x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors

Peter Sewell
University of Cambridge

Susmit Sarkar
University of Cambridge

Scott Owens
University of Cambridge

Francesco Zappa Nardelli
INRIA

Magnus O. Myreen
University of Cambridge

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

ABSTRACT

Exploiting the multiprocessors that have recently become ubiquitous requires high-performance and reliable concurrent systems code, for concurrent data structures, operating system kernels, synchronisation libraries, compilers, and so on. However, concurrent programming, which is always challenging, is made much more so by two problems. First, real multiprocessors typically do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have relaxed memory models, varying in subtle ways between processor families, in which different hardware threads may have only loosely consistent views of a shared memory. Second, the public vendor architectures, supposedly specifying what programmers can rely on, are often in ambiguous informal prose (a particularly poor medium for loose specifications), leading to widespread confusion.

In this paper we focus on x86 processors. We review several recent Intel and AMD specifications, showing that all contain serious ambiguities, some are arguably too weak to program above, and some are simply unsound with respect to actual hardware. We present a new *x86-TSO* programmer's model that, to the best of our knowledge, suffers from none of these problems. It is mathematically precise (rigorously defined in HOL4) but can be presented as an intuitive abstract machine which should be widely accessible to working programmers. We illustrate how this can be used to reason about the correctness of a Linux spinlock implementation and describe a general theory of data-race-freedom for x86-TSO. This should put x86 multiprocessor system building on a more solid foundation; it should also provide a basis for future work on verification of such systems.

1. INTRODUCTION

Multiprocessor machines, with many processors acting on a shared memory, have been developed since the 1960s; they are now ubiquitous. Meanwhile, the difficulty of programming concurrent systems has motivated extensive research on programming language design, semantics, and verification, from semaphores and monitors to program logics, software model checking, and so forth. This work has almost always assumed that concurrent threads share a single sequentially consistent memory [21], with their reads and writes interleaved in some order. In fact, however, real multiprocessors use sophisticated techniques to achieve high performance: store buffers, hierarchies of local cache, speculative execution, etc. These optimisations are not observable by sequential code, but in multithreaded programs different threads may see subtly different views of memory; such machines exhibit *relaxed*, or *weak*, *memory models* [6, 17, 19, 7].

For a simple example, consider the following assembly language program (SB) for modern Intel or AMD x86 multiprocessors: given two distinct memory locations x and y (initially holding 0), if two processors respectively write 1 to x and y and then read from y and x (into register EAX on processor 0 and EBX on processor 1), it is possible for both to read 0 *in the same execution*. It is easy to check that this result cannot arise from any interleaving of the reads and writes of the two processors; modern x86 multiprocessors do not have a sequentially consistent semantics.

SB

Proc 0	Proc 1
MOV [x]←1	MOV [y]←1
MOV EAX←[y]	MOV EBX←[x]
Allowed Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

Microarchitecturally, one can view this particular example as a visible consequence of store buffering: if each processor effectively has a FIFO buffer of pending memory writes (to avoid the need to block while a write completes), then the reads from y and x could occur before the writes have propagated from the buffers to main memory.

Other families of multiprocessors, dating back at least to the IBM 370, and including ARM, Itanium, POWER, and SPARC, also exhibit relaxed-memory behaviour. Moreover, there are major and subtle differences between different processor families (arising from their different internal design choices): in the details of exactly what non-sequentially-consistent executions they permit, and of what memory barrier and synchronisation instructions they provide to let the programmer regain control.

For any of these processors, relaxed-memory behaviour exacerbates the difficulties of writing concurrent software, as systems programmers cannot reason, at the level of abstraction of memory reads and writes, in terms of an intuitive concept of global time.

Still worse, while some vendors' architectural specifications clearly define what they guarantee, others do not, despite the extensive previous research on relaxed memory models. We focus in this paper on x86 processors. In Section 2 we introduce the key examples and discuss several vendor specifications, showing that they all leave key questions ambiguous, some give unusably weak guarantees, and

some are simply wrong, prohibiting behaviour that actual processors do exhibit.

For there to be any hope of building reliable multiprocessor software, systems programmers need to understand what relaxed-memory behaviour they *can* rely on, but at present that understanding exists only in folklore, not in clear public specifications. To remedy this, we aim to produce mathematically precise (but still appropriately loose) programmer's models for real-world multiprocessors, to inform the intuition of systems programmers, to provide a sound foundation for rigorous reasoning about multiprocessor programs, and to give a clear correctness criterion for hardware. In Section 3 we describe a simple x86 memory model, x86-TSO [27]. In contrast to those vendor specifications it is unambiguous, defined in rigorous mathematics, but it is also accessible, presented in an operational abstract-machine style. To the best of our knowledge it is consistent with the behaviour of actual processors. We consider the relevant vendor litmus tests in Section 3.2 and describe some empirical test results in Section 3.3.

Relaxed memory behaviour is particularly critical for low-level systems code: synchronisation libraries, concurrent data structure libraries, language runtime systems, compilers for concurrent languages, and so on. To reason (even informally) about such code, such as the implementation of an OS mutual exclusion lock, one would necessarily depend on the details of a specific model. Higher-level application code, on the other hand, should normally be oblivious to the underlying processor memory model. The usual expectation is that such code is in some sense *race free*, with all access to shared memory (except for accesses within the library code) protected by locks or clearly identified as synchronisation accesses. Most memory models are designed with the intention that such race-free code behaves *as if* it were executing on a sequentially consistent machine. In Section 4 we describe an implementation of spin locks for x86, from one version of the Linux kernel, and discuss informally why it is correct with respect to x86-TSO. In Section 5 we define a precise notion of data race for x86 and discuss results showing that programs that use spin locks but are otherwise race-free (except for the races within the lock implementation) do indeed behave as if executing on a sequentially consistent machine [26].

To support formal reasoning about programs, a memory model must be integrated with a semantics for machine instructions (a problem which has usually been neglected in the relaxed memory literature). In previous work [31, §3] we describe a semantics for core x86 instructions, with several innovations. We take care not to over-sequentialise the memory accesses within each instruction, parameterising the instruction semantics over parallel and sequential combinators. A single definition, with all the intricacies of flag-setting, addressing modes, etc., can then be used to generate both an event-based semantics that can be integrated with memory models, and a state-based semantics for sequential programs; the latter enables us to test the semantics against implementations. We also build an instruction decoding function, directly from the vendor documentation, to support reasoning about concrete machine code.

The intended scope of x86-TSO is typical user code and most kernel code: we cover 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. Within this domain, and together with our earlier instruction semantics, x86-TSO thus defines a complete semantics of programs.

Relaxed memory models play an important role also in the design of high-level concurrent languages such as Java or C++0x, where programs are subject not just to the memory model of the underlying processor but also to reorderings introduced by compiler optimisations. The Java Memory Model [24] attempts to ensure that data-race free programs are sequentially consistent; all programs satisfy memory safety/security properties; and common compiler optimisations are sound. Unfortunately, as shown by Ševčík and Aspinall [33], the last goal is not met. In the future, we hope that it will be possible to prove correctness of implementations of language-level memory models above the models provided by real-world processors; ensuring that both are precisely and clearly specified is a first step towards that goal.

2. ARCHITECTURE SPECIFICATIONS

To describe what programmers can rely on, processor vendors document *architectures*. These are loose specifications, claimed to cover a range of past and future processor implementations, which should specify processor behaviour tightly enough to enable effective programming, but without unduly constraining future processor designs. For some architectures the memory-model aspects of these specifications are expressed in reasonably precise mathematics, as in the normative Appendix K of the SPARC v.8 specification [2]. For x86, however, the vendor architecture specifications are informal prose documents. Informal prose is a poor medium for loose specification of subtle properties, and, as we shall see, such documents are almost inevitably ambiguous and sometimes wrong. 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. In this section we review the informal-prose Intel and AMD x86 specifications: the Intel 64 and IA-32 Architectures Software Developer's Manual (SDM) [5] and the AMD64 Architecture Programmer's Manual (APM) [3]. There have been several versions of these, some differing radically; we contrast them with each other, and with what we have discovered of the behaviour of actual processors. In the process we introduce the key discriminating examples.

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 see precisely what this prose means, especially without additional knowledge or assumptions about the microarchitecture of particular implementations. The uncertainty about x86 behaviour that at least some systems programmers had about earlier IA-32 processors can be gauged from an extensive discussion about the correctness of a proposed optimisation to a Linux spinlock implementation [1]. The discussion is largely in microarchitectural terms, not just in terms of the specified architecture, and seems to have been resolved only with input from Intel staff. We return to this optimisation in Section 4, where we can explain why it is sound with respect to x86-TSO.

2.2 IWP/AMD3.14/x86-CC

In August 2007, an Intel White Paper [4] (IWP) gave a somewhat more precise model, with 8 informal-prose principles P1–P8 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 in rev. 3.14 of their manual [3, Vol. 2,§7.2] (AMD3.14). These are essentially causal-consistency models [9], and they allow different processors to see writes to independent locations in different orders, as in the IRIW litmus test [11] below[1]. AMD3.14 allows this explicitly, while IWP allows it implicitly, as IRIW is not ruled out by the stated principles. Microarchitecturally, IRIW can arise from store buffers that are shared between some but not all processors.

IRIW

Proc 0	Proc 1	Proc 2	Proc 3
MOV [x]←1	MOV [y]←1	MOV EAX←[x] MOV EBX←[y]	MOV ECX←[y] MOV EDX←[x]
Forbidden Final State: Proc 2:EAX=1 ∧ Proc 2:EBX=0 ∧ Proc 3:ECX=1 ∧ Proc 3:EDX=0			

However, both require that, in some sense, causality is respected, as in the IWP principle *"P5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility)"*.

We used these informal specifications as the basis for a formal model, x86-CC [31], for which a key issue was giving a reasonable interpretation to this "causality", which is not defined in IWP or AMD3.14. 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 (instead, one would have to make liberal use of x86 LOCK'd instructions) [31, §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, even without MFENCEs. It appears that some JVM implementations depend on this fact, and would not be correct if one assumed only the IWP/AMD3.14/x86-CC architecture [15].

Second, more seriously, x86-CC and IWP are unsound with respect to current processors. The following example, n6, due to Paul Loewenstein [personal communication, Nov. 2008] 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 principles P1,2,4 and 6 [27, A.5].

n6

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[x] MOV EBX←[y]	MOV [y]←2 MOV [x]←2
Allowed Final State: Proc 0:EAX=1 ∧ Proc 0:EBX=0 ∧ [x]=1	

[1]We use Intel assembly syntax throughout except that we use an arrow ← to indicate the direction of data flow, so MOV [x]←1 is a write of 1 to address x and MOV EAX←[x] is a read from address x into register EAX. Initial states are all 0 unless otherwise specified.

To see why this could be allowed by multiprocessors with FIFO store 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 store 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.

The AMD3.14 manual is not expressed in terms of a clearly identified set of principles, and the main text (Vol. 2, §7.2) leaves the ordering of stores to a single location unconstrained, though elsewhere the manual describes a microarchitecture with store buffers and cache protocols that strongly implies that memory is coherent. In the absence of an analogue of the IWP P6, the reasoning prohibiting n6 does not carry over.

2.3 Intel SDM rev. 29–34 (Nov. 2008–Mar. 2010)

The most recent substantial change to the Intel memory-model specification, at the time of writing, was in revision 29 of the Intel SDM (revisions 29–34 are essentially identical except for the LFENCE text). This is in a similar informal-prose style to previous versions, again supported by litmus tests, but is significantly different to IWP/x86-CC/AMD3.14. First, the IRIW final state above is forbidden [5, Example 8-7, vol. 3A], and the previous coherence condition: *"P6. In a multiprocessor system, stores to the same location have a total order"* has been replaced by: *"Any two stores are seen in a consistent order by processors other than those performing the stores"* (we label this P9).

Second, the memory barrier instructions are now included. It is stated that reads and writes cannot pass MFENCE instructions, together with more refined properties for SFENCE and LFENCE.

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

This revision 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). The following examples (which we call n5 and n4b) illustrate potentially surprising behaviour that arguably violates coherence. Their final states are not allowed in x86-CC, are not allowed in a pure store-buffer implementation or in x86-TSO, and we have not observed them on actual processors. However, the principles stated in revisions 29–34 of the Intel SDM appear, presumably unintentionally, to allow them. The AMD3.14 Vol. 2, §7.2 text taken alone would allow them, but the implied coherence from elsewhere in the AMD manual would forbid them. These points illustrate once again the difficulty of writing unambiguous and correct loose specifications in informal prose.

n5

Proc 0	Proc 1
MOV [x]←1 MOV EAX←[x]	MOV [x]←2 MOV EBX←[x]
Forbidden Final State: Proc 0:EAX=2 ∧ Proc 1:EBX=1	

n4b

Proc 0	Proc 1
MOV EAX←[x]	MOV ECX←[x]
MOV [x]←1	MOV [x]←2
Forbidden Final State: Proc 0:EAX=2 ∧ Proc 1:ECX=1	

2.4 AMD3.15 (Nov. 2009)

In November 2009, AMD produced a new revision, 3.15, of their manuals. The main difference in the memory model specification is that IRIW is now explicitly forbidden.

Summarising the key litmus-test differences, we have the following, where $\surd$ and $\times$ entries are explicit in the specification text and starred entries indicate possible deductions, some of which may not have been intended.

	IWP / x86-CC	3.14	29–34	3.15	actual processors
IRIW	$\surd^*/\surd$	$\surd$	$\times$	$\times$	not observed
n6	$\times^*/\times$	$\surd^*$	$\surd^*$	$\surd^*$	observed
n5/n4b	$\times^*/\times$	$\times^*$	$\surd^*$	$\times^*$	not observed

There are also many non-differences: tests for which the behaviours coincide in all three cases. We return to these, and go through the other tests from the Intel and AMD documentation, in Section 3.2.

3. OUR X86-TSO PROGRAMMER'S 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 [31]. 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, the vendors' intentions, and the folklore in the area.

We emphasise that our aim is a *programmer's model*, of the allowable behaviours of x86 processors as observed by assembly programs, not of the internal structure of processor implementations, or of what could be observed on hardware interfaces. We present the model in an abstract-machine style to make it accessible, but are concerned only with its external behaviour; its buffers and locks are highly abstracted from the microarchitecture of processor implementations.

The fact that store buffering is observable, as in the SB and n6 examples, but IRIW is not (and IRIW is explicitly forbidden in the SDM revs. 29–34 and AMD3.15), together with additional tests that prohibit many other reorderings, strongly suggests that, apart from store buffering, all processors share the same view of memory. Moreover, different processors or hardware threads do not observably share store buffers. This is in sharp contrast to x86-CC, where each processor has a separate view order of its memory accesses and other processors' writes. To the best of our knowledge, for the usual write-back memory, no other aspects of the microarchitecture (the out-of-order execution, cache hierarchies and protocols, interconnect topology, and so on) are observable to the programmer, except in so far as they affect performance.

This is broadly similar to the SPARC Total Store Ordering (TSO) memory model [32, 2], which is essentially an axiomatic description of the behaviour of store-buffer multiprocessors. Accordingly, we have designed a TSO-like model for

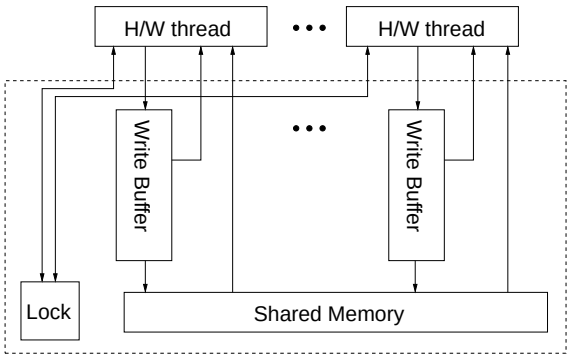

Figure 1: x86-TSO block diagram

x86, called x86-TSO [27]. It is defined mathematically in two styles: an abstract machine with explicit store buffers, and an axiomatic model that defines valid executions in terms of memory orders; they are formalised in HOL4 [20] and are proved equivalent. The abstract machine conveys the programmer-level operational intuition behind x86-TSO; we describe it informally in the next subsection. The axiomatic model supports constraint-based reasoning about example programs (e.g. by our `memevents` tool in Section 3.3); it is similar to that of SPARCv8 [2, App. K], but we also deal with x86 CISC instructions with multiple memory accesses and with x86 barriers and atomic (or *LOCK'd*) instructions. The x86 supports a range of atomic instructions: one can add a LOCK prefix to many read-modify-write instructions (ADD, INC, etc.), and the XCHG instruction is implicitly LOCK'd. There are three main memory barriers: MFENCE, SFENCE and LFENCE.

3.1 The Abstract Machine

Our programmer's model of a multiprocessor x86 system is illustrated in Figure 1. At the top of the figure are a number of hardware threads, each corresponding to a single in-order stream of instruction execution. (In this programmer's model there is no need to consider physical processors explicitly; it is hardware threads that correspond to the Proc N columns in the tests we give.) They interact with a storage subsystem, drawn as the dotted box.

The state of the storage subsystem comprises a shared memory that maps addresses to values, a global lock to indicate when a particular hardware thread has exclusive access to memory, and one store buffer per hardware thread.

The behaviour of the storage subsystem is described in more detail below, but the main points are:

- The store buffers are FIFO and a reading thread must read its most recent buffered write, if there is one, to that address; otherwise reads are satisfied from shared memory.
- An MFENCE instruction flushes the store buffer of that thread.
- To execute a LOCK'd instruction, a thread must first obtain the global lock. At the end of the instruction, it flushes its store buffer and relinquishes the lock. While the lock is held by one thread, no other thread can read.

- A buffered write from a thread can propagate to the shared memory at any time except when some other thread holds the lock.

More precisely, the possible interactions between the threads and the storage subsystem are described by the following *events*:

- $\mathsf{W}_p[a]{=}v$, for a write of value v to address a by thread p
- $\mathsf{R}_p[a]{=}v$, for a read of v from a by thread p
- F_p, for an MFENCE memory barrier by thread p
- L_p, at the start of a LOCK'd instruction by thread p
- U_p, at the end of a LOCK'd instruction by thread p
- τ_p, for an internal action of the storage subsystem, propagating a write from p's store buffer to the shared memory

For example, suppose a particular hardware thread p has come to the instruction INC [56] (which adds 1 to the value at address 56), and p's store buffer contains a single write to 56, of value 0. In one execution we might see read and write events, $\mathsf{R}_p[56]{=}0$ and $\mathsf{W}_p[56]{=}1$, followed by two τ_p events as the two writes propagate to shared memory. Another execution might start with the write of 0 propagating to shared memory, where it could be overwritten by another thread. Executions of LOCK;INC [56] would be similar but bracketed by L_p and U_p events.

The behaviour of the storage subsystem is specified by the following rules, where we define a hardware thread to be *blocked* if the storage subsystem lock is taken by another hardware thread, i.e., while another hardware thread is executing a LOCK'd instruction.

1. $\mathsf{R}_p[a]{=}v$: p can read v from memory at address a if p is not blocked, there are no writes to a in p's store buffer, and the memory does contain v at a;
2. $\mathsf{R}_p[a]{=}v$: p can read v from its store buffer for address a if p is not blocked and has v as the newest write to a in its buffer;
3. $\mathsf{W}_p[a]{=}v$: p can write v to its store buffer for address a at any time;
4. τ_p: if p is not blocked, it can silently dequeue the oldest write from its store buffer and place the value in memory at the given address, without coordinating with any hardware thread;
5. F_p: if p's store buffer is empty, it can execute an MFENCE (note that if a hardware thread encounters an MFENCE instruction when its store buffer is not empty, it can take one or more τ_p steps to empty the buffer and proceed, and similarly in 7 below);
6. L_p: if the lock is not held, it can begin a LOCK'd instruction; and
7. U_p: if p holds the lock, and its store buffer is empty, it can end a LOCK'd instruction.

Technically, the formal versions of these rules [27] define a labelled transition system (with the events as labels) for the storage subsystem, and we define the behaviour of the whole system as a parallel composition of that and transition systems for each thread, synchronising on the non-τ labels as in CCS [25].

Additionally, we tentatively impose a progress condition, that each memory write is eventually propagated from the relevant store buffer to the shared memory. This is not stated in the documentation and is hard to test. We are assured that it holds at least for AMD processors.

For write-back cacheable memory, and the fragment of the instruction set that we consider, we treat LFENCE and SFENCE semantically as no-ops. This follows the Intel and AMD documentation, both of which imply that these fences do not order store/load pairs which are the only reorderings allowed in x86-TSO. Note, though, that elsewhere it is stated that the Intel SFENCE flushes the store buffer [5, Vol.3A, §11.10].

3.2 Litmus Tests

For our introductory SB example from Section 1, x86-TSO permits the given behaviour for the same reasons as set forth there. For each of the examples in Section 2 (IRIW, n6, and n5/n4b), x86-TSO permits the given final state if and only if it is observable in our testing of actual processors, i.e. for IRIW it is forbidden (in contrast to IWP and AMD3.14), for n6 it is allowed (in contrast to IWP), and for n5/n4b it is forbidden (in contrast to the Intel SDM rev.29–34). For all the other relevant tests from the current Intel and AMD manuals the stated behaviour agrees with x86-TSO. We now go through Examples 8-1 to 8-10 from rev. 34 of the Intel SDM, and the three other tests from AMD3.15, and explain the x86-TSO behaviour in each case.

For completeness we repeat the Intel SDM short descriptions of these tests, e.g. *"stores are not reordered with other stores"*, but note that "not reordered with" is not defined there and is open to misinterpretation [27, §3.2].

Example 8-1. Stores Are Not Reordered with Other Stores.

Proc 0	Proc 1
MOV [x]←1	MOV EAX←[y]
MOV [y]←1	MOV EBX←[x]
Forbidden Final State: Proc 1:EAX=1 ∧ Proc 1:EBX=0	

This test implies that the writes by Proc 0 are seen in order by Proc 1's reads, which also execute in order. x86-TSO forbids the final state because Proc 0's store buffer is FIFO, and Proc 0 communicates with Proc 1 only through shared memory.

Example 8-2. Stores Are Not Reordered with Older Loads.

Proc 0	Proc 1
MOV EAX←[x]	MOV EBX←[y]
MOV [y]←1	MOV [x]←1
Forbidden Final State: Proc 0:EAX=1 ∧ Proc 1:EBX=1	

x86-TSO forbids the final state because reads are never delayed.

Example 8-3. Loads May be Reordered with Older Stores. This test is just the SB example from Section 1, which x86-TSO permits. The third AMD test (amd3) is

similar but with additional writes inserted in the middle of each thread, of 2 to x and y respectively.

Example 8-4. Loads Are not Reordered with Older Stores to the Same Location.

Proc 0
MOV [x]←1
MOV EAX←[x]
Required Final State: Proc 0:EAX=1

x86-TSO requires the specified result because reads must check the local store buffer.

Example 8-5. Intra-Processor Forwarding is Allowed. This test is similar to Example 8-3.

Example 8-6. Stores Are Transitively Visible.

Proc 0	Proc 1	Proc 2
MOV [x]←1	MOV EAX←[x] MOV [y]←1	MOV EBX←[y] MOV ECX←[x]
Forbidden Final State: Proc 1:EAX=1 ∧ Proc 2:EBX=1 ∧ Proc 2:ECX=0		

x86-TSO forbids the given final state because otherwise the Proc 2 constraints imply that y was written to shared memory before x. Hence the write to x must be in Proc 0's store buffer (or the instruction has not executed), when the write to y is initiated. Note that this test contains the only mention of "transitive visibility" in the Intel SDM, leaving its meaning unclear.

Example 8-7. Stores Are Seen in a Consistent Order by Other Processors. This test rules out the IRIW behaviour as described in Section 2.2. x86-TSO forbids the given final state because the Proc 2 constraints imply that x was written to shared memory before y whereas the Proc 3 constraints imply that y was written to shared memory before x.

Example 8-8. Locked Instructions Have a Total Order. This is the same as the IRIW Example 8-7 but with LOCK'd instructions for the writes; x86-TSO forbids the final state for the same reason as above.

Example 8-9. Loads Are not Reordered with Locks.

Proc 0	Proc 1
XCHG [x]←EAX MOV EBX←[y]	XCHG [y]←ECX MOV EDX←[x]
Initial state: Proc 0:EAX=1 ∧ Proc 1:ECX=1 (elsewhere 0)	
Forbidden Final State: Proc 0:EBX=0 ∧ Proc 1:EDX=0	

This test indicates that locking both writes in Example 8-3 would forbid the non-sequentially consistent result. x86-TSO forbids the final state because LOCK'd instructions flush the local store buffer. If only one write were LOCK'd (say the write to x), the Example 8-3 final state would be allowed as follows: on Proc 1, buffer the write to y and execute the read x, then on Proc 0 write to x in shared memory then read from y.

Example 8-10. Stores Are not Reordered with Locks.

Proc 0	Proc 1
XCHG [x]←EAX MOV [y]←1	MOV EBX←[y] MOV ECX←[x]
Initial state: Proc 0:EAX=1 (elsewhere 0)	
Forbidden Final State: Proc 1:EBX=1 ∧ Proc 1:ECX=0	

This is implied by Example 8-1, as we treat the memory writes of LOCK'd instructions as stores.

Test amd5.

Proc 0	Proc 1
MOV [x]←1 MFENCE MOV EAX←[y]	MOV [y]←1 MFENCE MOV EBX←[x]
Forbidden Final State: Proc 0:EAX=0 ∧ Proc 1:EBX=0	

For x86-TSO, this test has the same force as Example 8-8, but using MFENCE instructions to flush the buffers instead of LOCK'd instructions. The tenth AMD test is similar. None of the Intel litmus tests include fence instructions.

In x86-TSO adding MFENCE between every instruction would clearly suffice to regain sequential consistency (though obviously in practice one would insert fewer barriers), in contrast to IWP/x86-CC/AMD3.14.

3.3 Empirical Testing

To build confidence that we have a sound model of the behaviour of actual x86 processors we have tested the correspondence between them in various ways.

Firstly, for the memory model, we have a `litmus` tool that takes a litmus test (essentially as given in this paper) and builds a C program with embedded assembly to run the test repeatedly to try to produce all possible results, taking care to synchronise the different threads and with some randomisation of memory usage. We have run these on the Intel and AMD processors that we have access to. The results can be compared with the output of a `memevents` tool, that takes such tests and computes the set of all possible executions allowed by the x86-TSO model. We use a verified witness checker, extracted from the HOL4 definition of the model, to verify that any executions found are indeed allowed.

The results correspond exactly for all the tests given here and others we have tried, including amd3, n1 [31], n7 [27], the single-XCHG variant of Example 8-9, and an unfenced variant of RWC [11]. In general, though, there may be tests where x86-TSO allows some final state that cannot be observed in practice, perhaps because `litmus` does not drive the processor into the correct internal state (of store buffers, cache lines, etc.) to exhibit it, or perhaps because the particular implementations we tested cannot exhibit it. For example, we have only seen amd3 on a four-processor (×2 hyperthread) machine and only very rarely, 4 out of 3.2e9 times. Testing, especially this black-box testing of a complex and time-dependent system, is obviously subject to the usual limitations; it cannot conclusively prove that some outcome is not possible.

Secondly, for the behaviour of individual instructions, we have an `x86sem` tool that generates random instances of instructions, runs them on an actual machine, and generates a HOL4 conjecture relating the memory and register state before and after. These conjectures are then automatically verified, by a HOL4 script, for the 4600 instances that we tried.

4. A LINUX X86 SPINLOCK IMPLEMENTATION

In Section 2.1 we mentioned the uncertainty that arose in a discussion on a particular optimisation for Linux spinlocks [1]. In this section, we present a spinlock from the

Linux kernel (version 2.6.24.7) that incorporates the proposed optimisation, as an example of a small but non-trivial concurrent programming idiom. We show how one can reason about this code using the x86-TSO programmer's model, explaining in terms of the model why it works and why the optimisation is sound — thus making clear what (we presume) the developer's informal reasoning depended on. For accessibility we do this in prose, but the argument could easily be formalised as a proof.

The implementation comprises code to acquire and release a spinlock. It is assumed that these are properly bracketed around critical sections and that spinlocks are not mutated by any other code.

```
On entry the address of spinlock is in register EAX
and the spinlock is unlocked iff its value is 1
acquire: LOCK;DEC [EAX]     ; LOCK'd decrement of [EAX]
         JNS      enter     ; branch if [EAX] was ≥ 1
    spin: CMP     [EAX],0   ; test [EAX]
         JLE      spin      ; branch if [EAX] was ≤ 0
         JMP      acquire   ; try again
   enter:  ; the critical section starts here
release: MOV      [EAX]←1
```

A spinlock is represented by a signed integer which is 1 if the lock is free and 0 or less if the lock is held. To acquire a lock, a thread atomically decrements the integer (which will not wrap around assuming there are fewer than 2^{31} hardware threads). If the lock was free, it is now held and the thread can proceed to the critical section. If the lock was held, the thread loops, waiting for it to become free. Because there might be multiple threads waiting for the lock, once it is freed, each waiting thread must again attempt to enter through the LOCK'd decrement. To release the lock, a thread simply sets its value to 1.

The optimisation in question made the releasing MOV instruction not LOCK'd (removing a LOCK prefix and hence letting the releasing thread proceed without flushing its buffer).

For example, consider a spinlock at address x and let y be another shared memory address. Suppose that several threads want to access y, and that they use spinlocks to ensure mutual exclusion. Initially, no one has the lock and [x] = 1. The first thread t to try to acquire the lock atomically decrements x by 1 (using a LOCK prefix); it then jumps into the critical section. Because a store buffer flush is part of LOCK'd instructions, [x] will be 0 in shared memory after the decrement.

Now if another thread attempts to acquire the lock, it will not jump into the critical section after performing the atomic decrement, since x was not 1. It will thus enter the spin loop. In this loop, the waiting thread continually reads the value of x until it gets a positive result.

Returning to the original thread t, it can read and write y inside of its critical section while the others are spinning. These writes are initially placed in t's store buffer, and some may be propagated to shared memory. However, it does not matter how many (if any) are written to main memory, because (by assumption) no other thread is attempting to read (or write) y. When t is ready to exit the critical section, it releases the lock by writing the value 1 to x; this write is put in t's store buffer. It can now continue after the critical section (in the text below, we assume it does not try to re-acquire the lock).

If the releasing MOV had the LOCK prefix then all of the buffered writes to y would be sent to main memory, as would the write of 1 to x. Another thread could then acquire the spinlock.

However, since it does not, the other threads continue to spin until the write setting x to 1 is removed from t's write buffer and sent to shared memory at some point in the future. At that point, the spinning threads will read 1 and restart the acquisition with atomic decrements, and another thread could enter its critical section. However, because t's write buffer is emptied in FIFO order, any writes to y from within t's critical section must have been propagated to shared memory (in order) before the write to x. Thus, the next thread to enter a critical section will not be able to see y in an inconsistent state.

5. DATA-RACE FREEDOM

To make a relaxed-memory architecture usable for large-scale programming, it is highly desirable (perhaps essential) to identify programming idioms which ensure that one can reason in terms of a traditional interleaving model of concurrency, showing that any relaxed-memory execution is equivalent to one that is possible above a sequentially consistent memory model. One common idiom with this property is *data-race freedom*. Informally, a program has a data race if multiple threads can access the same location (where at least one is writing to the location) without a synchronisation operation separating the accesses. Programs where every shared access is in a critical section are one common example of data race free programs.

A variety of relaxed models, both for processors and for programming languages, have been proved to support sequentially consistent semantics for data-race free programs [8, 9, 10, 12, 16, 23]. Saraswat et al. [30] call supporting sequentially consistent semantics for data-race free programs the "fundamental property" of a relaxed memory model, and indeed memory models have sometimes been defined in these terms [6]. However, for a processor architecture, we prefer to define a memory model that is applicable to arbitrary programs, to support reasoning about low-level code, and have results about well-behaved programs as theorems above it.

The details of what constitutes a data race, or a synchronisation operation, vary from model to model. For x86-TSO, we define two events on different threads to be *competing* if they access the same address, one is a write, and the other is a read (for aligned x86 accesses, it is not necessary to consider write/write pairs as competing). We say that a program is data race free if it is impossible for a competing read/write pair to execute back-to-back. Critically, we require this property only of sequentially consistent executions (equivalently, the x86-TSO executions where store buffers are always flushed immediately after each write).

We have proved that x86-TSO supports interleaving semantics for data race free programs. However, this theorem alone is not often useful, because most programs *do* contain data races at this level of abstraction. For example, the read in the spin loop of Section 4's spinlock races with the write in the release. We have, therefore, identified an extended notion of data race freedom that the spinlock code does satisfy, and we have used it to prove that for well synchronised programs using the spinlock exhibit, every x86-TSO execution has an equivalent sequentially consistent execution [26].

Thus, the relaxed nature of x86-TSO is provably not a

concern for low-level systems code that uses spinlocks to synchronise. Extending this result to other synchronisation primitives, and to code compiled from high-level languages, is a major topic for future work.

6. RELATED WORK

There is an extensive literature on relaxed memory models, but most of it does not address x86. 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 [6], by Luchango [23], and by Higham et al. [19]. 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 rather closer to the operational style than ours is, and is idealised rather than x86-specific. Park and Dill [28] verify programs by model checking them directly above TSO. Burckhardt and Musuvathi [13, App. 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 instruction semantics and LOCK'd instructions with multiple events that happen atomically. Hangel et al. [18] describe the Sun TSOtool, checking the observed behaviour of pseudo-randomly generated programs against a TSO model. Roy et al. [29] 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. Loewenstein et al. [22] describe a "golden memory model" for SPARC TSO, somewhat closer to a particular implementation microarchitecture than the abstract machine we give in Section 3, that they use for testing implementations. They argue that the additional intensional detail increases the effectiveness of simulation-based verification. Boudol and Petri [12] 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. [14] define an x86 memory model based on IWP [4]. The mathematical form of their definitions is rather different to our axiomatic and abstract-machine models, 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. Saraswat et al. [30] also define memory models in terms of local reordering, and prove a DRF theorem, but focus on high-level languages.

7. 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. This work highlights the clarity of mathematically rigorous definitions, in contrast to informal prose, for subtle loose specifications.

We do not speak for any x86 vendor, and it is, of course, entirely possible that x86-TSO is not a good description of some existing or future x86 implementation (we would be very interested to hear of any such example). Nonetheless, we hope that this will clarify the semantics of x86 architectures as they exist, for systems programmers, hardware developers, and those working on the verification of concurrent software.

Acknowledgements We thank Luc Maranget for his work on `memevents` and `litmus`, Tom Ridge, Thomas Braibant and Jade Alglave for their other work on the project, and Hans Boehm, David Christie, Dave Dice, Doug Lea, Paul Loewenstein, and Gil Neiger for helpful remarks. We acknowledge funding from EPSRC grants EP/F036345 and EP/H005633 and ANR grant ANR-06-SETI-010-02.

8. REFERENCES

[1] Linux Kernel mailing list, thread "spin_unlock optimization(i386)", 119 messages, Nov. 20–Dec. 7th, 1999, `http://www.gossamer-threads.com/lists/engine?post=105365;list=linux`. Accessed 2009/11/18.

[2] *The SPARC Architecture Manual, V. 8*. SPARC International, Inc., 1992. Revision SAV080SI9308. `http://www.sparc.org/standards/V8.pdf`.

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

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

[5] *Intel 64 and IA-32 Architectures Software Developer's Manual (5 vols)*. Intel Corporation, Mar. 2010. rev. 34.

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

[7] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. C. ACM. To appear.

[8] S. V. Adve and M. D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, 1993.

[9] M. Ahamad, G. Neiger, J. Burns, P. Kohli, and P. Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[10] D. Aspinall and J. Ševčík. Formalising Java's data race free guarantee. In *Proc. TPHOLs, LNCS 4732*, pages 22–37, 2007.

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

[12] G. Boudol and G. Petri. Relaxed memory models: an operational approach. In *Proc. POPL*, 2009.

[13] 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.

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

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

[16] R. Friedman. Consistency conditions for distributed shared memories. *Israel Institute Of Technologie*, 1994.

[17] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. *WRL Research Report*, 95(9), 1995.

[18] 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.

[19] L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models part I: Definitions and comparisons. *Technical Report98/612/03, Department of Computer Science, The University of Calgary, January*, 1998. Full version of a paper in PDCS 1997.

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

[21] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, C-28(9):690–691, 1979.

[22] 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/`.

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

[24] J. Manson, W. Pugh, and S. Adve. The Java memory model. In *Proc. POPL*, 2005.

[25] R. Milner. *Communication and Concurrency*. Prentice Hall International, 1989.

[26] S. Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *Proc. ECOOP*, 2010. To appear.

[27] S. Owens, S. Sarkar, and P. Sewell. A better x86 memory model: x86-TSO. In *Proc. TPHOLs, LNCS 5674*, pages 391–407, 2009. Full version as Technical Report UCAM-CL-TR-745, Univ. of Cambridge.

[28] S. Park and D. L. Dill. An executable specification and verifier for relaxed memory order. *IEEE Trans. Computers*, 48(2):227–235, 1999.

[29] 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.

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

[31] 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.

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

[33] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, pages 27–51, 2008.