A Better x86 Memory Model: x86-TSO

Scott Owens   Susmit Sarkar   Peter Sewell

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

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.

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 [19] 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.

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 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 examples (known as 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 [4]. They allow independent readers to see independent writes (by different processors to different addresses) in different orders, as below (IRIW, see also [6]), but require that, in some sense, causality is respected: *"P5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility)"*.

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				

These informal specifications 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 [19, §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 change in the x86 vendor specifications, was in revision 29 of the Intel SDM (revision 30 is essentially identical, and we are told that there will be a future revision of the AMD specification on similar lines). 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). Programming above a model that lacks any such guarantee would be problematic. The following n5 and n4 examples illustrate the potential difficulties. 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; 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 are omitted here, but can be found in the extended version [16] or in [19]. They include the 9 other IWP tests, illustrating that the various load and store reorderings other than those shown in iwp2.3.a/amd4 (§1) are not possible; the AMD MFENCE tests amd5 and amd10; and several others.

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 [19]). Instead, we have to build a reasonable model that is consistent with the given litmus tests, with observed 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 [20, 21], 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 present here a rigorous x86-TSO model, with two equivalent definitions.

The first definition, in §3.1, is an abstract machine with explicit write buffers. The second definition, in §3.2, is an axiomatic model that defines valid executions in terms of memory orders and reads-from maps. In both, we deal with 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 abstract machine conveys the programmer-level operational intuition behind x86-TSO, whereas the axiomatic model supports constraint-based reasoning about example programs, e.g., by our `memevents` tool in §4.

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|⟩

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|⟩

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.

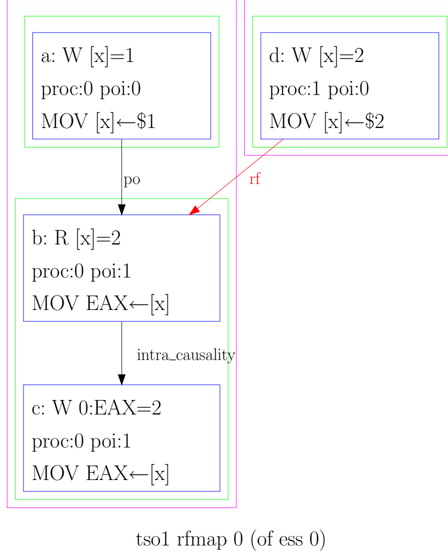
event_structure =⟨| *procs* : proc set;
  events : (*'reg* event)set;
  intra_causality : (*'reg* event)reln;
  atomicity : (*'reg* event)set set|⟩

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 in the on-line version) 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. 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. In Fig. 1, the states s correspond to the parts of the machine shown inside of the dotted line, and the labels l correspond to the communications that traverse the dotted line boundary.

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, including various caches. However, 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

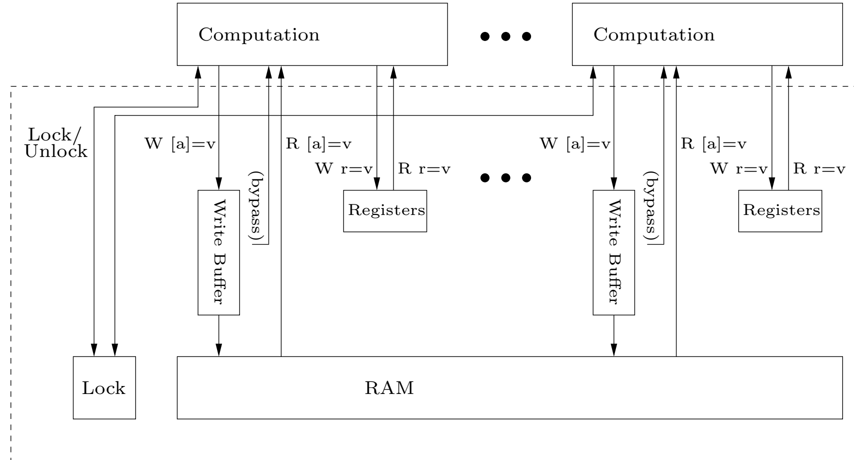


Fig. 1. The abstract machine

the HOL type:

$\mathsf{label} = \textsc{Tau} \mid \textsc{Evt}\ \mathbf{of}\ \mathsf{proc}\ \ ({'reg\ action}) \mid \textsc{Lock}\ \mathbf{of}\ \mathsf{proc} \mid \textsc{Unlock}\ \mathbf{of}\ \mathsf{proc}$

- Tau, for an internal action by the machine;
- 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.

$$
\begin{aligned}
\mathsf{machine_state} = \langle\!|\ & R : \mathsf{proc} \rightarrow {'reg} \rightarrow \mathsf{value\ \ option}; \text{(* per-processor registers *)}\\
& M : \mathsf{address} \rightarrow \mathsf{value\ \ option}; \text{(* main memory *)}\\
& B : \mathsf{proc} \rightarrow (\mathsf{address\#value})\mathsf{list}; \text{(* per-processor write buffers *)}\\
& L : \mathsf{proc\ \ option}\text{(* which processor holds the lock *)}|\!\rangle
\end{aligned}
$$

The behaviour of the x86-TSO machine, the transition relation $s \xrightarrow{l} s'$, is defined by the rules in Fig. 2. 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;

Read from memory

$$\frac{\text{not_blocked}\ \ s\ p \land (s.M\ \ a = \textsc{Some}\ \ v) \land \text{no_pending}\ (s.B\ \ p)a}{s \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ R\ (\textsc{Location_mem}\ a)v)} s}$$

Read from write buffer

$$\frac{\text{not_blocked}\ \ s\ p \land (\exists b_1\ \ b_2.(s.B\ \ p = b_1 +\!\!+ [(a,v)] +\!\!+ b_2) \land \text{no_pending}\ \ b_1\ \ a)}{s \xrightarrow{\textsc{Evt}\ p\ (\textsc{Access}\ R\ (\textsc{Location_mem}\ a)v)} s}$$

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

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

Write from write buffer to memory

$$\frac{\text{not_blocked}\ \ s\ p \land (s.B\ \ p = b +\!\!+ [(a,v)])}{s \xrightarrow{\textsc{Tau}} s \oplus \langle\!| M := s.M \oplus (a \mapsto \textsc{Some}\ \ v); B := s.B \oplus (p \mapsto b) |\!\rangle}$$

Write to register

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

Barrier

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

Lock

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

Unlock

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

Fig. 2. The x86-TSO Machine Behaviour

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.

Consider execution paths through the machine $s_0 \xrightarrow{l_1} s_1 \xrightarrow{l_2} s_2 \cdots$ consisting of finite or infinite sequences of states and labels. We define okMpath to hold for paths through the machine that start in a valid initial state (with empty write buffers, etc.) and satisfy the following progress condition: for each memory write in the path, the corresponding TAU transition appears later on. This ensures that no write can stay in the buffer forever. (We actually formalize okMpath for the event-annotated machine described below.)

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. There are several variants of the machine with different degrees of locking which we conjecture are observationally equivalent. For example, one could prohibit all activity by other processors when one holds the lock, or not require write buffers to be flushed at the start of a LOCK'd instruction.

We relate the machine to event structures in two steps, which we summarise here (the HOL details can be found on-line [16]). 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. Second, we relate paths of annotated labels and event structures with a predicate okEpath that holds when the path is a suitable linearization of the event structure: there is a 1:1 correspondence between non-TAU/LOCK/UNLOCK labels of *path* and the events of E, the order of labels in *path* is consistent with program order and intra-causality, and atomic sets are properly bracketed by LOCK/UNLOCK pairs. Thus, okMpath

describes paths that are valid according to the memory model, and okEpath describes those that are valid according to an event structure (that encapsulates the other aspects of processor semantics).

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

3.2 The x86-TSO Axiomatic Memory Model

Our x86-TSO axiomatic memory model is based on the SPARCv8 memory model specification [20, 21], but adapted to x86 and in the same terms as our earlier x86-CC model. (Readers unfamiliar with the SPARCv8 memory model can safely ignore the SPARC-specific comments in this section.) Compared with the SPARCv8 TSO specification, we omit instruction fetches (IF), instruction loads (IL), flushes (F), and stbars ($\not{S}$). 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.

$$
\begin{aligned}
&\mathsf{execution_witness} = \\
&\quad \langle\!| \, \mathit{memory_order} : ('\mathit{reg}\ \mathsf{event})\mathsf{reln}; \\
&\quad\ \ \mathit{rfmap} : ('\mathit{reg}\ \mathsf{event})\mathsf{reln}; \\
&\quad\ \ \mathit{initial_state} : ('\mathit{reg}\ \mathsf{location} \rightarrow \mathsf{value}\ \ \mathsf{option}) |\!\rangle
\end{aligned}
$$

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).

$$
\begin{aligned}
&\text{partial_order}\ (<_{X.\mathit{memory_order}})(\text{mem_accesses}\ \ E) \\
&\text{linear_order}\ ((<_{X.\mathit{memory_order}})|_{(\text{mem_writes}\ \ E)})(\text{mem_writes}\ \ E)
\end{aligned}
$$

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. (Strictly speaking, the rfmap is unnecessary; the constraints involving it can be stated directly in terms of memory order, as SPARCv8 does. However, we find it intuitive and useful. The SPARCv8 model has no initial states.)

$$
\begin{aligned}
&\text{reads_from_map_candidates}\ \ E\ \mathit{rfmap} = \\
&\quad \forall (ew, er) \in \mathit{rfmap}.(er\ \in\ \text{reads}\ \ E) \wedge (ew\ \in\ \text{writes}\ \ E) \wedge \\
&\qquad\qquad\qquad\qquad (\text{loc}\ \ ew = \text{loc}\ \ er) \wedge (\text{value_of}\ \ ew = \text{value_of}\ \ er)
\end{aligned}
$$

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.

$$
\begin{aligned}
&\text{po_strict } E = \\
&\quad \{(e_1, e_2) \mid (e_1.\mathit{iiid.proc} = e_2.\mathit{iiid.proc}) \wedge e_1.\mathit{iiid.poi} < e_2.\mathit{iiid.poi} \wedge \\
&\qquad\qquad e_1 \in E.\mathit{events} \wedge e_2 \in E.\mathit{events}\} \\
&<_{(\text{po_iico } E)} = \text{po_strict } E \cup E.\mathit{intra_causality}
\end{aligned}
$$

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.

$$
\begin{aligned}
&\text{previous_writes } E\ er\ <_{order} = \\
&\quad \{ew' \mid ew' \in \text{writes } E \wedge ew' <_{order} er \wedge (\text{loc } ew' = \text{loc } er)\} \\
&\text{check_rfmap_written } E\ X = \\
&\quad \forall (ew, er) \in (X.\mathit{rfmap}). \\
&\qquad \textbf{if } ew \in \text{mem_accesses } E \textbf{ then} \\
&\qquad\quad ew \in \text{maximal_elements } (\text{previous_writes } E\ er\ (<_{X.\mathit{memory_order}}) \cup \\
&\qquad\qquad\qquad\qquad \text{previous_writes } E\ er\ (<_{(\text{po_iico } E)})) \\
&\qquad\qquad\qquad (<_{X.\mathit{memory_order}}) \\
&\qquad \textbf{else } (\text{* ew IN reg_accesses E *}) \\
&\qquad\quad ew \in \text{maximal_elements } (\text{previous_writes } E\ er\ (<_{(\text{po_iico } E)}))(<_{(\text{po_iico } E)}) \\
&\text{check_rfmap_initial } E\ X = \\
&\quad \forall er \in (\text{reads } E \setminus \text{range } X.\mathit{rfmap}). \\
&\qquad (\exists l.(\text{loc } er = \text{SOME } l) \wedge (\text{value_of } er = X.\mathit{initial_state}\ l)) \wedge \\
&\qquad (\text{previous_writes } E\ er\ (<_{X.\mathit{memory_order}}) \cup \\
&\qquad\quad \text{previous_writes } E\ er\ (<_{(\text{po_iico } E)}) = \{\})
\end{aligned}
$$

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**):

  $$
  \begin{aligned}
  &\forall er \in (\text{mem_reads } E).\forall e \in (\text{mem_accesses } E). \\
  &\quad er <_{(\text{po_iico } E)} e \implies er <_{X.\mathit{memory_order}} e
  \end{aligned}
  $$

- 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):

  $$\forall es \in (E.atomicity).\forall e \in (\text{mem_accesses}\ E \setminus es).\ (\forall e' \in (es \cap \text{mem_accesses}\ E).e <_{X.memory_order} e') \vee (\forall e' \in (es \cap \text{mem_accesses}\ E).e' <_{X.memory_order} e)$$

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).

$$\text{finite_prefixes}\ (<_{X.memory_order})(\text{mem_accesses}\ E)$$

$$\forall ew \in (\text{mem_writes}\ E).\ \text{finite}\{er \mid er \in E.events \wedge (\text{loc}\ er = \text{loc}\ ew) \wedge er \not<_{X.memory_order} ew \wedge ew \not<_{X.memory_order} er\}$$

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. 2 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. 3. 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.

3.3 The Machine and Axiomatic x86-TSO Models are Equivalent

To prove that the abstract machine admits only valid executions, we define a function path_to_X from event-annotated paths that builds a linear execution witness by using the events from Tau and memory read labels in order. Thus, the memory ordering in the execution witness corresponds to the order in which events were read from and written to memory in the abstract machine.

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]

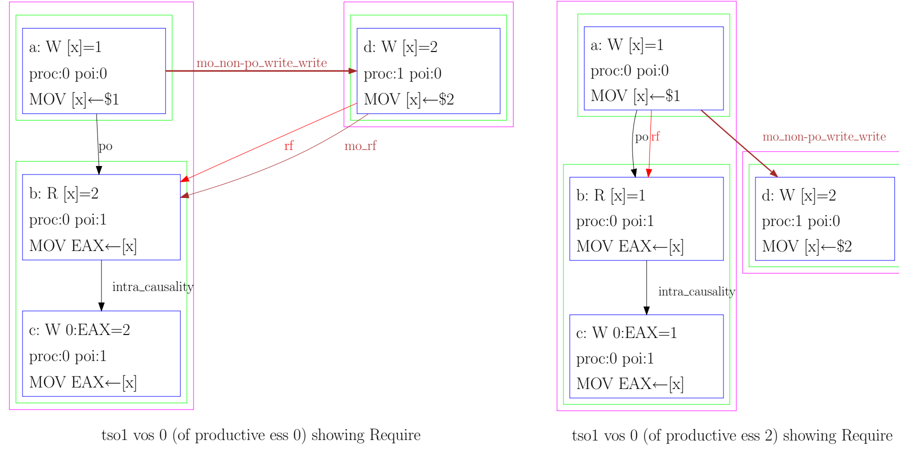


Fig. 3. Example valid execution witnesses (for two different event structures)

To prove that the abstract machine admits every valid execution, we first prove (in HOL) a lemma showing that any valid execution can be a turned into a stream-like linear order over labels that satisfies several conditions (label_order in the HOL sources) describing labels in an okMpath. We then have:

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, relying on the preceding lemma]

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 rwc-fenced. The remaining tests (amd3, n1, n7, n8, 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], and by Higham et al. [13]; the latter formalises a range of models, including a TSO model, in both operational and axiomatic styles, and proves equivalence results. Their axiomatic TSO model is rather closer to the operational style than ours is, and both are idealised rather than x86-specific. Burckhardt and Musuvathi [8, Appendix A] also give operational and axiomatic definitions of a TSO model and prove equivalence, but only for finite executions. Their models treat memory reads and writes and barrier events, but lack register events and locked instructions with multiple events that 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 [7] give an operational model with hierarchical write buffers (thereby permitting IRIW behaviours), and prove sequential consistency for data-race-free (DRF) programs. 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. Saraswat et al. [18] also define memory models in terms of local reordering, and prove a DRF theorem, but focus on high-level languages. Several groups have used proof tools to tame the intricacies of these models, including Yang et al. [22], using Prolog and SAT solvers to explore an axiomatic Itanium model, and Aspinall and Ševčík [5], 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, and 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 that 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 helpful remarks. We acknowledge funding from EPSRC grant EP/F036345.

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. M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.
5. D. Aspinall and J. Ševčík. Formalising Java's data race free guarantee. In *Proc. TPHOLs, LNCS*, 2007.
6. H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. In *Proc. PLDI*, 2008.
7. G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *Proc. POPL*, pages 392–403, 2009.
8. 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.
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. S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO (extended version). Technical Report UCAM-CL-TR-745, Univ. of Cambridge, 2009. Supporting material at `www.cl.cam.ac.uk/users/pes20/weakmemory/`.
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. V. Saraswat, R. Jagadeesan, M. Michael, and C. von Praun. A theory of memory models. In *Proc. PPoPP*, 2007.
19. 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.
20. P. S. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. In *Scalable Shared Memory Multiprocessors*, pages 25–42. Kluwer, 1991.
21. SPARC International, Inc. The SPARC architecture manual, v. 8. Revision SAV080SI9308. `http://www.sparc.org/standards/V8.pdf`, 1992.
22. Y. Yang, G. Gopalakrishnan, G. Lindstrom, and K. Slind. Nemos: A framework for axiomatic and executable specifications of memory consistency models. In *IPDPS*, 2004.