\_event}\ (\text{WEvt}\ e) = \text{Some}\ \ e) \wedge$
$(\text{get\_orig\_event}\ (\text{BEvt}\ e) = \text{Some}\ \ e) \wedge$
$(\text{get\_orig\_event}\ \_ = \text{None})$

$\text{locked\_segment}\ \ path\ i\ j\ p =$
$j + 1\ \in\ \text{PL}\ \ path \wedge$
$i < j \wedge$
$(\exists es.\, \text{nth\_label}\ \ i\ path = \text{LockE}\ p\ es) \wedge$
$(\exists es.\, \text{nth\_label}\ \ j\ path = \text{UnlockE}\ p\ es) \wedge$
$(\forall k\ es. i < k \wedge k < j\ \Longrightarrow\ \text{nth\_label}\ \ k\ path \neq \text{UnlockE}\ p\ es)$

$\text{okEpath}\ \ E\ path =$
(* The REvt, WEvt and BEvt labels are exactly the set of events *)
$(E.events = \{e \mid \exists i. i + 1\ \in\ \text{PL}\ \ path \wedge (\text{get\_orig\_event}\ (\text{nth\_label}\ \ i\ path) = \text{Some}\ \ e)\}) \wedge$
(* No REvt, WEvt, or BEvt appears twice as a label *)
$(\forall i\ j\ e_1\ e_2.$
$\quad i + 1\ \in\ \text{PL}\ \ path \wedge j + 1\ \in\ \text{PL}\ \ path \wedge$
$\quad (\text{get\_orig\_event}\ (\text{nth\_label}\ \ i\ path) = \text{Some}\ \ e_1) \wedge (\text{get\_orig\_event}\ (\text{nth\_label}\ \ j\ path) = \text{Some}\ \ e_2) \wedge$
$\quad (e_1 = e_2)$
$\quad \Longrightarrow$
$\quad (i = j)) \wedge$
(* The REvt, WEvt, and BEvt parts of the trace follow po_iico *)
$(\forall (e_1, e_2) \in (\text{po\_iico}\ \ E). \exists i\ j.$
$\quad i < j \wedge j + 1\ \in\ \text{PL}\ \ path \wedge$
$\quad (\text{get\_orig\_event}\ (\text{nth\_label}\ \ i\ path) = \text{Some}\ \ e_1) \wedge (\text{get\_orig\_event}\ (\text{nth\_label}\ \ j\ path) = \text{Some}\ \ e_2)) \wedge$
(* atomic sets of events are properly bracketed by lock/unlock pairs *)
$(\forall es \in (E.atomicity).$
$\quad \exists i\ j\ p.$
$\qquad \text{locked\_segment}\ \ path\ i\ j\ p \wedge$
$\qquad (\{e \mid e\ \in\ es \wedge e\ \in\ \text{mem\_accesses}\ \ E\}$
$\qquad =$
$\qquad \{e \mid \exists k. i < k \wedge k < j \wedge$
$\qquad\qquad (\text{get\_orig\_event}\ (\text{nth\_label}\ \ k\ path) = \text{Some}\ \ e) \wedge$
$\qquad\qquad e\ \in\ \text{mem\_accesses}\ \ E \wedge$
$\qquad\qquad (\mathsf{proc}\ e = p)\}))$

Finally, we require all paths to satisfy the okMpath predicate which ensures the head of the path is in a starting state and that no write event stays in the buffer forever (okpath is part of HOL's path library and simple says that *path* is a path through the given LTS).

$\text{evt\_machine\_init\_state}\ \ sc =$
$(\!|\ eR := (\lambda p\ r.\, \text{OPTION\_MAP}\ (\lambda v.(v, \text{None}))(sc\ (\text{Location\_reg}\ p\ r)));$
$\quad eM := (\lambda a.\, \text{OPTION\_MAP}\ (\lambda v.(v, \text{None}))(sc\ (\text{Location\_mem}\ a)));$
$\quad eB := (\lambda p.[\,]);$
$\quad eL := \text{None}\ |\!)$

$\text{evt\_is\_init}\ \ s = \exists sc. s = \text{evt\_machine\_init\_state}\ \ sc$

$\text{okMpath}\ \ path =$
$\text{evt\_is\_init}\ (\text{first}\ \ path) \wedge$
$\text{okpath}\ \ \text{evt\_machine\_trans}\ \ path \wedge$
$\forall i\ e.$
$i + 1\ \in\ \text{PL}\ \ path \wedge (\text{nth\_label}\ \ i\ path = \text{WEvt}\ e) \wedge \text{is\_mem\_access}\ \ e$
$\Longrightarrow$
$\exists j. j + 1\ \in\ \text{PL}\ \ path \wedge i < j \wedge (\text{nth\_label}\ \ j\ path = \text{TauEvt}\ e)$

To show that the two machines are equivalent, we define how labels and states are erased. erase_label simply removes the annotating information. erase_state is phrased as a relation, because, when erasing the write buffers, their type does not guarantee that an address/value pair can be extracted (although it will be the case for all states on path satisfying okMpath). The annotated_labels function returns the set of all labels that could be annotated versions of its input, given any potential write annotations.

(erase_label (TauEvt _) = Tau) ∧
(erase_label (REvt e _) = Evt (proc e)$e.action$) ∧
(erase_label (WEvt e) = Evt (proc e)$e.action$) ∧
(erase_label (BEvt e) = Evt (proc e)$e.action$) ∧
(erase_label (LockE p es) = Lock p) ∧
(erase_label (UnlockE p es) = Unlock p)

erase_state s s' =
($\forall p\ r.s'.R\ p\ r$ = OPTION_MAP FST ($s.eR\ p\ r$)) ∧
($\forall a.s'.M\ a$ = OPTION_MAP FST ($s.eM\ a$)) ∧
($\forall p.$(LENGTH ($s'.B\ p$) = LENGTH ($s.eB\ p$)) ∧
  $\forall n.n <$ LENGTH ($s.eB\ p$) $\implies$
    $\exists e\ a\ v.$(EL n ($s.eB\ p$) = e) ∧
      (proc e = p) ∧
      ($e.action$ = Access W (Location_mem a)v) ∧
      (EL n ($s'.B\ p$) = (a, v)))) ∧
($s'.L = s.eL$)

(annotated_labels Tau ew e_opt = {TauEvt ew}) ∧
(annotated_labels (Evt p (Access $R\ l\ v$))ew e_opt =
  {REvt e e_opt | e | ($e.action$ = Access $R\ l\ v$) ∧ (p = proc e)}) ∧
(annotated_labels (Evt p (Access W $l\ v$))ew e_opt =
  {WEvt e | ($e.action$ = Access W $l\ v$) ∧ (p = proc e)}) ∧
(annotated_labels (Evt p (Barrier b))ew e_opt =
  {BEvt e | ($e.action$ = Barrier b) ∧ (p = proc e)}) ∧
(annotated_labels (Lock p)ew e_opt =
  {LockE p es | es | **T**}) ∧
(annotated_labels (Unlock p)ew e_opt =
  {UnlockE p es | es | **T**})

The following erasure theorems ensure that the machines are equivalent.

Theorem 5

$\forall sc.$ erase_state (evt_machine_init_state sc)(machine_init_state sc)

Proof sketch (full proof in HOL):
Immediate from the definitions.

□

Theorem 6

$\forall s\ s'.$
erase_state s s'
$\implies$
(evt_machine_state_to_state_constraint s =
machine_state_to_state_constraint s')

Proof sketch (full proof in HOL):
Immediate from the definitions.

□

Theorem 7

$\forall l\ l'\ ew\ er\_opt.$
$l \in \text{annotated\_labels}\ l'\ ew\ er\_opt$
$\implies$
$(l' = \text{erase\_label}\ l)$

Proof sketch (full proof in HOL):
Immediate from the definitions.

□

Theorem 8

$\forall s_1\ l\ s_2\ s_1'.$
$\text{evt\_machine\_trans}\ s_1\ l\ s_2 \wedge$
$\text{erase\_state}\ s_1\ s_1'$
$\implies$
$\exists s_2'.$
$\text{erase\_state}\ s_2\ s_2' \wedge$
$\text{machine\_trans}\ s_1'\ (\text{erase\_label}\ l) s_2'$

Proof sketch (full proof in HOL):
By case analysis on which transition the event machine takes.

□

Theorem 9

$\forall s_1'\ l'\ s_2'\ s_1.$
$\text{machine\_trans}\ s_1'\ l'\ s_2' \wedge$
$\text{erase\_state}\ s_1\ s_1'$
$\implies$
$\exists ew\ ew\_opt.$
$\forall l. l \in \text{annotated\_labels}\ l'\ ew\ ew\_opt \implies$
$\exists s_2.\, \text{erase\_state}\ s_2\ s_2' \wedge \text{evt\_machine\_trans}\ s_1\ l\ s_2$

For this theorem, some of the annotation is "output" coming from the event machine itself: any possible *ew* and *ew_opt* events that should be recorded on TauEvt or REvt labels. However, the use of annotated_labels ensures that the event machine will be able to progress with any other annotating information (such as the event's iiid or eiid).
Proof sketch (full proof in HOL):
By case analysis on which transition the machine takes.

□

E.2 Linear valid executions

The following definition of linear_valid_execution is equivalent to the definition of valid_execution. It differs only by requiring a linear order over memory access events, instead of a partial order, and by omitting two properties that are redundant given a linear ordering (those are: that memory writes have a linear order, and that only finitely many memory reads are unrelated to a same-location memory write).

$\text{linear\_valid\_execution}\ E\ X =$
$\quad \text{linear\_order}\ X.memory\_order\ (\text{mem\_accesses}\ E) \wedge$
$\quad \text{finite\_prefixes}\ X.memory\_order\ (\text{mem\_accesses}\ E) \wedge$
$\quad (\forall er \in (\text{mem\_reads}\ E). \forall e \in (\text{mem\_accesses}\ E). (er, e) \in \text{po\_iico}\ E \implies (er, e) \in X.memory\_order) \wedge$
$\quad (\forall ew_1\ ew_2 \in (\text{mem\_writes}\ E). (ew_1, ew_2) \in \text{po\_iico}\ E \implies (ew_1, ew_2) \in X.memory\_order) \wedge$
$\quad (\forall ew \in (\text{mem\_writes}\ E). \forall er \in (\text{mem\_reads}\ E). \forall ef \in (\text{mfences}\ E).$
$\qquad (ew, ef) \in \text{po\_iico}\ E \wedge (ef, er) \in \text{po\_iico}\ E \implies (ew, er) \in X.memory\_order) \wedge$
$\quad (\forall e_1\ e_2 \in (\text{mem\_accesses}\ E). \forall es \in (E.atomicity).$
$\qquad (e_1 \in es \vee e_2 \in es) \wedge (e_1, e_2) \in \text{po\_iico}\ E$
$\qquad\quad \implies$
$\qquad (e_1, e_2) \in X.memory\_order) \wedge$
$\quad (\forall es \in (E.atomicity). \forall e \in (\text{mem\_accesses}\ E \setminus es).$

$(\forall e' \in (es \cap \text{mem\_accesses } E).(e, e') \in X.memory\_order) \vee$
$(\forall e' \in (es \cap \text{mem\_accesses } E).(e', e) \in X.memory\_order)) \wedge$
$X.rfmap \in \text{reads\_from\_map\_candidates } E \wedge$
$\text{check\_rfmap\_written } E\ X \wedge$
$\text{check\_rfmap\_initial } E\ X$

Theorem 10

$\forall E\ X.\, \text{linear\_valid\_execution } E\ X \implies \text{valid\_execution } E\ X$

Proof sketch (full proof in HOL):
Immediate from the definitions.

□

Theorem 11

$\forall E\ X\ memory\_order'.$
$\text{valid\_execution } E\ X \wedge$
$X.memory\_order \subseteq memory\_order' \wedge$
$\text{linear\_order } memory\_order' (\text{mem\_accesses } E) \wedge$
$\text{finite\_prefixes } memory\_order' (\text{mem\_accesses } E) \wedge$
$(\forall er \in (\text{mem\_reads } E).\forall ew \in (\text{mem\_writes } E).$
$(\text{loc } er = \text{loc } ew) \wedge (ew, er) \in memory\_order' \implies (ew, er) \in X.memory\_order)$
$\implies$
$\text{linear\_valid\_execution } E\ (X \oplus memory\_order := memory\_order')$

This theorem says that any linearization of memory order is still a valid memory ordering, as long as it obeys the following additional condition. If ew is a memory write and er is a memory read from the same location, then if the original memory ordering does not relate ew and er, the linearization must make er come before ew. Otherwise, in the linearization, er might see ew's write, but this should only happen if ew precedes er in the original memory order.
Proof sketch (full proof in HOL):
By unfolding all of the relevant definitions.

□

Theorem 12

$\forall E\ X.$
$\text{well\_formed\_event\_structure } E \wedge$
$\text{valid\_execution } E\ X$
$\implies$
$\exists memory\_order'.$
$X.memory\_order \subseteq memory\_order' \wedge$
$\text{linear\_order } memory\_order' (\text{mem\_accesses } E) \wedge$
$\text{finite\_prefixes } memory\_order' (\text{mem\_accesses } E) \wedge$
$(\forall er \in (\text{mem\_reads } E).\forall ew \in (\text{mem\_writes } E).$
$(\text{loc } er = \text{loc } ew) \wedge (ew, er) \in memory\_order' \implies (ew, er) \in X.memory\_order)$

Proof sketch (full proof in HOL):
Define the following relation, complete_memory_order, that extends memory order with any missing same-location memory read to memory write dependencies. Note that the definition below also adds in enough other edges to remain transitive.

$\text{complete\_memory\_order } E\ memory\_order =$
$memory\_order \cup$
$\{(e_1, e_2) \mid \exists ew\ er.(ew, er) \notin memory\_order \wedge (er, ew) \notin memory\_order \wedge$
$ew \in \text{mem\_writes } E \wedge er \in \text{mem\_reads } E \wedge (\text{loc } ew = \text{loc } er) \wedge$
$(e_1, er) \in memory\_order \wedge (ew, e_2) \in memory\_order\}$

We have the following theorem (proved in HOL) that allows us to linearize partial orders, maintaining the finite prefixes property.

$\forall r\ s.\,\text{countable}\ s \wedge \text{partial\_order}\ r\ s \wedge \text{finite\_prefixes}\ r\ s \implies \exists r'.\,\text{linear\_order}\ r'\ s \wedge \text{finite\_prefixes}\ r'\ s \wedge r \subseteq r'$

The rest of the theorem proceeds by unfolding definitions, noting that valid_execution requires $X.memory\_order$ to have finite prefixes, which allows us to prove that complete_memory_order has finite prefixes as well.

□

E.3 Abstract machine model/axiomatic model equivalence

E.3.1 Abstract machine validity

The path_to_X function produces an execution witness whose (linear) memory ordering comes from the labels of its input path, whose reads-from map comes from the annotations on the labels, and whose initial state corresponds to the first state in the path.

```
(get_mem_event (TauEvt e) = Some e) ∧
(get_mem_event (REvt e _) =
  if is_mem_access e then
    Some e
  else
    None) ∧
(get_mem_event _ = None)


path_to_X path =
⟦ memory_order :=
  {(e1, e2) |
  ∃j i l1 l2.
    j + 1 ∈ PL path ∧ i ≤ j ∧
    (nth_label i path = l1) ∧ (nth_label j path = l2) ∧
    (get_mem_event l1 = Some e1) ∧ (get_mem_event l2 = Some e2)};
  rfmap :={(ew, er) | (ew, er) | ∃i.i + 1 ∈ PL path ∧ (nth_label i path = REvt er (Some ew))};
  initial_state := evt_machine_state_to_state_constraint (first path)⟧
```

Theorem 13

```
∀E path.
well_formed_event_structure E ∧
okEpath E path ∧
okMpath path
 ⟹
linear_valid_execution E (path_to_X path)
```

Proof sketch (full proof in HOL):

By unfolding the definitions in linear_valid_execution and okEpath, and by using the following theorems that express how the labels of a valid path through the lts relate to each other. Each of these theorems is a "safety" style theorem that is proved (in HOL) by induction on the location of the largest label index mentioned. Some of them require nested inductions, and some rely on the being no duplicate writes (which okEpath ensures).

```
no_dup_writes path =
∀i j ew.
SUC i ∈ PL path ∧
SUC j ∈ PL path ∧
(nth_label i path = WEvt ew) ∧
(nth_label j path = WEvt ew)
 ⟹
```

$(i = j)$

Each TauEvt comes from a WEvt.

Theorem 14

$\forall path\ j\ e.$
okMpath $path \wedge$
$j + 1 \in \text{PL}\ path \wedge$
$(\text{nth\_label}\ j\ path = \text{TauEvt}\ e)$
$\implies$
$\exists i. i < j \wedge (\text{nth\_label}\ i\ path = \text{WEvt}\ e) \wedge \text{is\_mem\_access}\ e$

Each same processor pair of TauEvts comes from a pair of WEvts in order.

Theorem 15

$\forall path\ i\ j\ ew_1\ ew_2.$
okMpath $path \wedge$
$j + 1 \in \text{PL}\ path \wedge$
$i < j \wedge$
$(\mathsf{proc}\ ew_1 = \mathsf{proc}\ ew_2) \wedge$
$(\text{nth\_label}\ i\ path = \text{TauEvt}\ ew_1) \wedge$
$(\text{nth\_label}\ j\ path = \text{TauEvt}\ ew_2)$
$\implies$
$\exists k\ l. k < i \wedge l < j \wedge k < l \wedge$
$(\text{nth\_label}\ k\ path = \text{WEvt}\ ew_1) \wedge (\text{nth\_label}\ l\ path = \text{WEvt}\ ew_2)$

REvt labels have reads on them.

Theorem 16

$\forall path\ i\ e_1\ e_2.$
okMpath $path \wedge$
$i + 1 \in \text{PL}\ path \wedge$
$(\text{nth\_label}\ i\ path = \text{REvt}\ e_1\ e_2)$
$\implies$
$\exists l\ v. e_1.action = \text{Access}\ R\ l\ v$

WEvt labels have writes on them.

Theorem 17

$\forall path\ i\ e.$
okMpath $path \wedge$
$i + 1 \in \text{PL}\ path \wedge$
$(\text{nth\_label}\ i\ path = \text{WEvt}\ e)$
$\implies$
$\exists l\ v. e.action = \text{Access W}\ l\ v$

BEvt labels have barriers on them.

Theorem 18

$\forall path\ i\ e.$
okMpath $path \wedge$
$i + 1 \in \text{PL}\ path \wedge$
$(\text{nth\_label}\ i\ path = \text{BEvt}\ e)$
$\implies$
$\exists f. e.action = \text{Barrier}\ f$

Each same processor pair of WEvt labels is followed by a pair of TauEvt labels in order. This relies on the liveness property in okMpath.

Theorem 19

$\forall path\ i\ j\ ew_1\ ew_2.$
okMpath $path \wedge$
$j+1 \in \text{PL}\ path \wedge$
$i < j \wedge$
$(\mathsf{proc}\ ew_1 = \mathsf{proc}\ ew_2) \wedge$
$(\text{nth\_label}\ i\ path = \text{WEvt}\ ew_1) \wedge$
$(\text{nth\_label}\ j\ path = \text{WEvt}\ ew_2) \wedge$
is_mem_access $ew_1 \wedge$
is_mem_access $ew_2 \wedge$
no_dup_writes $path$
$\implies$
$\exists k\ l. l+1 \in \text{PL}\ path \wedge k < l \wedge (\text{nth\_label}\ k\ path = \text{TauEvt}\ ew_1) \wedge (\text{nth\_label}\ l\ path = \text{TauEvt}\ ew_2)$

TauEvts occur before a label that would require an empty buffer.

Theorem 20

$\forall path\ i\ j\ ef\ ew\ p\ es.$
okMpath $path \wedge$
$j+1 \in \text{PL}\ path \wedge$
$i < j \wedge$
$(((\text{nth\_label}\ j\ path = \text{BEvt}\ ef) \wedge (\mathsf{proc}\ ef = \mathsf{proc}\ ew) \wedge (ef.action = \text{Barrier Mfence})) \vee$
$(\text{nth\_label}\ j\ path = \text{UnlockE}\ (\mathsf{proc}\ ew) es) \vee$
$(\text{nth\_label}\ j\ path = \text{LockE}\ (\mathsf{proc}\ ew) es)) \wedge$
$(\text{nth\_label}\ i\ path = \text{WEvt}\ ew) \wedge$
is_mem_access ew
$\implies$
$\exists k. k < j \wedge (\text{nth\_label}\ k\ path = \text{TauEvt}\ ew)$

A REvt label reads from the most recent TauEvt label if there are no intermediate same processor, same location WEvt labels. Otherwise it reads from the most recent of those.

Theorem 21

$\forall path\ j\ er\ ew.$
okMpath $path \wedge$
$j+1 \in \text{PL}\ path \wedge$
no_dup_writes $path \wedge$
$(\text{nth\_label}\ j\ path = \text{REvt}\ er\ (\text{Some}\ ew))$
$\implies$
$(\text{loc}\ er = \text{loc}\ ew) \wedge$
$(\text{value\_of}\ er = \text{value\_of}\ ew) \wedge$
$(((\mathsf{proc}\ er = \mathsf{proc}\ ew) \wedge$
$\exists i.$
$i < j \wedge$
$(\text{nth\_label}\ i\ path = \text{WEvt}\ ew) \wedge$
$(\forall k. i < k \wedge k < j \implies \text{nth\_label}\ k\ path \neq \text{TauEvt}\ ew) \wedge$
$(\forall k\ ew'. i < k \wedge k < j \wedge (\text{nth\_label}\ k\ path = \text{WEvt}\ ew') \wedge (\mathsf{proc}\ er = \mathsf{proc}\ ew') \implies$
$\text{loc}\ ew \neq \text{loc}\ ew')) \vee$
$(\exists i.$
$i < j \wedge$
$(\text{nth\_label}\ i\ path = \text{TauEvt}\ ew) \wedge$
$(\forall k\ ew'. k < j \wedge (\text{nth\_label}\ k\ path = \text{WEvt}\ ew') \wedge (\mathsf{proc}\ er = \mathsf{proc}\ ew') \wedge$
$(\text{loc}\ ew' = \text{loc}\ er) \implies$
$\exists l. k < l \wedge l < j \wedge (\text{nth\_label}\ l\ path = \text{TauEvt}\ ew')) \wedge$
$(\forall k\ ew'. i < k \wedge k < j \wedge (\text{nth\_label}\ k\ path = \text{TauEvt}\ ew') \implies \text{loc}\ ew \neq \text{loc}\ ew')))$

A REvt that reads from the initial state is not preceded by write labels to its location.

Theorem 22

$\forall path\ i\ er.$
okMpath $path \wedge$
$i + 1 \in \text{PL}\ path \wedge$
(nth_label $i\ path = \text{REvt}\ er\ \text{None}$)
$\Longrightarrow$
(value_of $er = \text{evt\_machine\_state\_to\_state\_constraint}\ (\text{first}\ path)(\textbf{the}\ (\text{loc}\ er))) \wedge$
$\forall j\ ew.$
$j < i \wedge$
$((\text{nth\_label}\ j\ path = \text{TauEvt}\ ew) \vee ((\text{nth\_label}\ j\ path = \text{WEvt}\ ew) \wedge (\mathsf{proc}\ er = \mathsf{proc}\ ew)))$
$\Longrightarrow$
$\text{loc}\ ew \neq \text{loc}\ er$

Memory read and write labels inside of a locked segment are on the processor that is locked.

Theorem 23

$\forall path\ i\ j\ k\ p\ e\ e'.$
okMpath $path \wedge$
locked_segment $path\ i\ k\ p \wedge$
$i < j \wedge$
$j < k \wedge$
is_mem_access $e \wedge$
$((\text{nth\_label}\ j\ path = \text{REvt}\ e\ e') \vee$
$(\text{nth\_label}\ j\ path = \text{TauEvt}\ e))$
$\Longrightarrow$
$(\mathsf{proc}\ e = p)$

WEvt and TauEvt labels with the same write occur atomically with respect to locks on the write's processor.

Theorem 24

$\forall path\ i\ j\ ew.$
okMpath $path \wedge$
locked_segment $path\ i\ j\ (\mathsf{proc}\ ew) \wedge$
is_mem_access ew
$\Longrightarrow$
$((\exists k.i < k \wedge k < j \wedge (\text{nth\_label}\ k\ path = \text{WEvt}\ ew)) =$
$(\exists k.i < k \wedge k < j \wedge (\text{nth\_label}\ k\ path = \text{TauEvt}\ ew)))$

□

E.3.2 Abstract machine completeness

The proof that the abstract machine can perform any valid execution is currently partly in HOL and partly by hand. We intend to fully mechanize the parts that currently by hand.

We first need several additional definitions for grouping and operating on sets of labels.

memL $E\ X =$
$\{\text{TauEvt}\ e \mid e \in \text{mem\_writes}\ E\} \cup$
$\{\text{REvt}\ er\ \text{None} \mid er \in \text{mem\_reads}\ E \wedge er \notin \text{range}\ X.rfmap\} \cup$
$\{\text{REvt}\ er\ (\text{Some}\ ew) \mid er \in \text{mem\_reads}\ E \wedge (ew, er) \in X.rfmap\}$

localL $E\ X =$
$\{\text{REvt}\ er\ \text{None} \mid er \in \text{reads}\ E \wedge er \notin \text{range}\ X.rfmap\} \cup$
$\{\text{REvt}\ er\ (\text{Some}\ ew) \mid er \in \text{reads}\ E \wedge (ew, er) \in X.rfmap\} \cup$
$\{\text{WEvt}\ e \mid e \in \text{writes}\ E\} \cup$
$\{\text{BEvt}\ e \mid e \in \text{fences}\ E\}$

proc_es $es = \{\mathsf{proc}\ e \mid e \in es\}$

$$
\begin{array}{l}
\text{lockL } E\ X = \\
\{\textsc{LockE}\ p\ es \mid es \in E.atomicity \wedge (p \in \text{proc\_es } es)\} \cup \\
\{\textsc{UnlockE}\ p\ es \mid es \in E.atomicity \wedge (p \in \text{proc\_es } es)\}
\end{array}
$$

$$
\begin{array}{l}
\text{allL } E\ X = \\
\text{memL } E\ X \cup \\
\text{localL } E\ X \cup \\
\text{lockL } E\ X
\end{array}
$$

$$
\begin{array}{l}
(\text{l\_e } (\textsc{TauEvt}\ e) = \textsc{Some}\ e) \wedge \\
(\text{l\_e } (\textsc{REvt}\ e\ \_) = \textsc{Some}\ e) \wedge \\
(\text{l\_e } (\textsc{WEvt}\ e) = \textsc{Some}\ e) \wedge \\
(\text{l\_e } (\textsc{BEvt}\ e) = \textsc{Some}\ e) \wedge \\
(\text{l\_e } \_ = \textsc{None})
\end{array}
$$

$$
\begin{array}{l}
(\text{l\_es } (\textsc{LockE}\ p\ es) = \textsc{Some}\ es) \wedge \\
(\text{l\_es } (\textsc{UnlockE}\ p\ es) = \textsc{Some}\ es) \wedge \\
(\text{l\_es } \_ = \textsc{None})
\end{array}
$$

Our overall strategy is to characterize an ordering on labels label_order such that any sequence of labels that follows this ordering has a path through the machine (that is also an okEpath and okMpath). We then show that the properties in linear_valid_execution ensure that such a sequence exists. The first part of the proof is not currently mechanized in HOL, but is proved by hand. We have proved in HOL that there exists a partial ordering over labels lo such that label_order $\subseteq$ lo (which via hand proof has finite prefixes), and therefore there exists a sequence consistent with label_order (via the linearization theorem mentioned in Section E.2.

$$
\begin{array}{l}
\text{label\_order } E\ X = \\
\{(l, l') \mid l \in \text{memL } E\ X \wedge l' \in \text{memL } E\ X \wedge \\
\quad (\textbf{the } (\text{l\_e } l), \textbf{the } (\text{l\_e } l')) \in X.memory\_order\} \cup \\
\{(l, l') \mid l \in \text{localL } E\ X \wedge l' \in \text{localL } E\ X \wedge \\
\quad (\textbf{the } (\text{l\_e } l), \textbf{the } (\text{l\_e } l')) \in \text{po\_iico } E\} \cup \\
\{(\textsc{WEvt}\ e, \textsc{TauEvt}\ e) \mid \textsc{WEvt}\ e \in \text{allL } E\ X \wedge \textsc{TauEvt}\ e \in \text{allL } E\ X\} \cup \\
\{(\textsc{TauEvt}\ e, \textsc{BEvt}\ e') \mid \textsc{TauEvt}\ e \in \text{allL } E\ X \wedge \textsc{BEvt}\ e' \in \text{allL } E\ X \wedge \\
\quad e' \in \text{mfences } E \wedge (e, e') \in \text{po\_iico } E\} \cup \\
\{(\textsc{TauEvt}\ e, \textsc{LockE}\ p\ es) \mid \textsc{TauEvt}\ e \in \text{allL } E\ X \wedge \textsc{LockE}\ p\ es \in \text{allL } E\ X \wedge \\
\quad e \notin es \wedge \exists e'.e' \in es \wedge e' \in \text{mem\_accesses } E \wedge (e, e') \in \text{po\_iico } E\} \cup \\
\{(\textsc{LockE}\ p\ es, l) \mid \textsc{LockE}\ p\ es \in \text{allL } E\ X \wedge l \in \text{allL } E\ X \wedge \\
\quad \exists e.(\text{l\_e } l = \textsc{Some}\ e) \wedge e \in es \wedge e \in \text{mem\_accesses } E\} \cup \\
\{(l, \textsc{UnlockE}\ p\ es) \mid l \in \text{memL } E\ X \wedge \textsc{UnlockE}\ p\ es \in \text{allL } E\ X \wedge \\
\quad \exists e.(\text{l\_e } l = \textsc{Some}\ e) \wedge e \in es\} \cup \\
\{(\textsc{UnlockE}\ p\ es, l) \mid \textsc{UnlockE}\ p\ es \in \text{allL } E\ X \wedge l \in \text{allL } E\ X \wedge \\
\quad \exists e\ e'.(\text{l\_e } l = \textsc{Some}\ e) \wedge e' \in es \wedge e \notin es \wedge \\
\qquad (e', e) \in X.memory\_order \wedge \\
\qquad (l \in \text{memL } E\ X \vee (e \in \text{mem\_accesses } E \wedge (\mathsf{proc}\ e = p)))\} \cup \\
\{(l, \textsc{LockE}\ p\ es) \mid l \in \text{memL } E\ X \wedge \textsc{LockE}\ p\ es \in \text{allL } E\ X \wedge \\
\quad \exists e\ e'.(\text{l\_e } l = \textsc{Some}\ e) \wedge e' \in es \wedge e \notin es \wedge \\
\qquad (e, e') \in X.memory\_order\} \cup \\
\{(\textsc{UnlockE}\ p\ es, \textsc{LockE}\ p'\ es') \mid \textsc{UnlockE}\ p\ es \in \text{allL } E\ X \wedge \textsc{LockE}\ p'\ es' \in \text{allL } E\ X \wedge \\
\quad es \neq es' \wedge \exists e\ e'.e \in es \wedge e' \in es' \wedge (e, e') \in X.memory\_order\} \cup \\
\{(\textsc{WEvt}\ e, \textsc{WEvt}\ e') \mid \textsc{WEvt}\ e \in \text{allL } E\ X \wedge \textsc{WEvt}\ e' \in \text{allL } E\ X \wedge \\
\quad (e, e') \in X.memory\_order \wedge (\mathsf{proc}\ e = \mathsf{proc}\ e')\}
\end{array}
$$

lo1–lo4 correspond to clauses of label_order, and lo_events composes them to form a partial order. The proof that lo_events is a transitive and antisymmetric uses several lemmas that collapse particular relation compositions into shorter ones, and then relies on the transitivity and antisymmetry of the underlying memory and program orders. For example,

$\forall E\ X\ l_1\ l_2\ l_3\ l_4.\ \text{well\_formed\_event\_structure}\ E \wedge \text{linear\_valid\_execution}\ E\ X \wedge (l_1, l_2) \in \text{lo4}\ E\ X \wedge (l_2, l_3) \in \text{lo2}\ E\ X \wedge (l_3, l_4) \in \text{lo3}\ E\ X \implies (l_1, l_4) \in \text{lo1}\ E\ X$

$\text{lo1}\ E\ X =$
$\{(l, l') \mid l \in \text{memL}\ E\ X \wedge l' \in \text{memL}\ E\ X \wedge$
$\quad (\textbf{the}\ (\text{l\_e}\ l), \textbf{the}\ (\text{l\_e}\ l')) \in X.\mathit{memory\_order}\}$

$\text{lo2}\ E\ X =$
$\{(l, l') \mid l \in \text{localL}\ E\ X \wedge l' \in \text{localL}\ E\ X \wedge$
$\quad (\textbf{the}\ (\text{l\_e}\ l), \textbf{the}\ (\text{l\_e}\ l')) \in \text{po\_iico}\ E\} \cup$
$\{(\text{WEvt}\ e, \text{WEvt}\ e') \mid \text{WEvt}\ e \in \text{allL}\ E\ X \wedge \text{WEvt}\ e' \in \text{allL}\ E\ X \wedge$
$\qquad (e, e') \in X.\mathit{memory\_order} \wedge (\mathsf{proc}\ e = \mathsf{proc}\ e')\} \cup$
$\{(l, \text{WEvt}\ e) \mid l \in \text{localL}\ E\ X \wedge \text{WEvt}\ e \in \text{allL}\ E\ X \wedge$
$\qquad \exists e'.\text{WEvt}\ e' \in \text{localL}\ E\ X \wedge (\textbf{the}\ (\text{l\_e}\ l), e') \in \text{po\_iico}\ E \wedge$
$\qquad (e', e) \in X.\mathit{memory\_order} \wedge (\mathsf{proc}\ e' = \mathsf{proc}\ e)\}$

$\text{lo3}\ E\ X =$
$\{(\text{WEvt}\ e, \text{TauEvt}\ e) \mid \text{WEvt}\ e \in \text{allL}\ E\ X \wedge \text{TauEvt}\ e \in \text{allL}\ E\ X\}$

$\text{lo4}\ E\ X =$
$\{(\text{TauEvt}\ e, \text{BEvt}\ e') \mid \text{TauEvt}\ e \in \text{allL}\ E\ X \wedge \text{BEvt}\ e' \in \text{allL}\ E\ X \wedge$
$\quad e' \in \text{mfences}\ E\ X \wedge (e, e') \in \text{po\_iico}\ E\}$

$\text{lo\_events}\ E\ X =$
$\text{lo1}\ E\ X \cup \text{lo2}\ E\ X \cup$
$\text{lo1}\ E\ X \circ \text{lo2}\ E\ X \cup$
$\text{lo2}\ E\ X \circ \text{lo1}\ E\ X \cup$
$\text{lo2}\ E\ X \circ \text{lo1}\ E\ X \circ \text{lo2}\ E\ X \cup$
$\text{lo2}\ E\ X \circ \text{lo3}\ E\ X \circ \text{lo1}\ E\ X \cup$
$\text{lo2}\ E\ X \circ \text{lo3}\ E\ X \circ \text{lo1}\ E\ X \circ \text{lo2}\ E\ X \cup$
$\text{lo1}\ E\ X \circ \text{lo4}\ E\ X \circ \text{lo2}\ E\ X \cup$
$\text{lo2}\ E\ X \circ \text{lo1}\ E\ X \circ \text{lo4}\ E\ X \circ \text{lo2}\ E\ X \cup$
$\text{lo2}\ E\ X \circ \text{lo3}\ E\ X \circ \text{lo1}\ E\ X \circ \text{lo4}\ E\ X \circ \text{lo2}\ E\ X$

lo5 defines a partial order over just the lock labels, using the above proof technique of collapsing relational compositions.

$\text{lo5}\ E\ X =$
$\{(\text{LockE}\ p\ es, \text{LockE}\ p\ es) \mid \text{LockE}\ p\ es \in \text{lockL}\ E\ X\} \cup$
$\{(\text{UnlockE}\ p\ es, \text{UnlockE}\ p\ es) \mid \text{LockE}\ p\ es \in \text{lockL}\ E\ X\} \cup$
$\{(l1, l2) \mid l1 \in \text{lockL}\ E\ X \wedge l2 \in \text{lockL}\ E\ X \wedge$
$\qquad \textbf{the}\ (\text{l\_es}\ l1) \neq \textbf{the}\ (\text{l\_es}\ l2) \wedge$
$\qquad \exists e_1\ e_2.e_1 \in \textbf{the}\ (\text{l\_es}\ l1) \wedge e_2 \in \textbf{the}\ (\text{l\_es}\ l2) \wedge$
$\qquad\quad (e_1, e_2) \in X.\mathit{memory\_order}\} \cup$
$\{(\text{LockE}\ p\ es, \text{UnlockE}\ p\ es) \mid \text{LockE}\ p\ es \in \text{lockL}\ E\ X \wedge \text{UnlockE}\ p\ es \in \text{lockL}\ E\ X\}$

lo6–lo8 express how lock labels relate to other labels, to enforce the atomicity guarantees required of a locked instruction.

$\text{lo6}\ E\ X =$
$\{(\text{LockE}\ p\ es, l) \mid \text{LockE}\ p\ es \in \text{allL}\ E\ X \wedge l \in \text{allL}\ E\ X \wedge$
$\qquad \exists e.(\text{l\_e}\ l = \text{Some}\ e) \wedge e \in es \wedge e \in \text{mem\_accesses}\ E\}$

$\text{lo7}\ E\ X =$
$\{(l, \text{UnlockE}\ p\ es) \mid l \in \text{memL}\ E\ X \wedge \text{UnlockE}\ p\ es \in \text{allL}\ E\ X \wedge$
$\qquad \exists e.(\text{l\_e}\ l = \text{Some}\ e) \wedge e \in es\}$

$$\begin{array}{l}\text{lo8 } E\ X = \\ \{(\textsc{UnlockE}\ p\ es, l) \mid \textsc{UnlockE}\ p\ es\ \in\ \text{allL}\ E\ X \wedge l\ \in\ \text{allL}\ E\ X \wedge \\ \qquad \exists e\ e'.(\text{l\_e}\ l = \textsc{Some}\ e) \wedge e'\ \in\ es \wedge e\ \notin\ es \wedge \\ \qquad\qquad (e', e) \in\ X.memory\_order \wedge \\ \qquad\qquad (l\ \notin\ \text{memL}\ E\ X\ \Longrightarrow\ ((\mathsf{proc}\ e = p) \wedge e\ \in\ \text{mem\_accesses}\ E))\}\end{array}$$

$$\text{lo68 } E\ X = \text{lo6 } E\ X\ \cup\ \text{lo8 } E\ X$$

$$\text{lo79 } E\ X = \text{lo7 } E\ X\ \cup\ \text{lo9 } E\ X$$

Finally, lo defines a partial order over all labels in allL $E\ X$.

$$\begin{array}{l}\text{lo } E\ X = \\ \text{lo\_events } E\ X\ \cup \\ \text{lo5 } E\ X\ \cup \\ \text{lo5 } E\ X \circ \text{lo68 } E\ X \circ \text{lo\_events } E\ X \cup \\ \text{lo5 } E\ X \circ \text{lo68 } E\ X \circ \text{lo\_events } E\ X \cup \\ \text{lo\_events } E\ X \circ \text{lo79 } E\ X \circ \text{lo5 } E\ X \cup \\ \text{lo\_events } E\ X \circ \text{lo79 } E\ X \circ \text{lo5 } E\ X \circ \text{lo68 } E\ X \circ \text{lo\_events } E\ X\end{array}$$

Theorem 25

$$\begin{array}{l}\forall E\ X. \\ \text{well\_formed\_event\_structure } E \wedge \text{linear\_valid\_execution } E\ X \\ \Longrightarrow \\ \text{partial\_order (lo } E\ X)(\text{allL } E\ X)\end{array}$$

Proof sketch (full proof in HOL): The proof follows the methodology introduced above of collapsing certain relation compositions to other, shorter ones. The most intricate cases involve showing that there is no cycle formed between lo_events and lo6–lo9, since lock labels are not directly apparent in the execution witness. □

Theorem 26

$$\begin{array}{l}\forall E\ X. \\ \text{well\_formed\_event\_structure } E \wedge \text{linear\_valid\_execution } E\ X \\ \Longrightarrow \\ \text{label\_order } E\ X\ \subseteq\ \text{lo } E\ X\end{array}$$

Proof sketch (full proof in HOL): Immediate by expanding the definitions. □

Theorem 27

$$\begin{array}{l}\forall E\ X.\ \text{well\_formed\_event\_structure } E \wedge \text{linear\_valid\_execution } E\ X \\ \Longrightarrow \\ \exists path.\ \text{okEpath } E\ path \wedge \text{okMpath } path\end{array}$$

Proof sketch: Create a sequence from label_order and assume that the machine has gone i steps through the sequence, and show that it can take one more by cases on the next label. □

E.4 Executable checker

The executable checker essentially replaces sets with lists and quantification with iteration over the lists.

$$\begin{array}{l}\mathsf{ch\_event\_structure} = \langle\![\ ch\_procs : \mathsf{proc\ list}; \\ \qquad\qquad\qquad ch\_events : (\prime reg\ \mathsf{event})\mathsf{list}; \\ \qquad\qquad\qquad ch\_intra\_causality : (\prime reg\ \mathsf{event})\mathsf{ch\_reln}; \\ \qquad\qquad\qquad ch\_atomicity : (\prime reg\ \mathsf{event})\mathsf{list\ list}]\!\rangle\end{array}$$

$$\mathsf{ch\_execution\_witness} = \langle\![\ (*\ \text{the memory order is the transitive closure of the pairs in ch\_memory\_order}\ *)$$

```
                    ch_memory_order : ('reg event)ch_reln;
                    ch_rfmap : 'reg event → 'reg event  option;
                    ch_initial_state : 'reg location → value  option⟫

cis_mem_access e =
case e.action of
  ACCESS d (LOCATION_MEM a)v → T
| _ → F

is_mem_read e =
case e.action of
  ACCESS R (LOCATION_MEM a)v → T
| _ → F

is_mem_write e =
case e.action of
  ACCESS W (LOCATION_MEM a)v → T
| _ → F

is_read e =
case e.action of
  ACCESS R l v → T
| _ → F

is_write e =
case e.action of
  ACCESS W l v → T
| _ → F

is_barrier e =
case e.action of
  BARRIER MFENCE → T
| _ → F

check_po_iico intra e1 e2 =
if proc e1 = proc e2 then
  if e1.iiid.poi < e2.iiid.poi then
    T
  else if e1.iiid.poi = e2.iiid.poi then
    e1 ≠ e2 ∧ MEM (e1, e2)intra
  else
    F
else
  F

check_po_iico_in_mo intra mo e1 e2 =
if check_po_iico intra e1 e2 then
  MEM (e1, e2)mo
else
  T

barrier_separated intra barriers e1 e2 =
```

```
(proc e1 = proc e2) ∧
EXISTS (λeb. check_po_iico intra e1 eb ∧ check_po_iico intra eb e2)
       barriers


previous_writes1 er r =
MAP FST (FILTER (λ(ew, er').(er' = er) ∧ is_write ew ∧ (loc ew = loc er))r)


previous_writes2 er intra es =
FILTER (λew.(loc ew = loc er) ∧ check_po_iico intra ew er)es


check_maximal1 x xs r =
MEM x xs ∧
EVERY (λx'.if x ≠ x' then ¬(MEM (x, x')r) else T)xs


check_maximal2 x xs intra =
MEM x xs ∧
EVERY (λx'.if x ≠ x' then ¬(check_po_iico intra x x') else T)xs


(cross[]_ = []) ∧
(cross ((x, y) ∈ r)r' = MAP (λ(x', y').(x, y'))r' ++ cross r r')


tinsert (x, y)r =
let left = FILTER (λ(x', y').y' = x)r in
let right = FILTER (λ(x', y').x' = y)r in
(x, y) ∈ r ++
MAP (λ(x', y').(x', y))left ++
MAP (λ(x', y').(x, y'))right ++
cross left right


(tclose[]acc = acc) ∧
(tclose ((x, y) ∈ r)acc = tclose r (tinsert (x, y)acc))


check_valid_execution E X =
let mo = tclose (FILTER (λ(e1, e2).e1 ≠ e2)X.ch_memory_order)[] in
let writes = FILTER is_write E.ch_events in
let reads = FILTER is_read E.ch_events in
let mwrites = FILTER is_mem_write writes in
let mreads = FILTER is_mem_read reads in
let barriers = FILTER is_barrier E.ch_events in
let intra = tclose E.ch_intra_causality[] in
(* partial order *)
EVERY (λ(e1, e2).e1 ≠ e2)mo ∧
EVERY (λ(e1, e2).cis_mem_access e1 ∧ cis_mem_access e2 ∧
                 MEM e1 E.ch_events ∧ MEM e2 E.ch_events)X.ch_memory_order ∧
(* linear order on mwrites *)
EVERY (λe1. EVERY (λe2.if e1 ≠ e2 then
                         MEM (e1, e2)mo ∨ MEM (e2, e1)mo
                       else
                         T)
                  mwrites)
       mwrites ∧
(* po_iico in memory_order *)
EVERY (λer1. EVERY (λer2. check_po_iico_in_mo intra mo er1 er2)mreads)
```

```
        mreads ∧
EVERY (λer. EVERY (λew. check_po_iico_in_mo intra mo er ew)mwrites)
        mreads ∧
EVERY (λew1. EVERY (λew2. check_po_iico_in_mo intra mo ew1 ew2)mwrites)
        mwrites ∧
EVERY (λew.
  EVERY (λer.
    if barrier_separated intra barriers ew er ∨
         EXISTS (λes. MEM ew es ∨ MEM er es)E.ch_atomicity then
       check_po_iico_in_mo intra mo ew er
     else
       T)
    mreads)
  mwrites ∧
(* atomicity *)
EVERY (λes.
  EVERY (λe.
    if ¬(MEM e es) then
      EVERY (λe'.
        if is_mem_read e' ∨ is_mem_write e' then
          MEM (e, e')mo
         else
          T)
        es ∨
      EVERY (λe'.
        if is_mem_read e' ∨ is_mem_write e' then
          MEM (e', e)mo
         else
          T)
        es
     else
       T)
    (mreads ++ mwrites))
  E.ch_atomicity ∧
  (* rfmc *)
  EVERY (λer.
    case X.ch_rfmap er of
      SOME ew →
        is_read er ∧ is_write ew ∧ MEM ew E.ch_events ∧
        (loc er = loc ew) ∧ (value_of er = value_of ew)
    ‖ NONE → T)
    E.ch_events ∧
  (* rfmap written and initial*)
  EVERY (λer.
    case X.ch_rfmap er of
      SOME ew →
        if is_mem_write ew then
          check_maximal1 ew (previous_writes1 er mo ++
                                 previous_writes2 er intra writes)mo
         else
          check_maximal2 ew (previous_writes2 er intra writes)intra
    ‖ NONE →
        (case loc er of
            SOME l →
              (value_of er = X.ch_initial_state l) ∧
              (previous_writes1 er mo = [ ]) ∧
              (previous_writes2 er intra writes = [ ])
```

```
                || NONE → F))
      reads

check_set_eq es1 es2 = EVERY (λe. MEM e es2) es1 ∧ EVERY (λe. MEM e es1) es2

check_well_formed_event_structure E =
let intra = tclose (FILTER (λ(e1, e2). e1 ≠ e2) E.ch_intra_causality)[] in
EVERY (λe. MEM (proc e) E.ch_procs) E.ch_events ∧
EVERY (λe1.
  EVERY (λe2.
    if (e1.iiid = e2.iiid) ∧ (e1.eiid = e2.eiid) then
      e1 = e2
    else T)
    E.ch_events)
  E.ch_events ∧
EVERY (λ(e1, e2). MEM e1 E.ch_events ∧ MEM e2 E.ch_events) E.ch_intra_causality ∧
EVERY (λ(e1, e2). e1 ≠ e2) intra ∧
EVERY (λ(e1, e2). e1.iiid = e2.iiid) intra ∧
¬ MEM[] E.ch_atomicity ∧
EVERY (λes. EVERY (λe. MEM e E.ch_events) es) E.ch_atomicity ∧
EVERY (λes1.
  EVERY (λes2.
    if ¬ check_set_eq es1 es2 then
      EVERY (λe1. EVERY (λe2. e1 ≠ e2) es1) es2
    else
      T)
    E.ch_atomicity)
  E.ch_atomicity ∧
EVERY (λes1. EVERY (λe1. EVERY (λe2. e1.iiid = e2.iiid) es1) es1) E.ch_atomicity ∧
EVERY (λe. case loc e of SOME (LOCATION_REG p r) → p = proc e || _ → T) E.ch_events ∧
EVERY (λ(e1, e2). ¬ is_mem_write e1) intra ∧
EVERY (λe1.
  EVERY (λe2.
    if is_write e1 ∧ e1 ≠ e2 ∧ (is_write e2 ∨ is_read e2) ∧
       (e1.iiid = e2.iiid) ∧ (loc e1 = loc e2) then
      MEM (e1, e2) intra ∨ MEM (e2, e1) intra
    else
      T)
    E.ch_events)
  E.ch_events ∧
EVERY (λes.
  EVERY (λe1.
    EVERY (λe2.
      if e1.iiid = e2.iiid then MEM e2 es else T)
      E.ch_events)
    es)
  E.ch_atomicity ∧
EVERY (λes. EXISTS (λe. is_mem_read e) es) E.ch_atomicity
```

The chE_to_E and chX_to_X functions convert from the list-based representation of event structures and execution witnesses to the set-based one.

```
chE_to_E E =
⟦ procs := set E.ch_procs;
  events := set E.ch_events;
  intra_causality :=
  (set E.ch_intra_causality)+ ∪ {(e, e) | e ∈ (set E.ch_events)};
```

$atomicity := \mathsf{set}\ (\text{MAP}\ \mathsf{set}\ E.ch\_atomicity)\rrbracket\rangle$

chX_to_X $E\ X =$
$\langle\!\lbrack\ memory\_order := (\mathsf{set}\ X.ch\_memory\_order)^{+} \cup$
$\{(e, e) \mid e \in \text{mem\_accesses}\ (\text{chE\_to\_E}\ E)\};$
$rfmap := \{(ew, er) \mid \text{MEM}\ er\ E.ch\_events \wedge (X.ch\_rfmap\ er = \textsc{Some}\ ew)\};$
$initial\_state := X.ch\_initial\_state\rrbracket\rangle$

Theorem 28

$\forall E.$ check_well_formed_event_structure $E =$ well_formed_event_structure (chE_to_E E)

Proof sketch (full proof in HOL):
By expanding the definitions and using various theorems about EVERY, EXISTS and MEM from HOL's list library.

□

Theorem 29

$\forall E\ X.$
well_formed_event_structure (chE_to_E E)
$\Longrightarrow$
(check_valid_execution $E\ X =$ valid_execution (chE_to_E E)(chX_to_X $E\ X$))

Proof sketch (full proof in HOL):
By expanding the definitions and using various theorems about EVERY, EXISTS and MEM from HOL's list library.

□

F Change History

- r1068, 2008-12-19: First version of the axiomatic model.
- r1105, 2009-01-07: Fixed the definition of valid execution for infinite executions, adding the condition that only a finite number of same-location reads can be unrelated to a write to that location. Fixed the definition of check_rfmap_written for reads from registers.
- *Revision* : 1746, 2009-03-25: Technical Report version, including the x86-TSO abstract machine memory model, the equivalence result, the verified checker, discussion of litmus tests, and proof outlines.

References

[1] *AMD64 Architecture Programmer's Manual (3 vols).* Advanced Micro Devices, Sept. 2007. rev. 3.14.

[2] *Intel 64 and IA-32 Architectures Software Developer's Manual (5 vols).* Intel Corporation, Nov. 2008. rev. 29.

[3] S. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, Dec 1996.

[4] D. Aspinall and J. Sevcik. Formalising Java's data race free guarantee. In *Proc. TPHOLs, LNCS*, 2007.

[5] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, pages 68–78, 2008.

[6] G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *Proc. POPL*, pages 392–403, 2009.

[7] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. Technical Report MSR-TR-2008-12, Microsoft Research, 2008. Conference version in Proc. CAV 2008, LNCS 5123.

[8] S. Burckhardt, M. Musuvathi, and V. Singh. Verifying compiler transformations for concurrent programs, Jan. 2009. Technical report MSR-TR-2008-171.

[9] D. Dice. Java memory model concerns on Intel and AMD systems. `http://blogs.sun.com/dave/entry/java_memory_model_concerns_on`, Jan. 2008.

[10] S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S. Narayanan. TSOtool: A program for verifying memory systems using the memory consistency model. In *Proc. ISCA*, pages 114–123, 2004.

[11] The HOL 4 system. `http://hol.sourceforge.net/`.

[12] Intel. Intel 64 architecture memory ordering white paper, 2007. SKU 318147-001.

[13] L.Higham, J.Kawash, and N. Verwaal. Defining and comparing memory consistency models. In *PDCS*, 1997. Full version as TR #98/612/03, U. Calgary.

[14] P. Loewenstein. Personal communication, Nov. 2008.

[15] P. N. Loewenstein, S. Chaudhry, R. Cypher, and C. Manovit. Multiprocessor memory model verification. In *Proc. AFM (Automated Formal Methods)*, Aug. 2006. FLoC workshop. `http://fm.csl.sri.com/AFM06/`.

[16] V. M. Luchangco. *Memory consistency models for high-performance distributed computing.* PhD thesis, MIT, 2001.

[17] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang. Fast and generalized polynomial time memory consistency verification. In *CAV*, pages 503–516, 2006.

[18] A. Roychoudhury. Formal reasoning about hardware and software memory models. In *Proc. ICFEM*, pages 423–434, 2002.

[19] V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *Proc. PPoPP*, 2007.

[20] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *Proc. POPL 2009*, Jan. 2009.

[21] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. In *Scalable Shared Memory Multiprocessors*, pages 25–42. Kluwer, 1991.

[22] S.Owens, S. Sarkar, and P. Sewell. The x86-TSO model, 2009. `www.cl.cam.ac.uk/users/pes20/weakmemory/`.

[23] SPARC International, Inc. The SPARC architecture manual, v. 8. Revision SAV080SI9308. `http://www.sparc.org/standards/V8.pdf`, 1992.

[24] Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*, 2004.