The Semantics of x86-CC Multiprocessor Machine Code

Susmit Sarkar[1] Peter Sewell[1] Francesco Zappa Nardelli[2]
Scott Owens[1] Tom Ridge[1] Thomas Braibant[2] Magnus O. Myreen[1] Jade Alglave[2]
[1]University of Cambridge [2]INRIA
http://www.cl.cam.ac.uk/users/pes20/weakmemory

Abstract

Multiprocessors are now dominant, but real multiprocessors do not provide the sequentially consistent memory that is assumed by most work on semantics and verification. Instead, they have subtle relaxed (or weak) memory models, usually described only in ambiguous prose, leading to widespread confusion.

We develop a rigorous and accurate semantics for x86 multiprocessor programs, from instruction decoding to relaxed memory model, mechanised in HOL. We test the semantics against actual processors and the vendor litmus-test examples, and give an equivalent abstract-machine characterisation of our axiomatic memory model. For programs that are (in some precise sense) data-race free, we prove in HOL that their behaviour is sequentially consistent. We also contrast the x86 model with some aspects of Power and ARM behaviour.

This provides a solid intuition for low-level programming, and a sound foundation for future work on verification, static analysis, and compilation of low-level concurrent code.

Categories and Subject Descriptors C.1.2 [*Multiple Data Stream Architectures (Multiprocessors)*]: Parallel processors; D.1.3 [*Concurrent Programming*]: Parallel programming; F.3.1 [*Specifying and Verifying and Reasoning about Programs*]

General Terms Documentation, Reliability, Standardization, Theory, Verification

Keywords Relaxed Memory Models, Semantics

1. Introduction

Problem Multiprocessor machines, with many processors acting on a shared memory, have been developed since the 1960s, but have suddenly become dominant in the last few years: laptops, desktops and servers now routinely have 2, 4, or 16 cores, and the trend to even more concurrency is set to continue. Meanwhile, the difficulty of programming concurrent systems has given rise to extensive research over the last 40 years on semantics, program logics, and so forth. This work has almost always assumed that concurrent processes share a sequentially consistent memory [23], but in fact real multiprocessors typically exhibit *relaxed*, or *weak*, *memory models*. Internally, they use sophisticated techniques to achieve high performance: hierarchies of local cache, write buffers, speculative execution, etc. The visible manifestation of these optimisations, at the assembly language level, is that individual reads and writes to memory may be reordered in surprising ways. For example, consider the following x86 program.

proc:0	proc:1
MOV [100] ← $1	MOV [200] ← $1
MOV EAX ← [100]	MOV ECX ← [200]
MOV EBX ← [200]	MOV EDX ← [100]

This consists of two straight-line programs running in parallel on two processors, proc:0 and proc:1. The instruction MOV [100] ← $1 writes value 1 to memory address 100; the instruction MOV EAX ← [100] reads a value from memory address 100 into register EAX; and so on. There is a rare, but possible, counter-intuitive execution, from an initial state with register and memory values all 0, to a final state with proc:0 EAX and proc:1 ECX both 1 (so each processor has seen its own write), but the proc:0 EBX and proc:1 EDX both 0, so each processor has seen the value that the respective memory location had *before* the other processor wrote it. In other words, proc:0 and proc:1 saw the writes to 100 and 200 in opposite orders to each other.

This means that the read and write events cannot be consistently embedded in a single linear order: x86 multiprocessors do not have a sequentially consistent semantics, and low-level programmers cannot reason in terms of simple notions of global time or causality.

To compound this difficulty, the reordering guarantees that *are* provided by multiprocessors are often poorly specified, usually only in natural-language documentation that leaves key questions ambiguous. This has led to, for example, real uncertainty about the correctness of OS spinlock implementations [3]. There is a clear need for precise semantics of real-world multiprocessors, both to inform the intuition of low-level programmers, and to provide a sound foundation for rigorous reasoning about multiprocessor programs.

Contribution In this paper we give a rigorous semantics for x86 multiprocessor programs, with an accurate relaxed memory model.

Section 2 develops an axiomatic definition of the memory model. Given a collection of memory and register read and write events, we define a valid execution to be a collection of linear orders, one per processor, of the events that it sees, subject to a number of axioms constraining the allowable reorderings. We believe this to be the first such model for x86, though there is an extensive literature on models for other processors, discussed in Section 8.

To support reasoning about programs, the memory model must be integrated with a semantics for machine instructions (a problem which has usually been neglected in the memory

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

model literature). In Section 3 we describe a semantics for core x86 instructions. We parameterise the instruction semantics over an interface of combinators (analogous to a monad), so a single definition, with all the intricacies of flag-setting, addressing modes, etc., can be used to generate both an event-based semantics integrated with the §2 memory model, and a state-based semantics for sequential programs. We also build an instruction decoding function, directly from the vendor documentation, to support reasoning about concrete machine code.

To develop confidence in the semantics, we test it, with three tools described in Section 4. The first is an implementation of the axiomatic model, calculating and displaying the sets of valid executions for small programs. This has been invaluable in building intuition for the consequences of the axioms. The second and third tools test the semantics empirically against the behaviour of actual hardware: focussing on the memory model and the instruction semantics respectively; the latter uses the sequential state-based semantics.

An understanding of the memory model is critical for low-level multiprocessor programming, e.g. for implementations of locks, software transactional memory, non-locking data-structures, garbage collectors, and compilers. For higher-level programs that are race-free (or properly synchronised) one would hope that it is sound to reason in terms of a sequentially consistent model. In Section 5 we prove a theorem to this effect, for an x86-specific notion of race.

The axiomatic style of definition is relatively easy to relate to the informal documentation, and a good basis for some metatheory, but it can be hard to see the consequences of the axioms with their global-time perspective. We therefore also develop an equivalent operational model, in Section 6: an abstract machine of queues and buffers with stepwise behaviour (at present without locked instructions).

In Section 7 we contrast the x86 behaviour with some aspects of Power and ARM behaviour, with their very different memory models, and we conclude in Section 9. Our definitions and proofs, except the §6 proofs, are mechanised in the HOL proof assistant [21]. For lack of space only key extracts are included here, but the full details are freely available online [6], together with additional discussion of the examples, our experimental data, and some of our toolset.

2. The x86-CC Axiomatic Memory Model

2.1 Scope, Criteria, and Sources

The intended scope of our semantics is typical user code: using coherent write-back memory, without exceptions, misaligned accesses, 'non-temporal' operations (e.g. MOVNTI), self-modifying code, or page-table changes. Within that, we deal with a reasonable range of instructions, as we describe in Section 3, and there should be no fundamental difficulty in covering the rest of the instruction set.

Within this scope, the semantics should be in some sense accurate. Processor vendors document 'architectures', such as Intel's Intel 64 and IA-32 [5] and AMD's AMD64 [4] (we refer to the common core of these as 'x86'). These are descriptions of the behaviour that programmers may rely on from a class of past and present (and perhaps future) processor families, each of which may contain many particular devices. They tend to vary slowly, and to be loose specifications of the device behaviour, in order to admit more rapid variation in how the devices are actually implemented. Such variations might well be observable by assembly programs, but so long as they are within the architecture-allowable behaviour (and developers have in fact programmed to that), they should be benign. Ideally, we would aim for our semantics to be sound with respect to these architectures, and hence with respect to a broad range of devices. In practice, however, the architectures are described only in informal prose and pseudocode, with a mixture of concepts from different levels of abstraction (mixing the programmer model and microarchitectural notions). As we shall see, they leave some key issues ambiguous. We can therefore only aim to be consistent with the published documents and with the behaviour of some sample devices. In the other direction, the semantics should not be substantially looser than the architecture, or it might be too weak to verify programs.

In addition to the documents cited above, our semantics is based on the *Intel 64 Architecture Memory Ordering White Paper* (IWP) [22], which states 8 one-sentence principles, illustrated by 10 small litmus-test example programs (which we label iwpNN), and the AMD documentation [4, vol.2,p.164ff], which gives similar prose and 10 largely identical examples (which we label amdNN). The IWP examples have been added to recent versions of the Intel SDM [5, vol.3A, §7.2.3], unchanged except that the proc:1 part of iwp2.3.b was removed. We have benefited also from a number of very helpful conversations with Intel staff, though of course any errors remain ours. The semantics represents our best understanding of the x86 architecture[1]; it is not a normative definition from any vendor.

2.2 Basic Types

The example in Section 1 shows that we cannot use a simple interleaving semantics, with a single transition relation over a global memory state, because different processors may see writes in different orders. Instead, we axiomatise the possible orders in which each processor sees the events of a complete execution. The semantics must be at the level of individual reads and writes, not instructions — a single x86 instruction, such as INC [100] (which increments memory location 100) may comprise multiple individual reads and writes, and it is these that are the primitive atomic events of the semantics. For example, given the program

inc-inc	proc:0	proc:1
poi:0	INC [100]	INC [100]

with initial state [100]=0, it is possible to reach a final memory state [100]=1, after an execution in which both INC instructions read 0 and wrote 1. Individual aligned memory accesses are guaranteed atomic, however.

We also include explicit events for reads and writes of processor registers (in contrast to most previous work on relaxed memory) so that information flow dependencies through registers can be calculated instead of assumed. We take accesses to the 32-bit registers and the 1-bit status flags to be atomic (it should be routine also to cover 64, 16, and 8-bit operations). The registers are as follows:

Xreg = EAX | EBX | ECX | EDX | ESP | EBP | ESI | EDI
Xeflags = X_CF | X_PF | X_AF | X_ZF | X_SF | X_OF
reg = REG32 **of** Xreg | REG1 **of** Xeflags | REGEIP

We 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

[1] But see the addendum at the end of the paper, added in press.

| LOCATION_MEM **of** address

These constructors are curried, so LOCATION_REG : proc → reg → location. To identify an instance of an instruction in an execution, we specify its processor and its index in program order (i.e., in the program with an unfolding of all branches):

iiid =⟪ *proc* : proc;
program_order_index : num⟫

An action is either a read or write of a value at some location:

dirn = R | W

action = ACCESS **of** dirn location value

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⟫

The semantics of an instruction must also record any intra-instruction causality relationships among its events, e.g., for INC [100], between the read of a value from [100] and the write of an incremented value. Furthermore, certain x86 instructions can be prefixed with a 'LOCK' byte, which guarantees that all the accesses of the instruction take place atomically. Adding LOCK prefixes to the previous example:

proc:0	proc:1
LOCK; INC [100]	LOCK; INC [100]

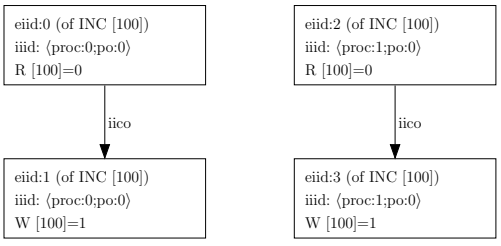
ensures that the only possible final state has [100]=2. We therefore have to record which events belong to the same locked instruction.

Collecting this together, we define an event structure E to comprise 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:

event_structure =⟪ *procs* : proc set;
events : event set;
intra_causality : event reln;
atomicity : event set set⟫

subject to various well-formedness conditions which we omit here. Note that, while these event structures are loosely inspired by those of Winskel [30], they are not exactly the same structure.

The overall semantics will be factored into two parts: the instruction semantics defines, for any program, a set of candidate event structures, and the axiomatic memory model defines, for each event structure, its valid executions (if any). For example, given the program with two unlocked INC's above, one possible event structure is below.



inc-inc: (event structure 1)

Note the intra-instruction causality (iico) edges, and the concrete values in the events. Here the atomicity PER is empty, whereas, for the locked INC example, there would be two atomicity equivalence classes, containing the events of the two instructions respectively. The semantics also includes a read and a write of the instruction pointer (or program counter) EIP, and writes of the status flags, but we suppress both in most diagrams for clarity.

2.3 View Orders

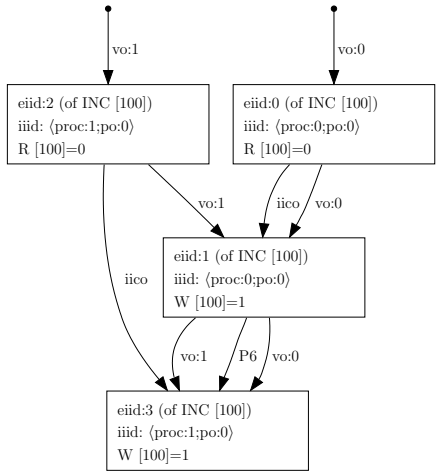
Given an event structure, a candidate execution witness consists of an initial state constraint and a processor-indexed family of view orders, together with other data that we explain in the following subsections.

type_abbrev state_constraint = location → value option

type_abbrev view_orders = proc → event reln

execution_witness =
⟪ *initial_state* : state_constraint;
vo : view_orders;
write_serialization : event reln;
lock_serialization : event reln;
rfmap : event reln⟫

A well-formed view order for processor p is a linear order over all of its events together with all the memory write events of other processors (we write viewed_events $E\ p$ for that union). The rest of this section is devoted to axiomatising when an execution witness is valid. For example, one valid execution for the previous event structure is shown below, with the two view orders labelled vo:0 and vo:1 respectively (we return to the edge labelled P6 later).



inc-inc: (event structure 1)

2.4 Preserved Program Order

Five of the eight IWP [22] principles (which we label P1–8) have a similar, and relatively straightforward, character: explicit assertions about what reorderings are or are not allowed. To make this paper self-contained we recall them here, together with their illustrative examples, before formalising them; we see how this definition is used in §2.7.

P1. LOADS ARE NOT REORDERED WITH OTHER LOADS.

P2. STORES ARE NOT REORDERED WITH OTHER STORES.

These are illustrated with a single example, which, instantiating symbolic registers and addresses to give a concrete program, and labelling with program order indices po:N, is:

iwp2.1/amd1	proc:0	proc:1
po:0	MOV [100]←$1	MOV EAX←[200]
po:1	MOV [200]←$1	MOV EBX←[100]
Required: (1:EAX=1)⇒(1:EBX=1)		

This stores to two locations, 100 and 200, on processor 0, and loads from those locations, in the other order, on processor 1. If the initial state is 0 everywhere (which we assume in all examples unless otherwise stated), then in the final state it is required that if 1:EAX=1 then 1:EBX=1. Hence, the two proc:0 stores must have been seen by proc:1 in proc:0's program order, and the two proc:1 loads must have been performed in its program order.

P3. Stores are not reordered with older loads.

iwp2.2/amd2	proc:0	proc:1
po:0	MOV EAX←[100]	MOV EBX←[200]
po:1	MOV [200]←$1	MOV [100]←$1
Forbidden: 0:EAX=1 ∧ 1:EBX=1		

This is very similar to the previous example: on processor 0, the store to location 200 cannot be reordered before the load from location 100, and, on processor 1, the write of location 100 cannot be reordered before the read of location 200.

P4. Loads may be reordered with older stores to different locations but not with older stores to the same location. There are two examples here. The first is described as illustrating "the case in which a load may be reordered with an older store, i.e. if the store and load are to different non-overlapping locations".

ipw2.3a/amd4	proc:0	proc:1
poi:0	MOV [100]←$1	MOV [200]←$1
poi:1	MOV EAX←[200]	MOV EBX←[100]
Allowed: 0:EAX=0 ∧ 1:EBX=0		

One can imagine a possible execution in which the proc:0 load from 200 is reordered before the proc:0 store to 100.

Interestingly, however, the reordering of loads with older stores is not essential for this test to give the specified outcome: in Fig. 1 we show a valid execution in which each processor sees its own events in program order, and then finally the memory write of the other processor. Below we give a new test (n1) that does require this reordering.

proc:0	proc:1	proc:2
MOV [100]←$2	MOV [200]←$1	MOV EBX←[100]
MOV EAX←[200]	MOV [100]←$1	MOV ECX←[100]
Allowed: 0:EAX=0 ∧ 2:EBX=1 ∧ 2:ECX=2		

(Consider the proc:0 view order. By P1 the W [200] 1 is before the W [100] 1, and by P6 below and the proc:2 observations, that must be before the W [100] 2. Then the R [200] 0 can only be inserted at the start, otherwise the rfmap conditions below are violated.)

The second example for P4 shows that "loads may not be reordered with a prior store to the same location".

iwp2.3.b	proc:0	proc:1
po:0	MOV [100]←$1	MOV [200]←$1
po:1	MOV EAX←[100]	MOV EBX←[200]
Required: 0:EAX=1 ∧ 1:EBX=1		

P8. Loads and stores are not reordered with locked instructions. Here there are two examples, for loads and stores respectively. They use an XCHG instruction, which exchanges two values and has an implicit LOCK prefix.

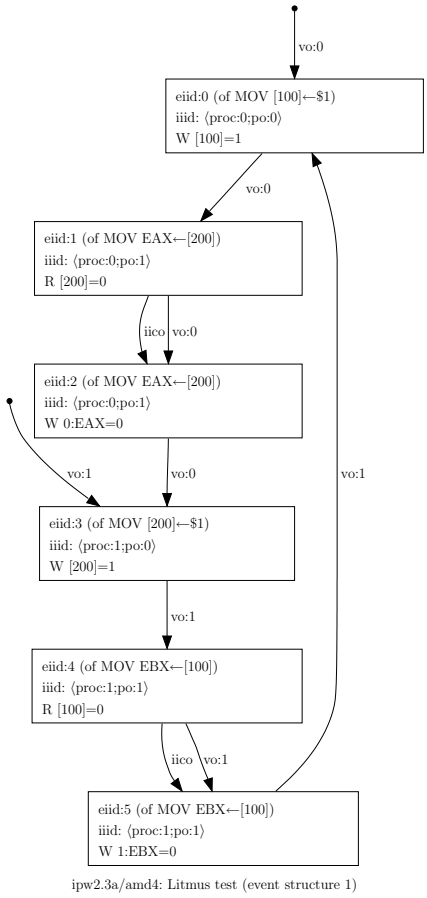


Figure 1. An iwp2.3.a/amd4 execution without reordering

iwp2.8.a	proc:0	proc:1
po:0	XCHG [100]←EAX	XCHG [200]←ECX
po:1	MOV EBX←[200]	MOV EDX←[100]
Initial state: 0:EAX= 1 ∧ 1:ECX= 1 (elsewhere 0)		
Forbidden: 0:EBX=0 ∧ 1:EDX=0		

iwp2.8.b	proc:0	proc:1
po:0	XCHG [100]←EAX	MOV EBX←[200]
po:1	MOV [200]←$1	MOV ECX←[100]
Initial state: 0:EAX= 1 (elsewhere 0)		
Forbidden: 1:EBX=1 ∧ 1:ECX=0		

Interestingly, it appears that P8 may be redundant. According to the Intel documentation [5, vol.3A, §7.1.2.2], the LOCK prefix can only be prepended to particular instructions, all of which both read and write some memory location. As those pairs of a read and write are atomic, no event from any other instruction instance can come between them. Hence, any third read or write is prevented from being reordered with the locked pair by P1.

Formalising P1–4,8 We capture the above principles by identifying the pairs of events (e_1, e_2) in program order, from an event structure E, that must not be reordered:

iwp2.6	proc:0	proc:1	proc:2	proc:3
po:0	MOV [100]←$1	MOV [100]←$2	MOV EAX←[100]	MOV ECX←[100]
po:1			MOV EBX←[100]	MOV EDX←[100]
Forbidden: 2:EAX=1 ∧ 2:EBX=2 ∧ 3:ECX=2 ∧ 3:EDX=1				

amd6	proc:0	proc:1	proc:2	proc:3
po:0	MOV [100]←$1	MOV [200]←$1	MOV EAX←[100]	MOV ECX←[200]
po:1			MOV EBX←[200]	MOV EDX←[100]
Allowed: 2:EAX=1 ∧ 2:EBX=0 ∧ 3:ECX=1 ∧ 3:EDX=0				

iwp2.7/amd7	proc:0	proc:1	proc:2	proc:3
po:0	XCHG [100]←EAX	XCHG [200]←EBX	MOV ECX←[100]	MOV ESI←[200]
po:1			MOV EDX←[200]	MOV EDI←[100]
Initial state: 0:EAX= 1 ∧ 1:EBX= 1 (elsewhere 0)				
Forbidden: 2:ECX=1 ∧ 2:EDX=0 ∧ 3:ESI=1 ∧ 3:EDI=0				

n2	proc:0	proc:1	proc:2	proc:3
po:0	MOV [200]←$1	MOV [100]←$2	MOV EAX←[100]	MOV ECX←[300]
po:1	MOV [100]←$1	MOV [300]←$1	MOV EBX←[100]	MOV EDX←[200]
Forbidden: 2:EAX=1 ∧ 2:EBX=2 ∧ 3:ECX=1 ∧ 3:EDX=0				

n3	proc:0	proc:1	proc:2	proc:3
po:0	XCHG [100]←EAX	MOV [200]←$1	MOV EBX←[200]	MOV ESI←[100]
po:1			MOV ECX←[100]	MOV EDI←[200]
po:2			MOV EDX←[100]	MOV EBP←[200]
Initial state: 0:EAX= 1 (elsewhere 0)				
Allowed: 2:EBX=1 ∧ 2:ECX=0 ∧ 2:EDX=1 ∧ 3:ESI=1 ∧ 3:EDI=0 ∧ 3:EBP=1				

Figure 2. Tests iwp2.6, amd6, iwp2.7/amd7, n2, and n3

```
preserved_program_order E =
  {(e1, e2) | (e1, e2) ∈ (po_strict E) ∧
    ((∃p r.(loc e1 = loc e2) ∧
            (loc e1 = SOME (LOCATION_REG p r))) ∨
    (mem_load e1 ∧ mem_load e2) ∨
    (mem_store e1 ∧ mem_store e2) ∨
    (mem_load e1 ∧ mem_store e2) ∨
    (mem_store e1 ∧ mem_load e2 ∧ (loc e1 = loc e2)) ∨
    ((mem_load e1 ∨ mem_store e1) ∧ locked E e2) ∨
    (locked E e1 ∧ (mem_load e2 ∨ mem_store e2)))}
```

Here po_strict relates two events on the same processor if the first strictly precedes the second in program order. The various auxiliary functions used should be clear; we refer the reader to the HOL for their formal definitions. We also have to constrain register read and write events. The first disjunct prevents reordering of reads or writes to the same register, which seems the most conservative reasonable choice.

2.5 Reads-from Information Flow

Our event structures do not constrain the values of memory or register read events. Intuitively, one would expect the value read to be that of the most recent write to the same location, or that of the initial state if there is none — where 'recent' is with respect to the relevant view order. It turns out that to capture the appropriate notion of causality (c.f. §2.7 below) we need to consider not just the values but the intensional property of *which* write each read reads from. We define the candidate reads-from maps for an event structure E, each *rfmap* identifying, for some of the read events, a write event to the same location with the same value. Other read events are taken to read from the initial state.

```
reads_from_map_candidates E rfmap =
  ∀(ew, er) ∈ rfmap. er ∈ E.events ∧ ew ∈ E.events ∧
    ∃l v.(er.action = ACCESS R l v) ∧
         (ew.action = ACCESS W l v)
```

Two conditions check that these are consistent with a view order and initial state, ensuring that there are no intervening writes to the same location between an (ew, er) reads-from pair, or before an er that reads from the initial state:

```
check_rfmap_written E vo rfmap =
∀p ∈ (E.procs).
  ∀(ew, er) ∈ (rfmap|_(viewed_events E p)).
    ∀ew' ∈ (writes E).
      ¬(ew = ew') ∧ (ew, ew') ∈ (vo p) ∧ (ew', er) ∈ (vo p)
        ⟹ ¬(loc ew = loc ew')

check_rfmap_initial E vo rfmap initial_state =
∀p ∈ (E.procs).
  ∀er ∈ (((reads E) \ (range rfmap))
         ∩ viewed_events E p).
    ∃l v.(er.action = ACCESS R l v) ∧
         (initial_state l = SOME v) ∧
         ∀ew' ∈ writes E.
           (ew', er) ∈ (vo p) ⟹ ¬(loc ew' = loc er)
```

2.6 Total Store and Lock Orders

The x86 has two global ordering properties, P6 and P7.

P6. IN A MULTIPROCESSOR SYSTEM, STORES TO THE SAME LOCATION HAVE A TOTAL ORDER.

Test iwp2.6, in Fig. 2, illustrates the fact that writes by two different processors to the same location must be seen (by all processors) in the same order. This complements P2, which required writes by a single processor to any locations to be seen (by all processors) in program order. However, in the remaining possibility, of writes by two different processors to two different locations, these writes *can* be seen in different orders. This is illustrated by the example of Section 1 (which is Test iwp2.4/amd9) in the special case where the writing and observing processors are the same, and by Test amd6 in Fig. 2 (essentially Boehm and Adve's IRIW [13]), in the special case where they are

different. We formalise this by defining the candidate per-location write serialisations:

write_serialization_candidates E $cand$ =
$(\forall (e_1, e_2) \in cand.$
  $\exists l. e_1 \in (\text{get\_l\_stores } E\ l) \wedge e_2 \in (\text{get\_l\_stores } E\ l)) \wedge$
$(\forall l.\ \text{strict\_linear\_order}(\ cand|_{(\text{get\_l\_stores } E\ l)})$
  $(\text{get\_l\_stores } E\ l))$

where get_l_stores E l gives the memory write events to location l (and is empty if l is not a memory location), and strict_linear_order R A iff R is a strict linear order over A.

P7. In a multiprocessor system, locked instructions have a total order.

This is illustrated by Test iwp2.7/amd7, in Fig. 2, in which proc:2 and proc:3 have to see the two locked XCHG instructions in the same order. It is a property of instructions, not events, so we formalise it by defining the candidate orders based on arbitrary linearisations of the locked instructions, projected down onto their events.

lock_serialization_candidates E =
  let lin_ec = strict_linearisations $E.atomicity$ **in**
    $\{\{(e_1, e_2) \mid \exists (es_1, es_2) \in lin. e_1 \in es_1 \wedge e_2 \in es_2\}$
    $\mid lin \in lin\_ec\}$

The following condition checks that locked instructions really are atomic, i.e. that there are no intervening events within each in any view order.

check_atomicity E vo =
  $\forall p \in (E.procs). \forall es \in (E.atomicity).$
    $\forall e_1\ e_2 \in es. (e_1, e_2) \in (vo\ p) \implies$
      $\forall e. (e_1, e) \in (vo\ p) \wedge (e, e_2) \in (vo\ p) \implies e \in es$

The documentation is silent on the question of whether a locked instruction and a write, on two different processors, can be seen elsewhere in different orders, as illustrated in Test n3 of Fig. 2, and we have received conflicting informal opinions. We therefore opt for the more conservative (looser) alternative, permitting it.

2.7 Causality

Treating causality correctly is the key issue in defining the semantics: without some causal consistency constraint, the semantics would be much too liberal, with the view orders of different processors allowed to vary wildly. However, the prose vendor documentation is particularly ambiguous on this point. We have ([22]):

> P5. In a multiprocessor system, memory ordering obeys causality (memory ordering respects transitive visibility).

and ([4, vol.2,p.166]):

> *"Dependent stores between different processors appear to occur in program order [...] A globally consistent ordering is maintained for such stores."*

but what "causality", "transitive visibility", or "dependent stores" mean is, a priori, unclear. Test iwp2.5/amd8 below shows transitivity in one specific case, from a reads-from relationship to a preserved-program-order relationship.

proc:0	proc:1	proc:2
MOV [100]←$1	MOV EAX←[100]	MOV EBX←[200]
	MOV [200]←$1	MOV ECX←[100]
Required: (1:EAX=1 ∧ 2:EBX=1)⇒(2:ECX=1)		

After much discussion, we believe that x86 causality is also transitive through the write serialisation and lock serialisation relations of §2.6. Test n2 of Fig. 2 illustrates the former: transitivity through preserved program order of the proc:0 and proc:1 events and the write serialisation for [100] (which has W [100] 1 before W [100] 2 by the proc:2 observations). We also include the intra-instruction causality relation, e.g. to include the edge from an R [100] v to a W [100] $(v+1)$ of an INC [100] instruction, the intensional reads-from information flow of §2.5, and the preserved program order of §2.4. We propose the following definition of causality, for event structure E and execution witness X:

happens_before E X =
  $E.intra\_causality \cup$
  $(\text{preserved\_program\_order } E) \cup$
  $X.write\_serialization \cup$
  $X.lock\_serialization \cup$
  $X.rfmap$

This appears to match the vendors intentions. We require that all view orders are consistent with happens-before (implicitly transitively closed in the acyclic check):

check_causality E vo (happens_before E X) =
  $\forall p \in (E.\text{procs}).$
    acyclic((strict($vo\ p$)) $\cup$ (happens_before E X))

Note that there is a single happens-before relation which includes the preserved program order edges of all processors, and it may constrain events in the view order of one processor via a transitive path through other processor's read events, which do not occur in that view order.

Models of architectures with weaker orders sometimes involve visibility of other processors' reads [7, 11], but this seems to be unnecessary for the fragment of x86 we consider. Other models have sometimes also used the dual of our reads-from edges (e.g. [9]): suppose there is a reads-from edge from ew to er, and there is some later (in the write serialisation order) ew' to the same location, then one could think of a 'from-read' edge from er to ew' (with an eye to the cache coherency protocol of some implementation, in which cache-line ownership is transferred in a linear order). However, adding such edges to happens-before is inconsistent with the iwp2.4/amd9 test in §1.

2.8 Valid Executions

Finally, we can collect together these conditions, defining the valid execution witnesses X for an event structure E:

valid_execution E X =
  view_orders_well_formed E $X.vo \wedge$
  $X.write\_serialization \in$ write_serialization_candidates $E \wedge$
  $X.lock\_serialization \in$ lock_serialization_candidates $E \wedge$
  $X.rfmap \in$ reads_from_map_candidates $E \wedge$
  check_causality E $X.vo$(happens_before E X) $\wedge$
  check_rfmap_written E $X.vo$ $X.rfmap \wedge$
  check_rfmap_initial E $X.vo$ $X.rfmap$ $X.initial\_state \wedge$
  check_atomicity E $X.vo$

We call this memory model *x86 causal consistency*, or *x86-CC* for short. It gives the correct results for all the tests that we have seen, and is, to the best of our knowledge, consistent with all the published architecture documents.

For finite valid executions there is an unambiguous notion of final state, with the final state for each memory location determined by the last write in its write serialisation, and the final state for each register determined by the last write in the relevant view order. Note also that, in a valid execution,

the *rfmap*, *write_serialization*, and *lock_serialization* are uniquely determined by the view orders.

2.9 Nice Executions

The axiomatic semantics was defined conservatively, taking care not to impose restrictions absent from the documentation. For example, register reads and writes on different registers can be arbitrarily reordered, and also reordered arbitrarily with memory reads and writes as long as the intra-instruction causality is respected. However, we can prove that some of this reordering is not observable to the programmer, showing that the architecture is tighter than a naive reading would suggest, and providing a more convenient model for future software verification.

One cannot require that all events of a processor are always seen by itself in an order consistent with program order, as Test n1 of §2.4 shows that read speculation is observable, but one can require all events except memory writes to be observed by the issuing processor in an order consistent with program order.

$$\text{nice\_execution } E\ X = \forall p \in (E.procs).\ (\text{po\_strict } E)|_{(\text{viewed\_events } E\ p \setminus \text{mem\_store})} \subseteq (X.vo\ p)$$

THEOREM 1 (Valid executions can, w.l.g., be nice).

$$\forall E\ X.(\text{well\_formed\_event\_structure } E \wedge \text{valid\_execution } E\ X) \implies \exists X'.\ \text{valid\_execution } E\ X' \wedge \text{nice\_execution } E\ X' \wedge (X' \llbracket\ vo := (\lambda p.\{\}) \rrbracket = X \llbracket\ vo := (\lambda p.\{\}) \rrbracket)$$

HOL proof outline: For each processor, we construct a new view order, which preserves the order of memory events and locked events, and gathers register events into program order. The new view order is constructed inductively. At stage n, all register events or local memory reads with program order n or less are ordered. Care must be taken to ensure that any local or foreign writes that are not compelled to appear at any particular stage appear somewhere in the new view order. All clauses of valid-execution are checked inductively; to check compatibility with happens-before it suffices to ensure there are no happens-before reverse edges at each stage.

2.10 Instruction Pointer Events, Branches, and Speculation

The semantics deals smoothly with control flow instructions (jumps, conditional branches, call, and return) without any special machinery. The axiomatic model treats the instruction pointer (EIP) like any other register, while the instruction semantics for a typical instruction gives event structures with an EIP read and a write of a suitably incremented value. These are linked by the intra-causality relation, but (except for CALL, RET, etc.) are unrelated to other events of the instruction. A program-semantics condition ensures that the values of any EIP reads of an instruction match its address. The x86 allows very little local reordering (preserved_program_order is rather strong), but this permits a load to be reordered before a store to a different address even across a branch, which we believe to be accurate (the documentation is unclear on this point).

2.11 Alignment

The older Intel and AMD documents only discuss programs that do not make unaligned accesses, and for the moment that assumption is built into our model. But this means that one cannot reason at all about incorrect or malicious code, so a more complete (but loose) specification is highly desirable. We believe that it would be sound w.r.t. current devices to treat unaligned accesses as an unordered set of byte accesses, and it would be straightforward to add this to our model. The current Intel SDM [5, vol.3A, §7.1.1] appears to state that, on P6 or later processors, unaligned 16-, 32-, and 64-bit accesses within a cache line are atomic, but also [§7.2.3.1] that unaligned instructions may be implemented with multiple accesses.

The documents are also silent about aligned accesses to different addresses within a cache line. It may be that current devices actually provide strong ordering for such accesses, and exposing this would permit algorithms to save the high cost of locked instructions in some circumstances. It could be modelled by taking write serialisation orders per (minimal-sized) cache line, rather than per location.

2.12 Fences

The x86 also includes fence instructions, or memory barriers, LFENCE, SFENCE, and MFENCE. For the coherent write-back fragment that we consider, without non-temporal operations, we understand the first two to be (perhaps costly) no-ops. For MFENCE, the documentation is less clear, and so we did not include it in our HOL model. IWP [22] does not mention it, while the Intel SDM [5, vol.2A,p3-624;vol.3A,§7.2.5] is ambiguous (the text could be read as asserting that MFENCEs of different processors are globally serialised). The most conservative (weakest) plausible semantics is that an MFENCE simply ensures that program order is preserved around it, preventing the reordering of a load before a store that we saw in Test iwp2.3.a/amd4 of §2.4. Test amd5 (like iwp2.3.a/amd4 but with an MFENCE after each store) confirms this holds on AMD64, and informal discussion suggests it also does on Intel 64/IA-32. It would be easy to add this to the model, strengthening the preserved_program_order definition of §2.4.

AMD64 [4] includes one final test, amd10 (an analogue of iwp2.3.1/amd4) which shows a strictly stronger semantics, but just how much stronger is unclear (consider, for example, analogues of Test amd6 in Fig. 2 with one or more MFENCEs).

amd10	proc:0	proc:1
po:0	MOV [100]←$1	MOV [200]←$1
po:1	MFENCE	MFENCE
po:2	MOV EAX←[100]	MOV ECX←[200]
po:3	MOV EBX←[200]	MOV EDX←[100]
Forbidden: 0:EBX=0 ∧ 1:EDX=0		

2.13 Recovering Sequential Consistency

In most cases, one would like to program in an idiom that guarantees sequentially consistent behaviour, ensuring that the reorderings that the memory model permits are not observable. We return to this in §5, but mention two extreme possibilities here. Contrary to what might be expected, adding an MFENCE between every instruction does *not* suffice, according to the x86-CC semantics above: this would still permit processors to see different store orders. However, programming exclusively with locked instructions would suffice, as the *lock_serialization* order would determine an SC execution.

3. Instruction Semantics

We now give the other half of the semantics: we define the possible event structures for a program; combining this with the memory model to give the possible event

structures with their valid executions. At present we cover the following instructions: data transfer MOV, CMOVE, CMOVNE, XADD, XCHG, CMPXCHG, LEA; binary operations ADD, AND, CMP, OR, SUB, TEST, XOR, SHR, SAR, SHL; unary operations INC, DEC, NOT, NEG; stack operations POP, PUSH, PUSHAD, POPAD; and control transfers JUMP, CALL, RET, LOOP. We cover all the various 32-bit addressing modes, including indexing, scaling, etc.

3.1 Decoding

The first step is to decode a machine code program, a code memory containing bytes, of type program_word8 = address → word8 option, to an abstract syntax program, giving the instructions Xinst (and their lengths) at each address: program_Xinst = (address → (Xinst ∗ num) option). The vendor documentation includes tables with one to 50 or so lines for each instruction, e.g.

```
" 8B /r      | MOV r32, r/m32 ";
" B8+rd id   | MOV r32, imm32 ";
```

giving symbolic expressions for their opcodes. For example, `B8+rd id` represents an opcode with a first byte `B8` added to a code for the 32-bit register `r32`, followed by a 4-byte immediate operand for the `imm32`. To make the semantics scalable, without introducing many errors, we formalised the interpretation of these encodings inside the HOL logic, and built a HOL decoding function by directly copying the relevant lines from the manual into the HOL script. (We found one error in the Intel manual in the process: the `id` in the second line shown is actually omitted there [5, vol.2A,p3-640], highlighting the need for testing.)

3.2 Instructions

A single x86 instruction can involve a complex pattern of register and memory accesses. In defining the possible event structures for an instruction, with the right intra-instruction causality relation among these accesses, we have to avoid over-sequentialising them. For example, two independent reads should be unrelated, whereas an [EAX] operand resolves into a register read of EAX followed by a memory read of that address. Moreover, the pattern of accesses (not just their values) can depend on the values read, and to keep the semantics manageable we have to deal as uniformly as possible with all the various addressing modes and with the various binary and unary operations. We must also accommodate loose specification of values, for flag values that are explicitly undefined in the architecture.

We express the semantics in terms of a small 'microcode' language of combinators, analogous to a monad for a type constructor $'a$ M, but with both sequential and parallel composition:

seqT : $'a$ M → ($'a$ → $'b$ M) → $'b$ M
parT : $'a$ M → $'b$ M → ($'a * 'b$)M
constT : $'a$ → $'a$ M
failureT : unit M
mapT : ($'a$ → $'b$) → $'a$ M → $'b$M
lockT : unit M → unit M
write_reg : iiid → Xreg → word32 → unit M
read_reg : iiid → Xreg → word32 M
write_eip : iiid → word32 → unit M
read_eip : iiid → word32 M
write_eflag : iiid → Xeflags → bool option → unit M
read_eflag : iiid → Xeflags → bool M
write_m32 : iiid → word32 → word32 → unit M
read_m32 : iiid → word32 → word32 M

For example, for binary operation XBINOP *binop_name ds*, with destination and source *ds*, at instruction instance *ii*, and *len* bytes long, we have (using various auxiliaries):

x86_exec *ii* (XBINOP *binop_name ds*) *len* = parT_unit
(seqT (read_eip *ii*) (λ*x*. write_eip *ii* (*x* + *len*)))
(seqT
  (parT (read_src_ea *ii ds*) (read_dest_ea *ii ds*))
  (λ((*ea_src*, *val_src*), (*ea_dest*, *val_dest*)).
    write_binop *ii binop_name val_dest val_src ea_dest*))

The event structure semantics implements the combinators for $'a$ M below (threading through a gensym eiid_state to make eiid's unique by construction). The seqT and parT combinators both build the set of event-structure unions of pairs of event structures from their arguments, with seqT adding intra-causality edges.

$'a$ M = eiid_state → ((eiid_state ∗ $'a$ ∗ event_structure)set)

3.3 Sequential Semantics

We also build a more directly executable semantics for sequential programs, simply re-implementing the combinators for a state monad while keeping the body of the instruction semantics unchanged:

$'a$ M = x86_state → ($'a$ ∗ x86_state) option
x86_state = (Xreg → word32)
  ∗(word32) (* EIP *)
  ∗(Xeflags → bool option)
  ∗(word32 → word8 option)

3.4 Programs

Finally, to define the possible event structures for a decoded program, we identify the well-formed run skeletons — sequences (finite or infinite, and downclosed) of addresses of instructions for each processor:

proc → (program_order_index → address option)

For each run skeleton, we first calculate the sets of event structures for each instruction instance it contains, then take the event-structure union of each possible choice thereof. Combining this with the axiomatic model to give valid execution witnesses, the overall semantics has the type below.

x86_semantics : program_word8 → state_constraint →
  (run_skeleton ∗ program_Xinst
    ∗((event_structure ∗ (execution_witness set))set))set

This is significantly more involved than a typical sequentially consistent interleaving semantics — largely because the values read in a valid execution are constrained by the axiomatic memory model, which is in terms of the possible orderings of all the events of a putative execution, instead of being values known at a particular time, that one could simply provide to the instruction semantics. Additional complexity arises from dealing with concrete located machine code rather than assembly language, which is necessary to build correct EIP values at call points.

THEOREM 2. *The event structures built above are all well-formed.* [Proof: HOL, with a large automated case analysis]

4. Testing the Semantics

4.1 Executing the axiomatic memory model

In developing the axiomatic memory model of §2, it was vital to explore the consequences of the axioms for example programs. However, this quickly becomes too complex to do by hand. The litmus tests in §2 have around 6–10 instructions on $p = 2$–4 processors, which generate $n_e = 10$–20 events, and the naive number of candidate execution witnesses is roughly $(n_e!)^p$. We built a `memevents` tool to test the semantics, which is efficient enough to run on all the examples shown, and which confirms their stated behaviours in the axiomatic model. It also produces pictorial representations of the possible view orders and executions, as in the simple example shown in Fig. 1. To test the impact of individual conditions of the axiomatic model, selected checks can be turned off one by one. The tool is written in OCaml. It is heavily parameterised, both over the microcode language of §3.2, which can be instantiated to produce an event structure and a state monad semantics, and over the concrete architecture, which can be instantiated with the x86 structures described here or with Power and ARM semantics, which are under construction.

The OCaml code has been checked, by hand, to correspond with the HOL definition. In future work, we plan to instantiate the HOL definition with a third implementation of the microcode language, for symbolic execution, and to use the HOL code generation facilities to build the checker core directly.

4.2 Validation of the memory model

To validate the memory-model semantics against actual hardware, we built a `litmus` tool. Given a test specified with a syntax similar to that used in this paper, this tool initialises the machine state (memory and registers), spawns the threads that compose the test and compares the final state with the constraint specified by the test. Care is taken to use memory locations in different cache lines, to pre-fill caches and write buffers, and to synchronise the threads, and each test is repeated many times, to maximise the probability of observing non-sequentially consistent behaviours.

We tested the litmus tests of §2 on several multiprocessor machines[2]. In all cases, the results we observed were admitted by our semantics. On all machines, we observed the non-sequentially consistent behaviours of tests iwp2.3.a/amd4 and iwp2.4/amd9, e.g. between 5 and 5000 times out of 200000 runs for the latter. We did not find witnesses for the reorderings of tests n1, amd6, and n3. It may be that these tests were not repeated sufficiently, or that the tool does not stress the memory subsystem properly to highlight these behaviours, but it may also be that the particular processors we tested do not exploit these reorderings, even if they are allowed by the architecture (we believe that many processors actually provide a much stronger TSO-based model). The gap between particular devices and the architecture means that one can conclude little from an absence of witnesses.

4.3 Validation of the sequential semantics

To validate the details of the instruction semantics (decoding, arithmetic details, etc.), which are largely orthogonal to the memory model, we tested them against a Pentium 4 processor. For efficiency, this uses the sequential semantics of §3.3; the checking is done entirely within HOL, for high confidence. We implemented an `x86sem` tool that, given an x86 instruction, builds a valid assembler program that dumps the state of the machine (including registers, flags, stack, and memory) immediately before and after the instruction being tested. For a simple example, one instance of testing the instruction MOV EAX←EBX generated the following HOL conjecture:

$$
\begin{array}{l}
(\text{XREAD\_REG EBX } s = \texttt{0x6F5BE65Bw}) \implies \\
(\text{XREAD\_EIP } s = \texttt{0x804848Bw}) \implies \\
(\text{XREAD\_MEM } \texttt{0x804848Bw } s = \text{SOME } \texttt{0x89w}) \implies \\
(\text{XREAD\_MEM } \texttt{0x804848Cw } s = \text{SOME } \texttt{0xD8w}) \implies \\
(\text{XREAD\_REG EAX}(\textbf{the}(\text{X86\_NEXT } s)) = \texttt{0x6F5BE65Bw}) \wedge \\
(\text{XREAD\_REG EBX}(\textbf{the}(\text{X86\_NEXT } s)) = \texttt{0x6F5BE65Bw}) \wedge \\
(\text{XREAD\_EIP}(\textbf{the}(\text{X86\_NEXT } s)) = \texttt{0x804848Dw})
\end{array}
$$

This states that, for all s : `x86_state`, if the memory pointed by the instruction pointer contains the encoding of MOV EAX←EBX, i.e. `89D8`, and EBX has the given initial state, the machine evolves in the §3.3 semantics (function X86_NEXT) to a state where EAX contains the double word from EBX. Such conjectures can then be automatically proved by the HOL automation, using an appropriate set of simplification rules.

The tool covers all the instructions defined in HOL (except, at present, PUSHAD, POPAD, LEA, SHR, SAR, SHL), including direct and indirect memory addressing modes, and includes a random initialisation of the state of the machine, chosen to prefer corner cases. The semantics has been tested on 4600 random instruction instances, also generated from the opcode tables (about 75 per line). The tool highlighted several mistakes in the decoding functions, but did not reveal inconsistencies between the HOL sequential semantics and the behaviour of the processor.

5. Data-race-free Programs

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 need only consider sequentially consistent executions. For example, one can consider 'properly synchronised' programs, in which shared accesses are protected by locks. Indeed, memory models have sometimes been defined in these terms [8]. For a processor ISA, we prefer to define a memory model that is applicable to arbitrary programs, to support reasoning about low-level code (including implementations of locks, for example), and have results about well-behaved programs as theorems above it.

Say a valid *sequential execution* for an event structure E is a linear ordering so on its events such that all instructions in an atomic group appear uninterrupted, and such that the unique execution witness built from that order is valid according to the axiomatic memory model. Such behaviours are manifestly sequentially consistent.

$$
\begin{array}{l}
\text{sequential\_execution } E\ so = \\
\quad \text{linear\_order } so\ E.events \wedge \\
\quad (\forall (es \in (E.atomicity))(e_1 \in es)(e_2 \in es)e. \\
\qquad (e_1, e) \in so \wedge (e, e_2) \in so \implies e \in es)
\end{array}
$$

$$
\begin{array}{l}
\text{valid\_sequential\_execution } E\ initial\_state\ so = \\
\quad \text{sequential\_execution } E\ so \wedge \\
\quad \text{valid\_execution } E(\text{so\_to\_exec\_witness } E\ initial\_state\ so)
\end{array}
$$

The first step is to show that if an execution is valid in the memory model then there is a similar valid sequential

[2] One machine was equipped with two Intel Xeon CPUs, one with four Quad-Core AMD Opteron CPUs, one with 4 Dual-Core AMD Opteron CPUs, and one with an Intel Core 2 Quad CPU.

execution, as long as the former has no data races (Theorem 3 below). An execution has a data race if there is a pair of memory access events to the same location that can compete, i.e. that are unrelated by happens-before. This is an intensional and x86-specific notion of data race: note that one event must be a read and the other a write—two writes to the same memory location can never be a data race because of the *write_serialization* ordering. Note also that two locked events from different instructions can never compete, and a write followed by a read of the same memory address in some view order must be related by happens-before, and so do not compete.

$$\begin{array}{l}\text{competes } E\ X = \\ \{(e_1, e_2) \mid \neg(e_1 = e_2) \wedge (\text{loc } e_1 = \text{loc } e_2) \wedge \\ ((e_1 \in \text{writes } E \wedge \text{mem\_store } e_1 \wedge e_2 \in \text{reads } E) \vee \\ \quad (e_2 \in \text{writes } E \wedge \text{mem\_store } e_2 \wedge e_1 \in \text{reads } E))\} \\ \setminus ((\text{happens\_before } E\ X)^+ \cup ((\text{happens\_before } E\ X)^+)^{-1})\end{array}$$

$$\begin{array}{l}\text{race\_free } E\ X = \forall e_1\ e_2 \in (E.\mathit{events}). \\ \quad \neg((e_1, e_2) \in \text{competes } E\ X)\end{array}$$

Theorem 3 (Sequential Order).

$$\begin{array}{l}\forall E\ X.\ \text{well\_formed\_event\_structure } E \wedge \textbf{finite } E.\mathit{events} \wedge \\ \text{race\_free } E\ X \wedge \text{valid\_execution } E\ X \\ \implies \exists \mathit{so}. \\ \text{valid\_sequential\_execution } E\ X.\mathit{initial\_state}\ \mathit{so} \wedge \\ (\text{happens\_before } E(\text{so\_to\_exec\_witness } E\ X.\mathit{initial\_state}\ \mathit{so})) \\ \quad \subseteq (\text{strict } \mathit{so}) \wedge \\ (X.\mathit{write\_serialization} = \text{so\_to\_write\_serialization } \mathit{so}) \wedge \\ (X.\mathit{lock\_serialization} = \text{so\_to\_lock\_serialization } E\ \mathit{so}) \wedge \\ (X.\mathit{rfmap} = \text{so\_to\_rfmap } E\ \mathit{so})\end{array}$$

HOL proof outline: Induction on the size of E. We remove a happens-before-maximal element e from E, inductively sequentialise the rest as so', and add e to so' as the maximal element. We rely on data-race freedom only in showing that the resulting rfmap is unchanged (which in turn ensures that check_rfmap_written and check_rfmap_initial pass). To ensure that atomic events appear contiguously, we choose for e an event that is in an atomicity set if and only if that entire set is happens-before-maximal, with respect to outside instructions—the *lock_serialization* ordering ensures that this condition can always be satisfied.

Two ways to strengthen Theorem 3 appear desirable at first sight, but are not true. Consider first whether each processor's view order when restricted to local events (i.e., not including others' memory writes) can have the same order as the sequential order. Test iwp2.3.a/amd4 provides a counter-example.

The sequential order also cannot be made to keep the events of non-atomic instructions adjacent (the INC/INC example of §2 shows that in a racy situation, but it remains true even for race-free executions). In a situation similar to iwp2.3.a/amd4, but where the both events of each processor are in the same instruction, if neither read is to the initial state, then it is easy to check that neither possible instruction atomic sequential execution preserves the rfmap (and is hence invalid since the written and initial values differ). One can set up such a situation in practice using PUSH instructions after setting the ESP register.

Theorem 3 allows the existence of valid executions for an event structure to be established using sequential reasoning only after all data races have been ruled out with respect to the weak memory model. Theorem 4 below shows that data-race freedom can also be established using only sequential reasoning. An event structure is *sequentially data-race free* if all of its sequential executions are data-race free. Because we are working in terms of a concrete event structure, we must also consider prefixes of E. Otherwise, an event structure with no sequential executions would be trivially sequentially race free, and therefore by Theorem 4 sequentialisable, a contradiction. Notice that the notion of sequential data-race freedom does not depend on the view order of X.

$$\begin{array}{l}\text{prefixes } E\ X = \{E' \mid \text{sub\_event\_structure } E'\ E \wedge \forall e_1\ e_2. \\ \quad e_2 \in E'.\mathit{events} \wedge (e_1, e_2) \in (\text{happens\_before } E\ X) \implies \\ \quad e_1 \in E'.\mathit{events}\}\end{array}$$

$$\begin{array}{l}\text{sequential\_race\_free } E\ X = \forall (E' \in (\text{prefixes } E\ X))\mathit{so}. \\ \text{valid\_sequential\_execution } E'\ X.\mathit{initial\_state}\ \mathit{so} \implies \\ \quad \forall e_1\ e_2.\neg((e_1, e_2) \in \text{competes } E' \\ \qquad (\text{so\_to\_exec\_witness } E'\ X.\mathit{initial\_state}\ \mathit{so}))\end{array}$$

Theorem 4 (Data race freedom).

$$\begin{array}{l}\forall E\ X.\ \text{well\_formed\_event\_structure } E \wedge \textbf{finite } E.\mathit{events} \wedge \\ \text{sequential\_race\_free } E\ X \wedge \text{valid\_execution } E\ X \\ \implies \\ \text{race\_free } E\ X \wedge [\textit{the conclusion of Thm. 3}]\end{array}$$

HOL proof outline: Complete induction on the size of E, showing that if E is sequentially data-race free it is data-race free, then using Theorem 3. For the first part, assume for a contradiction that there is a data race on two events e_1 and e_2 for a particular execution witness. Consider the prefix of E that consists of only those two events and those that precede them in the happens-before order. Call it E', and assume w.l.o.g. that e_1 is the write. Remove e_1 from E' and by induction sequentialise the remainder as so' (here we use the fact that a prefix of a sequentially data-race free program is still sequentially data-race free). Add e_1 to the end of so as above and check that this is a valid sequential execution of E', with a data race between e_1 and e_2, contradicting the assumption of sequential data-race freedom (again for prefixes too). If e_1 is in an atomicity set, it is necessary to ensure that no other element of the set in E' happens before an element of E' not in the set. This could fail if an instruction had some events inside an atomicity set and others not inside an atomicity set. However, in the x86 architecture no instructions are partially atomic.

Ultimately, this result should be lifted to an interleaving transition-system semantics over the `x86_state` of §3.3, so that race freedom can be determined without reference to the event structures semantics at all. We have defined such a semantics, as a further instantiation of the microcode combinators, but leave the proof to future work.

6. The Abstract-Machine Memory Model

The global style of the axiomatic model, in terms of possible orderings of events in a complete execution, fits well with the informal statements of the vendor documentation, and with most previous work on relaxed memory. However, it is difficult to relate it to operational intuitions of machines. We therefore develop an alternative abstract-machine characterisation of the axiomatic model, with the same set of possible behaviours. At present this covers non-locked instructions only, though we believe that the extension to cover them is reasonably straightforward.

Given Theorem 1 and preserved_program_order, the only reordering the machine must permit is of memory writes after independent reads. Memory writes from the same processor must be observed in program order, and memory writes to the same location must be observed in the same write serialisation order. We capture this in the machine with two pieces of state: pending FIFO queues of write operations $F\ p\ q$, for writes issued by processor p to be seen on processor q, and per-location write serialisations $G\ a$. On issuing a write, processor p enqueues the writes on each queue $F\ p\ \_$. The write is considered observed by q when it is dequeued

from $F\ p\ q$. Since there is a queue $F\ p\ p$, processor p may choose to delay observing its own write. The FIFO nature of the $F\ p\ q$ queue ensures that writes issued by p are observed in the same order as they are issued. Dequeueing is subject to the constraint that all $G\ a$ predecessors have already been observed, and adds the current write to the global order $G\ a$ if it is not already present.

Unfortunately, this alone does not suffice to ensure that the transitive closure of the happens-before relation is respected. The easiest way to do so is to build up the happens-before relation incrementally, and check that all happens-before predecessor events have been observed as a precondition of observing events. The machine is still operational, in the sense that it enjoys the progress property below, and so backtracking is not required. However, it would still be fairly costly to implement, so this should be considered only a first step.

THEOREM 5 (Machine progress). *For the transition system defined by the machine for a program, either the machine has a τ transition, or it can make a visible transition matching the event structure of the program, or no processor has any more events and the machine queues are empty.*

It also matches the axiomatic model precisely.

THEOREM 6 (Machine correctness). *1. For any finite nice valid execution of the axiomatic semantics, there is a corresponding trace of the machine.*

2. The execution witnesses built from complete traces of the machine, for finite well-formed E, are valid executions in the axiomatic semantics.

The detailed operational semantics of the machine (in HOL), and the (hand) proofs of the above results, are available [6].

Given such a machine, it should be feasible to build a demonic x86 emulator, with aggressive reordering, so that low-level code (e.g. lock-free datastructure implementations) can be tested against the architecture, rather than just against particular devices.

7. Power and ARM Contrasts

To give a flavour of the large design space in which the x86 relaxed memory model lies, we contrast it briefly with the behaviour of the Power and ARM multiprocessors (which have broadly similar memory models). Preliminary HOL definitions of these memory models are available [6].

First, the Power and ARM have weaker program-order preservation constraints. In the Power analogue of Test ipw2.1/amd1, we can observe (using our `litmus` tool) that a final outcome with the first register holding 1 and the second 0 is possible. By adding dataflow dependencies to one or other processor, we can confirm that both load/load and store/store reorderings are possible. We can observe also that loads from two addresses into the same register can be reordered.

Second, transitivity, as in analogues of Test iwp2.5/amd8, is explicitly not guaranteed for ARM, as noted by Chong and Ishtiaq [16], and we believe also for Power.

Third, the existence of duplicate (or shadow) registers in Power can be observed by the programmer. For example, consider the test below, adapted from Adir et al. [7], with read and written values annotated. On proc:1 there is (preserved) data dependency from each instruction to the next, and similarly on proc:0 between the lwz and mr, and between the li and stw. Because the mr $r_2 \leftarrow r_1$ and li $r_1 \leftarrow 1$ both involve r_1, one might expect a preserved program order between them, which would lead to a cycle in the view orders. In the architecture (though we have not yet observed this), the r_1 used in the lwz and mr, and the r_1 used in the li and stw, can be two different duplicates.

proc:0			proc:1		
$\{x{=}2\}$	lwz $r_1 \leftarrow [x]$	$\{r_1{=}2\}$	$\{y{=}1\}$	lwz $r_3 \leftarrow [y]$	$\{r_3{=}1\}$
$\{r_1{=}2\}$	mr $r_2 \leftarrow r_1$	$\{r_2{=}2\}$	$\{r_3{=}1\}$	addi $r_3 \leftarrow r_3$, 1	$\{r_3{=}2\}$
	li $r_1 \leftarrow 1$	$\{r_1{=}1\}$	$\{r_3{=}2\}$	stw $r_3 \rightarrow [x]$	$\{x{=}2\}$
$\{r_1{=}1\}$	stw $r_1 \rightarrow [y]$	$\{y{=}1\}$			
Allowed: $0{:}r_1{=}1 \wedge 0{:}r_2 {=}2 \wedge 1{:}r_3 {=}2 \wedge x = 2 \wedge y = 1$					

8. Related Work

Reasonably precise definitions of relaxed memory models were first studied in the Computer Architecture community, e.g. with the early work of Dubois et al. [18] and Collier [17], and an extensive literature has developed since then. We refer the reader to the surveys by Adve and Gharachorloo [8], Luchango [26], and Higham, Kawash, and Verwaal [25] for an overview; the latter relates several axiomatic and abstract-machine (or operational) definitions.

Many of these models are rather idealised with respect to actual processors: we are not aware of any other detailed x86 model, or a model integrated with a substantial instruction semantics. The Itanium and SPARC have vendor specifications in informal mathematics [1, 2] leading to Itanium work by Higham et al. [20] and Yang et al. [31], which builds an oracle from an axiomatic model. Park and Dill produced a specification for SPARC RMO which could be executed on litmus test examples [28]. Adir et al. study the PowerPC [7], and there is early work on a HOL model for Alpha by Gordon [19]. More recently, several authors have considered model-checking of programs above simple weak memory models, e.g. in the work of Burckhardt and colleagues [15, 14], respectively above TSO and above a general relaxed memory model. The latter is a conservative approximation to several models, but does not admit the different views of the x86.

Our x86 memory model is in a similar style to the causal memories of Ahamad et al. [10], and our data-race freedom theorem and proof have a similar structure to theirs. Their causal memories are weaker than the x86 model, lacking a (per memory address) global write serialisation and locked instructions, although they do discuss the possibility of adding additional causality edges to support synchronisation constructs.

Another line of work addresses memory models for high-level programming languages such as Java, X10, and C++ [27, 12, 29, 13]. Here one must consider both the underlying architecture models and compiler-optimisation reorderings, and the lack of clear definitions of the former has led to a need for documents such as Lea's JSR-133 Cookbook for Compiler Writers [24]. Our x86-CC model validates Boehm and Adve's WRC write-to-read causality property [13, Fig.5] (iwp2.5/amd8 is a fence-less x86 analogue of their tests) and their CC property [13, Fig.7] (again for a fence-less analogue). It does not validate their RWC read-to-write causality test [13, Fig.6] without fences. We believe it still would not do so with the MFENCE semantics sketched in §2.12, but would if the fences were replaced by LOCK'd instructions.

There is, of course, also a great deal of work on semantics and architecture description for sequential processor behaviour; space precludes an overview here.

9. Conclusion

Our main contribution is a semantics for multiprocessor x86 programs, with integrated relaxed memory model, instruction semantics, and machine-code decoding. The key difficulty was to go from the informal-prose vendor documentation, with its often-tantalising ambiguity, to a fully rigorous definition (mechanised in HOL) that one can be reasonably confident is an accurate reflection of the vendor architectures (Intel 64 and IA-32, and AMD64). We made particular choices, e.g. in the treatment of events and instructions, the structure of view orders, reads-from maps, etc., the definition of happens-before, and the precise axioms of §2; based on a combination of the prose documentation, discussions with sources, and the testing of §4.

The model provides a necessary foundation for sound reasoning about low-level concurrent x86 code, in many contexts: program logics, algorithm verification, static analysis, compilation, model-checking, proof-carrying code, and so on. It should also provide a solid intuition for low-level programmers, and support the design of high-level language memory models. In §5 and §6 we took the first steps in two directions, with results on the behaviour of race-free programs and a machine characterisation of the memory model, and mentioned some specific items of future work, but the existence of a sound semantics opens up many more opportunities.

The architectures advertised by processor vendors are a key interface, between them and programmers. They must often be loose specifications, to permit processor implementations to change. It appears that the imprecision of informal prose has sometimes been used as a deliberate tool for loose specification, making it extremely hard for low-level programmers to understand the behaviour of their code. We have taken care in our semantics not to over-specify: to the best of our knowledge, the semantics does not commit to anything that vendors should consider unreasonable, and thus it demonstrates that it is feasible to have completely precise, but sufficiently loose, specifications in this area. The semantics also provide a vocabulary for discussing subtle alternatives from the programmer's point of view, without reference to hardware implementation concepts.

Acknowledgements We thank Michael Fetterman, Andy Glew, and Gil Neiger for invaluable discussions about Intel architectures and devices; Nathan Chong and Samin Ishtiaq for discussions about the ARM; and Kathryn Gray, Mike Hicks, Warren Hunt, and Samin Ishtiaq for comments on drafts. We acknowledge the support of a Royal Society University Research Fellowship (Sewell), EPSRC grants GR/T11715, EP/C510712, and EP/F036345, and ANR grant ANR-06-SETI-010-02.

References

[1] A formal specification of Intel Itanium processor family memory ordering. `http://developer.intel.com/design/itanium/downloads/251429.htm`.

[2] The SPARC architecture manual, v. 9. `http://developers.sun.com/solaris/articles/sparcv9.pdf`.

[3] Linux kernel traffic, 1999. `http://www.kernel-traffic.org/kernel-traffic/kt19991220_47.txt`.

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

[5] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, April (vol 1,2A,2B; rev.27), Feb. (vol.3A,3B; rev.26) 2008.

[6] The semantics of multiprocessor machine code, 2008. `www.cl.cam.ac.uk/users/pes20/weakmemory`.

[7] A. Adir, H. Attiya, and G. Shurek. Information-flow models for shared memory with an application to the powerpc architecture. *IEEE Trans. Parallel Distrib. Syst.*, 14(5):502–515, 2003.

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

[9] M. Ahamad, R. A. Bazzi, R.John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proc. SPAA '93*, 1993.

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

[11] ARM. *ARM Architecture Reference Manual (ARMv7-A and ARMv7-R edition)*. 2008. Available from ARM.

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

[13] H.-J. Boehm and S. Adve. Foundations of the C++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, 2008.

[14] S. Burckhardt, R. Alur, and M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, 2007.

[15] S. Burckhardt and M. Musuvathi. Effective program verification for relaxed memory models. In *Proc. CAV, LNCS 5123*, 2008.

[16] N. Chong and S. Ishtiaq. Reasoning about the ARM weakly consistent memory model. In *Proc. MSPC*, 2008.

[17] W. Collier. *Reasoning about parallel architectures*. Prentice-Hall, Inc., 1992.

[18] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *ISCA*, 1986.

[19] M. Gordon. Memory access semantics for a multiprocessor instruction set. Unpublished note (c.1993) `www.cl.cam.ac.uk/ftp/hvg/papers/AlphaProg.ps.gz`.

[20] L. Higham, L. A. Jackson, and J. Kawash. Programmer-centric conditions for itanium memory consistency. In *Proc. ICDCN*, 2006.

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

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

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

[24] D. Lea. The JSR-133 cookbook for compiler writers. `gee.cs.oswego.edu/dl/jmm/cookbook.html`.

[25] L.Higham, J.Kawash, and N. Verwaal. Defining and comparing memory consistency models. In *PDCS*, 1997.

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

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

[28] S. Park and D. L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *Proc. SPAA '95*, 1995.

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

[30] G. Winskel. Event structures. In *Advances in Petri Nets, LNCS 255*, 1986.

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

Addendum, added in press

This paper developed a formal model, x86-CC, which is based on our understanding of the current published Intel and AMD specifications:

- the *Intel 64 Architecture Memory Ordering White Paper* (IWP) [22];
- recent versions of the Intel SDM [5, vol.3A, §7.2.3], both revision 26 (Feb. 2008) and revision 28 (Sept. 2008), which essentially incorporate IWP; and
- the AMD documentation [4, vol.2,p.164ff].

To the best of our knowledge, it does accurately reflect these, and the mathematical and empirical results in the paper hold.

However, whether these specifications (in either formal or informal versions) are useful descriptions of the actual processors, suitable for reasoning about x86 software, is in question.

In one direction, the specifications are arguably too weak (with respect to actual processors). The difficulty of implementing the Java Memory Model above these specifications (recall the weak guarantees provided by MFENCE, as discussed in §2.12,2.13) seems to have motivated a change in the Intel and AMD specifications: we are told that future specifications by both Intel and AMD will exclude the amd6 (IRIW) example shown in Fig. 2. This would bring the model much closer to TSO, but, complicating the issue still further, a draft revised specification seems to admit some non-TSO (indeed, non-coherent) behaviour.

In the other direction, the specifications appear not to include some behaviour that actual processors may exhibit. Consider the following example, due to Paul Loewenstein.

n6	proc:0	proc:1
poi:0	MOV [100]←$1	MOV [200]←$2
poi:1	MOV EAX←[100]	MOV [100]←$2
poi:2	MOV EBX←[200]	
Forbidden: 0:EAX=1 ∧ 0:EBX=0 ∧ [100]=1		

The final state is:

- disallowed by our x86-CC formal model;
- disallowed by any reasonable interpretation, as far as we can tell, of the current Intel and AMD published specifications; but
- according to our preliminary test results, allowed by at least one Intel processor (we find approximately one witness in $2 * 10^6$ executions, reproducibly, on an Intel Core 2, with our `litmus` tool).

It is also allowed by TSO.

The situation is clearly unsatisfactory, and we hope to produce a more useful revised formal model as soon as possible. However, it does, ironically, illustrate the main point of the paper very well: there is a clear need for precise specifications of multiprocessor behaviour, and those specifications must be exercised in some way — one should have little confidence in a loose specification (even if precise) that is not exercised by testing or verification with respect to the hardware, proof of metatheory, and a large body of concurrent programming.

We would like to thank David Christie, Dave Dice, Doug Lea, Paul Loewenstein, and Gil Neiger for their helpful remarks.