The Semantics of x86 Multiprocessor Machine Code Supplementary Examples

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`

October 24, 2008

Abstract

This note contains supplementary details for the paper *The Semantics of x86 Multiprocessor Machine Code* [SSZN+09], with an informal-mathematics presentation of the axiomatic memory model, and illustrating the possible event structures and valid executions of the examples. All the figures were produced with the `memevents` tool, described in the paper.

1 Introduction

The paper *The Semantics of x86 Multiprocessor Machine Code* [SSZN+09] describes our relaxed memory model for the x86 architecture. To help in understanding the memory model, this note gives an informal-mathematics presentation of the model and expands on the example programs given there, as well as a few new ones. This note should be read in conjunction with the paper and accompanying HOL definitions, which include discussion of the choices involved in the definitions, a precise definition of the memory model, a definition of the semantics of some representative instructions, discussion of tools for testing the semantics, and theorems about data-race-free programs and an abstract-machine characterisation of the memory model.

We exhibit various different event structures and allowed view orders in the memory model given in the paper. A particular program may have many executions, with different view orders between its events. This quickly brings on a combinatorial explosion of possibilities. We have developed a tool `memevents` in OCaml to explore these possibilities. We will show pictorially a chosen event structure from the space of possibilities to exhibit the interesting feature of the tests. All the pictures herein are produced by `memevents`.

Many of these examples are based on those in the *Intel 64 Architecture Memory Ordering White Paper* [Int07] (IWP), which states 8 one-sentence principles, illustrated by 10 small litmus-test example programs (which we label iwpNNN), and the AMD documentation [AMD07, vol.2,p.164ff], which gives similar prose and 10 largely identical examples (which we label amdNNN). We also have developed a few of our own tests to exhibit some interesting corner cases.

2 The x86 Axiomatic Memory Model, Informally

In this section we recapitulate briefly the axiomatic memory model [SSZN+09, §2], omitting the accompanying discussion and presenting the model in informal mathematics rather than the higher-order logic formulae of the underlying definition.

2.1 Events

The axiomatic model defines the valid candidate executions. It is in terms of possible orderings over the individual read and write *events*, to memory addresses or processor registers, in a candidate execution. In more detail, an event, such as that shown below

eiid:1 iiid: ⟨ proc:0; po:0 ⟩ R [100] = 0

comprises:

- an *event instance id* eiid (here 1).
- an *instruction instance id*, identifying an instance of an instruction in the candidate execution by giving the processor on which it executed (here proc:0) and the index in program order of the instruction instance on that processor (here po:0, denoting the 0th instruction, in the program with all branches unfolded, on proc:0).
- an *action*, either read (R) or write (W), to either a location which is either a memory address (as the [100] here) or to a register (e.g. EAX or a status flag such as CF). Actions contain the particular value read or written (here R [100] = 0 specifies that value 0 was read).

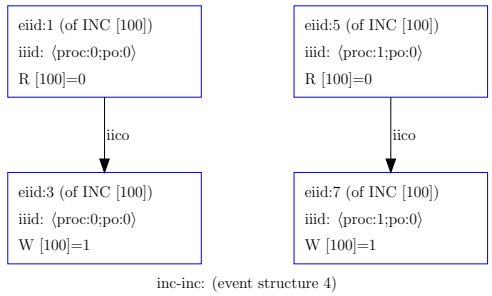
The event instance ids must be unique among all the events with the same instruction instance id, but have no other significance (in particular, they are not ordered in any way).

2.2 Event structures

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 address 100) and the write of an incremented value. For example, given a program with two unlocked INC's

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

we might have four events as below, with a read and a write from each instruction instance.



inc-inc: (event structure 4)

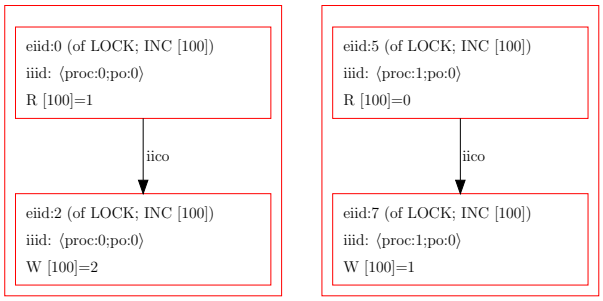
The intra-instruction causality relation is a binary relation over events, shown as the edges labelled iico. Note that this is just one among a large set of possible collections of events for this program: here both reads read value 0, but any other 32-bit values are, a priori, possible. The full semantics also includes events to read and write flags, and the instruction pointer register EIP, but we elide them from most diagrams for clarity.

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:

locked-inc-inc	proc:0	proc:1
po:0	LOCK; INC [100]	LOCK; INC [100]

ensures that, from initial state [100]=0, the only possible final state has [100]=2. We therefore have to record which events belong to the same locked instruction, which we do with an atomicity relation (a

partial equivalence relation) over events, as shown by the outer boxes below.



locked-inc-inc: (event structure 2)

Events from non-locked instructions are not related to anything (even themselves) in this relation.

Collecting this together, we define an *event structure* E to comprise a set of events, an intra-instruction causality relation, and an atomicity relation, subject to various well-formedness conditions which we omit here.

2.3 Execution witnesses

The instruction semantics of [SSZN+09, §3] defines, for any given program, a set of possible event structures. That definition ensures that within each event structure, the events of each individual instruction instance are consistent with the processor behaviour. For example, it specifies what values the status flags have after each arithmetic or logical operation (or if they are explicitly undefined in the architecture).

The rest of this section is devoted to the remaining problem, of defining when the events of multiple instructions are mutually consistent according to the memory model. We do so by defining an *execution witness*, a collection of relations over an event structure, and then axiomatising when an execution witness is valid.

In full, an execution witness X comprises:

- An *initial state constraint*, specifying values contained in (some, but not necessarily all) memory addresses and processor registers and flags.
- A family of *view orders*, one for each processor. A view order for a processor must be a linear order over all of its events and all the memory write events of other processors. If e_1 and e_2 are adjacent in the view order for processor p, we draw an edge $e_1 \xrightarrow{\text{vo:}p} e_2$.
- A *write serialization* relation, which must be the union, over all memory addresses, of family of strict linear orders, each over the memory write events to the relevant address. If e_1 and e_2 are adjacent in the write serialization relation (in which case they must be memory write events to the same address), we draw an edge $e_1 \xrightarrow{\text{P6}} e_2$, after the IWP principle:

  P6. In a multiprocessor system, stores to the same location have a total order.
- A *lock serialization* relation, which must be constructed from a strict linear order of the instances of locked instructions (i.e., a strict linear order of the atomicity equivalence classes of events). If e_1 and e_2 are adjacent in the lock serialization relation (in which case they must be events of two different locked-instruction instances, with the first before the second in the linear order), we draw an edge $e_1 \xrightarrow{\text{P7}} e_2$, after the IWP principle:

  P7. In a multiprocessor system, locked instructions have a total order.
- A *reads-from* relation, which contains a set of pairs $e_w \xrightarrow{\text{rf}} e_r$ where e_w and e_r are a write and a read, respectively, to the same memory address or processor register/flag, and where the value written by e_w is equal to the value read by e_r. For a given read event e_r, if there is no e_w such that $e_w \xrightarrow{\text{rf}} e_r$, we take e_r to have read from the initial state.

2.4 Preserved program order

Consider two events e_1 and e_2 from an event structure, with the same processor, and with e_1 strictly before e_2 in program order. We capture *preserved program order* with the following definitions:

- $e_1 \xrightarrow{\text{reg}} e_2$ if e_1 and e_2 are accesses to the same processor register
- $e_1 \xrightarrow{\text{P1}} e_2$ if e_1 is a memory load and e_2 is a memory load
- $e_1 \xrightarrow{\text{P2}} e_2$ if e_1 is a memory store and e_2 is a memory store
- $e_1 \xrightarrow{\text{P3}} e_2$ if e_1 is a memory load and e_2 is a memory store
- $e_1 \xrightarrow{\text{P4}} e_2$ if e_1 is a memory store and e_2 is a memory load and e_1 and e_2 are to the same address
- $e_1 \xrightarrow{\text{P8}} e_2$ if either e_1 is a memory load or store and e_2 is an event of a locked instruction (i.e. e_2 is in an atomicity set) or vice versa.

modelling the IWP principles P1–4,8:

P1. Loads are not reordered with other loads.

P2. Stores are not reordered with other stores.

P3. Stores are not reordered with older loads.

P4. Loads may be reordered with older stores to different locations but not with older stores to the same location.

P8. Loads and stores are not reordered with locked instructions.

2.5 Causality

Given an event structure E and an execution witness X, we define a *causality*, or *happens-before*, relation over the events to be the transitive closure of the union of:

- the intra-instruction causality relation $\xrightarrow{\text{iico}}$
- the preserved program order relations $\xrightarrow{\text{reg}}$, $\xrightarrow{\text{P1}}$, $\xrightarrow{\text{P2}}$, $\xrightarrow{\text{P3}}$, $\xrightarrow{\text{P4}}$, and $\xrightarrow{\text{P8}}$
- the write serialization relation $\xrightarrow{\text{P6}}$
- the lock serialization relation $\xrightarrow{\text{P7}}$
- the reads-from-map relation $\xrightarrow{\text{rf}}$

2.6 The Axioms

Finally, given an event structure E and an execution witness X, we can define when that execution witness is valid. It has to satisfy the following four conditions, for each processor p:

check_causality: p's view order is consistent with happens-before, i.e., there is no cycle of $\xrightarrow{\text{vo:}p}$ and happens-before edges;

check_rfmap_written: the reads-from map is satisfied by the view orders, i.e., for any $e_w \xrightarrow{\text{rf}} e_r$ where e_r is a read of processor p, there is no other write to the same location that is between e_w and e_r in p's view order;

check_rfmap_initial: the initial-state reads are satisfied by the view orders and initial state, i.e., for any read e_r of processor p, for which there is no e_w such that $e_w \xrightarrow{\text{rf}} e_r$, the initial state has the correct value at the memory address or register of e_r, and there there is no other write to that that is before e_r in p's view order; and

check_atomicity: the atomicity conditions are satisfied by each view order, i.e., for each equivalence class of atomic events (equivalently, for each set of the events of a single locked instruction), there is no event of another instruction that is between any two of those events in p's view order.

The check_causality condition models the IWP principle

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

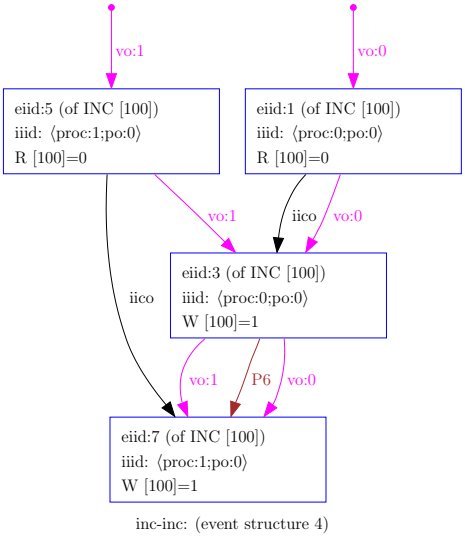
3 Examples

3.1 Two Increments: inc-inc

This test is a short introductory example for two increments on different processors. Each increment instruction reads and writes the same location.

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

There is a possibility for the read and write events from each instruction being interleaved. We exhibit an interleaving below of reads and writes where one increment is lost. This exhibits the fact that we have to work at a finer granularity than the instruction level, in the presence of complex instructions.



In the picture above, we notice the four read and write events labelled with their action and the instruction that gave rise to them. The intra instruction causality is depicted by black arrows labelled `iico`. Each processor `p`'s view order is depicted by magenta arrows labelled `vo:p`. Notice the fact that foreign read events do not appear in a processor's view order. Finally, the conditions P1-P8 imposed by the memory model are depicted by arrows labelled by the condition.

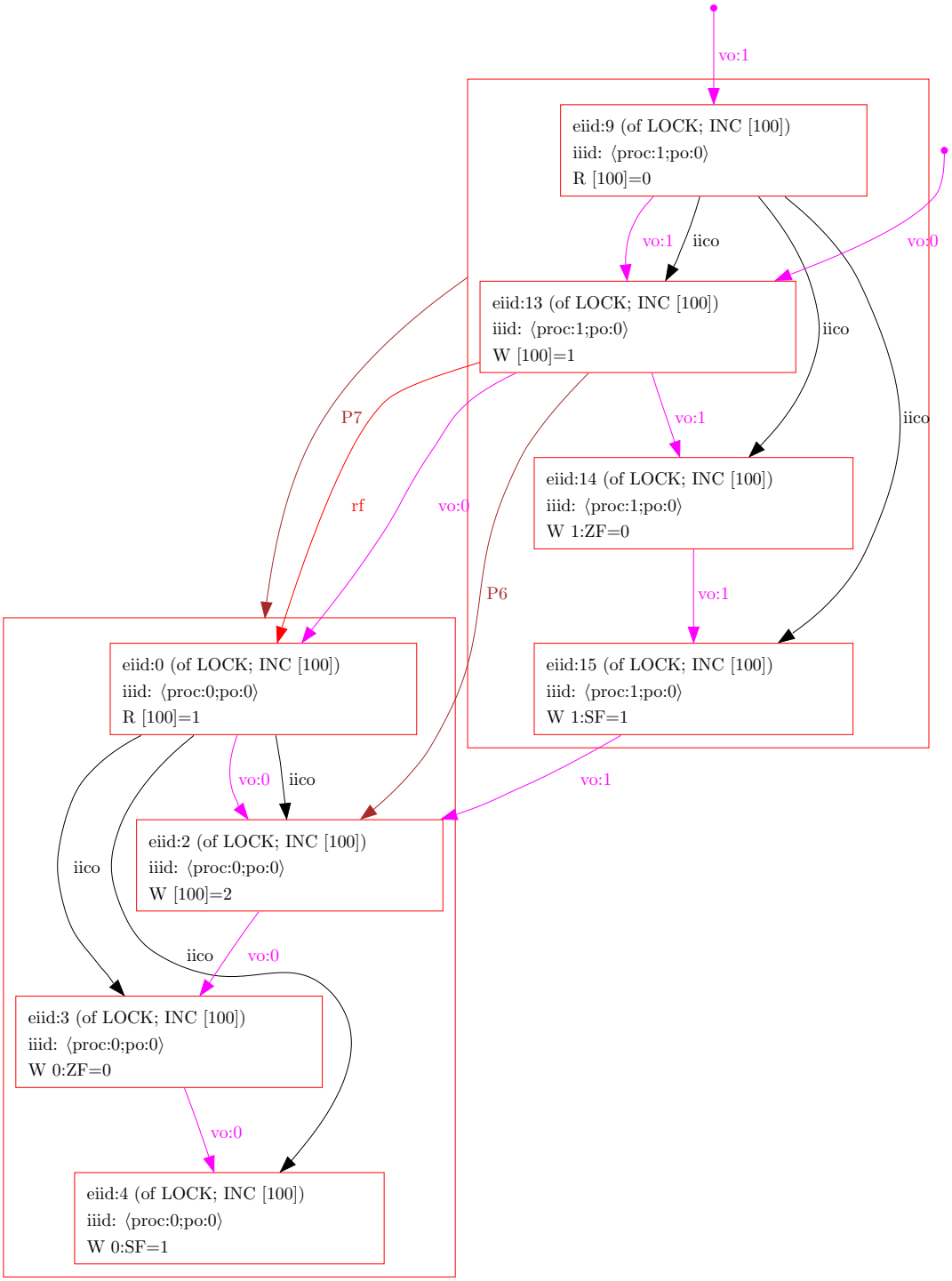
3.2 Two Locked Increments: locked-inc-inc

If instructions are locked, all the events of that instruction must occur atomically in any view order. This ensures that the behaviour of the unlocked increment is not possible, and we are assured a sequentially consistent behaviour.

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

In drawing the pictures, we enclose the events which are supposed to be atomic in a red atomicity box. This comes typically from LOCKed instructions, that is, when the instruction is prefixed by a LOCK, or is an XCHG. In the above diagram we have also displayed the write events to two of the

status flags, SF and ZF.



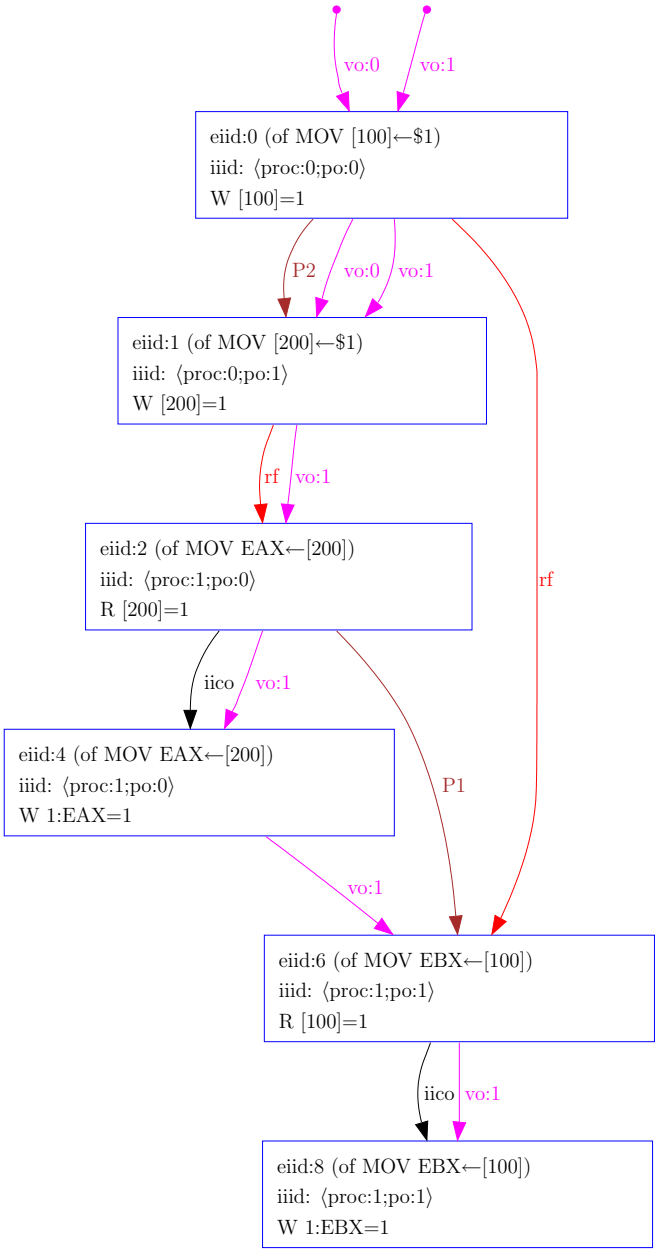
locked-inc-inc: (event structure 2)

Another feature included in the above diagram is a reads-from arrow in red, marked `rf`, which connects a read to the write it reads the value of. If a read does not have such an arrow, as in the unlocked increments in 3.1, it reads values from the initial state. Recall that `check_rfmap` checks that if a read reads-from a write, or if a read reads-from the initial state, there are no intervening writes.

3.3 Loads and Stores reordering: iwp2.1/amd1

This test comes from both the Intel white paper as well as the AMD manual. This test stores to two locations, 100 and 200, on processor 0, and loads from those locations, in the other order, on processor 1. Two loads issued by a processor cannot be reordered, and neither can two stores issued by one processor on this architecture. This ensures that whenever the second store of processor 0 is seen by processor 1, later loads of processor 1 must also see the first store of processor 0.

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)		



iwp2.1/amd1: Litmus Test (event structure 1)

3.4 Stores do not pass preceding loads: iwp2.2/amd2

This test, also from the Intel white paper, is quite similar to the test 3.3. On processor 0, the write of location 200 (eiid:4) cannot be reordered before the read of location 100(eiid:0), and on processor 1, the write of location 100 (eiid:9) cannot be reordered before the read of location 200 (eiid:6). This ensures that one or the other load can see the store on the other processor occurring before, but not both.

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		

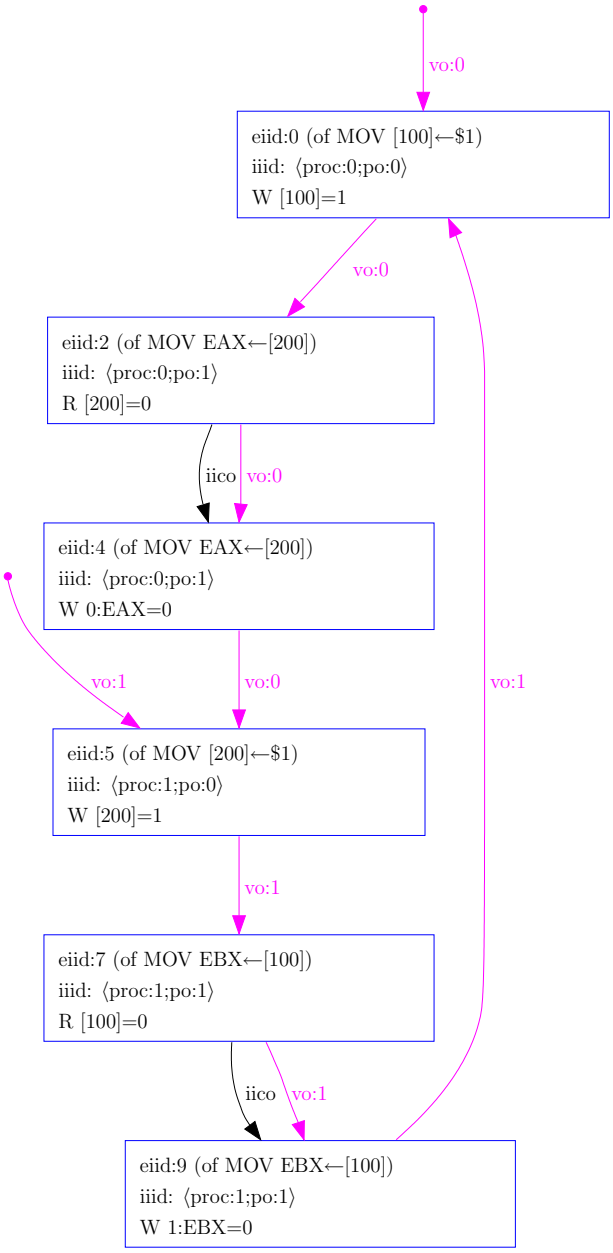
3.5 Loads can pass preceding stores: iwp2.3.a/amd4

This test comes from both the Intel white paper and AMD. If, unlike test 3.4, the writes on each processor are before the loads in program order, the architecture is allowed to reorder them.

iwp2.3.a/amd4	proc:0	proc:1
po:0	MOV [100]←$1	MOV [200]←$1
po:1	MOV EAX←[200]	MOV EBX←[100]
Allowed: 0:EAX=0 ∧ 1:EBX=0		

Interestingly, the test does not need to speculate reads to show this behaviour. Below we show an iwp2.3.a/amd4 execution without reordering, in which each processor sees its own events in program

order, but delays seeing the write from the other processor.



iwp2.3.a/amd4: Litmus Test (event structure 4)

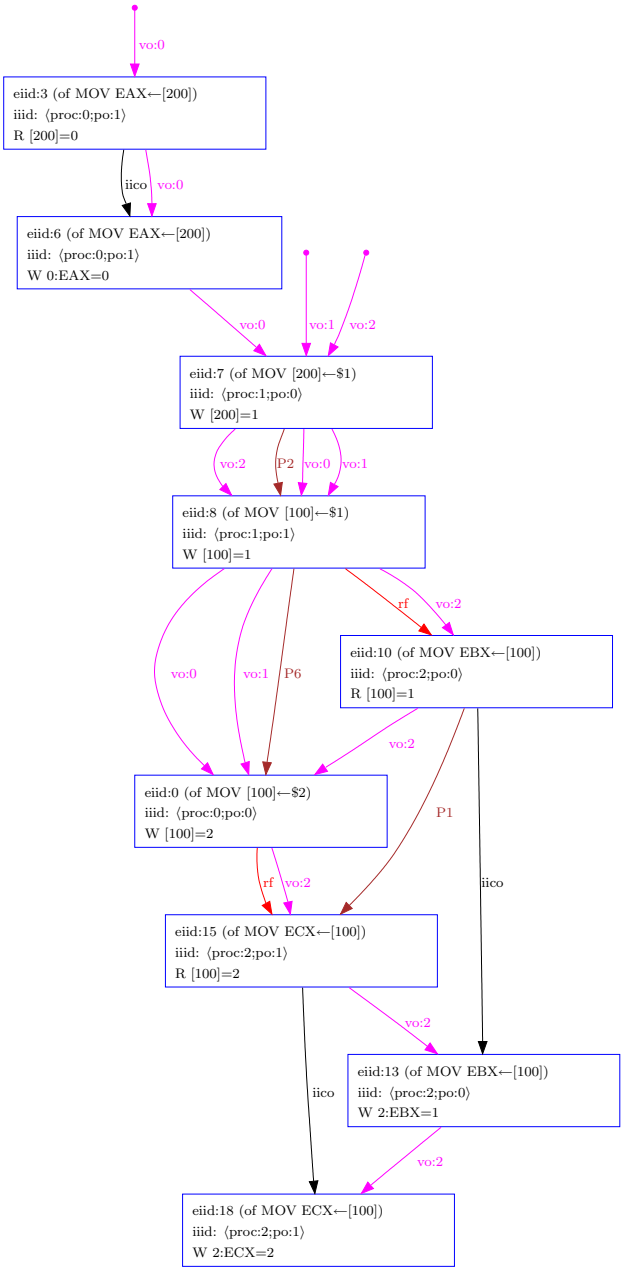
3.6 Read speculation observed: n1

Sometimes however, read speculation is observable. Consider the test below, which is new:

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

In this case, the read of location 200 on processor 0 must be speculated before the write of location 100 in order not to see the write from processor 1. The reads on processor 2 indicate a particular write

serialisation order, which on x86 must be respected in all view orders.

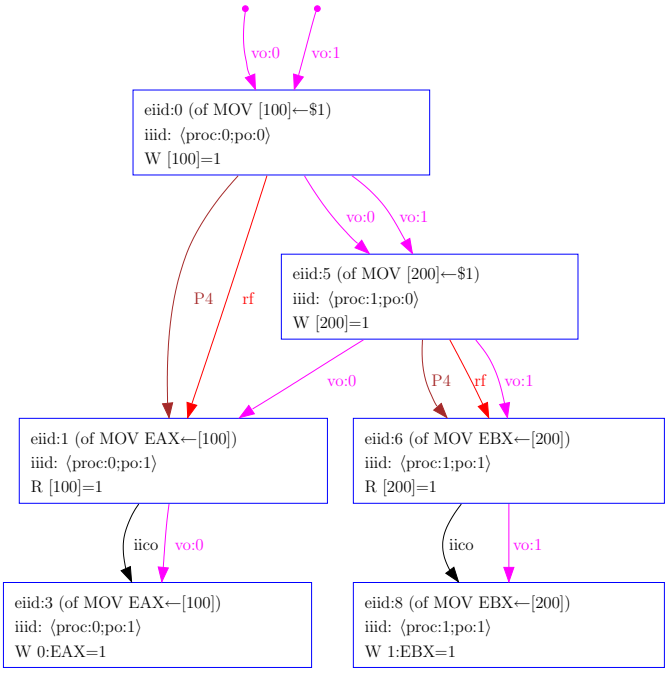


n1: Susmit's Read Speculation Test (event structure 22)

3.7 Loads do not pass preceding stores to same address: iwp2.3.b

There is an important exception to permissible reorderings of loads and preceding stores. If they are both to the same address, as seen on two processors in this test from the Intel white paper (the April 2008 version of the Software Developer's Manual has the same test, but only on one processor), then they have to be seen in program order.

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		



iwp2.3.b: Litmus Test (event structure 1)

3.8 Loads do not pass locked instructions: iwp2.8a

This test and the next one, both from the Intel white paper, show that loads and stores are not reordered past locked instructions. We believe this condition is redundant for the current instruction set, since locked instructions all include a memory read and a memory write. Interpreting loads and stores to include all loads and stores, including locked ones, gives us already the program ordering constraints from conditions P1–P4.

We suspect that the LOCK prefix can in fact also be applied to other instructions in current or future generations of the processors, and P8 is included so that the memory model is correctly defined in those cases. It's also conceivable that P8 is redundant; or that we should reinterpret P1–P4 as referring only to unLOCK'd loads and stores.

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		

3.9 Stores do not pass locked instructions: iwp2.8b

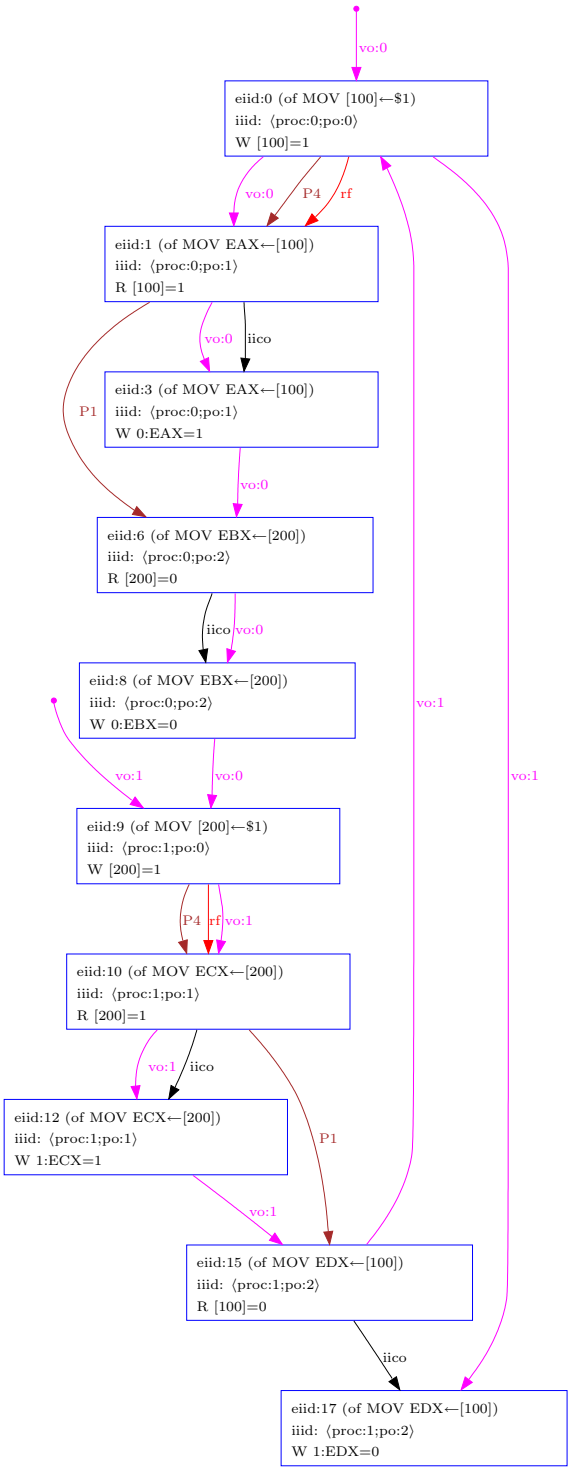
iwp2.8.b	proc:0	proc:1
po:0	XCHG [100]←EAX	MOV EBX←[200]
po:1	MOV [200]←$1	MOV ECX←[100]
Initial state: 0:EAX= 1 (elsewhere 0)		
Forbidden: 1:EBX=1 ∧ 1:ECX=0		

3.10 Store buffering observable: iwp2.4/amd9

This test, which comes from both the Intel white paper and the AMD documentation, can be seen as a combination of tests 3.5 and 3.7. It shows that each processor can see its own writes while delaying seeing the other processor's writes. This points to the existence of store buffers local to the processor,

which can satisfy its own reads without necessarily propagating to global memory. Notice the interesting fact that each processor is forced to see its own events in program order by the preserved program order conditions P1–P4.

iwp2.4/amd9	proc:0	proc:1
po:0	MOV [100]←$1	MOV [200]←$1
po:1	MOV EAX←[100]	MOV ECX←[200]
po:2	MOV EBX←[200]	MOV EDX←[100]
Allowed: 0:EBX=0 ∧ 1:EDX=0		



iwp2.4/amd9: Litmus Test (event structure 6)

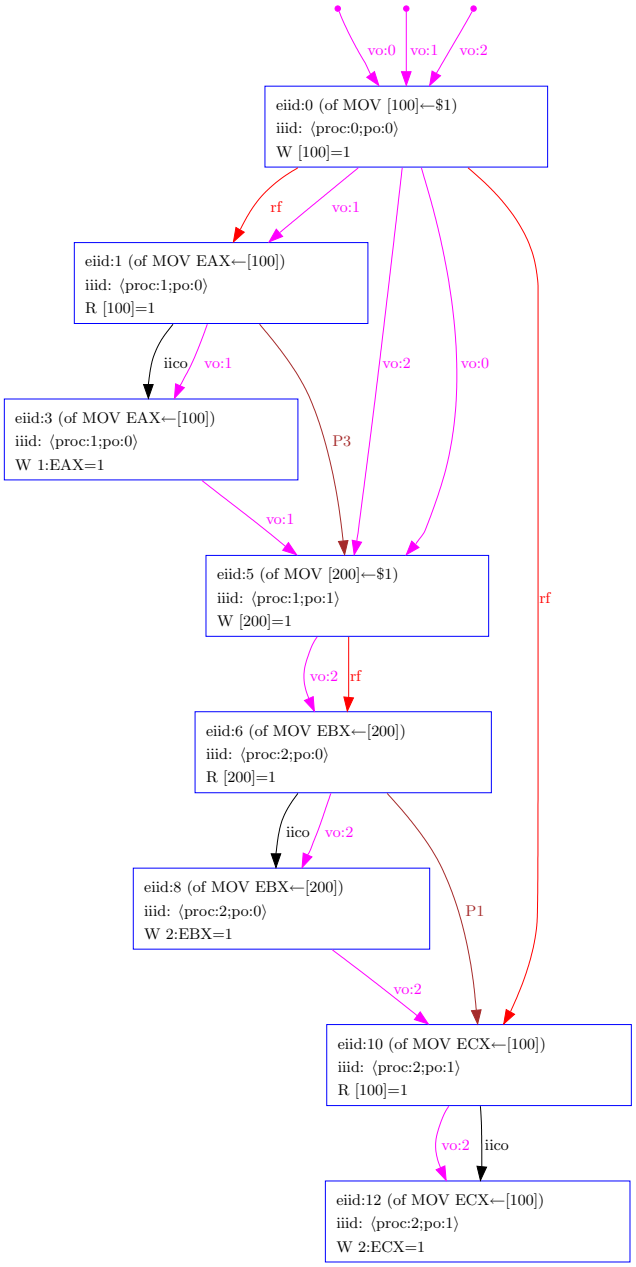
3.11 Transitive causality: iwp2.5/amd8

This test shows that there is transitive causality from a combination of a reads-from relationship (that is, a read must come after a write it reads the value of) and a preserved-program-order relationship.

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

Consider the read of location 100 on processor 1. Since this reads the value written by processor 0, it must come after that write. Also, since subsequent writes have a preserved program order, the write of location 200 is constrained to be after the read in view order of processor 1. Now consider the read of location 200 on processor 2. Since this reads the value written by processor 1, it must come after the write in the view order of processor 2. The architecture guarantees that all these conditions are

transitively imposed, and hence the second read on processor 2 must read 1 as well.



iwp2.5/amd8: Litmus Test (event structure 1)

3.12 Writes to a location are serialised: iwp2.6

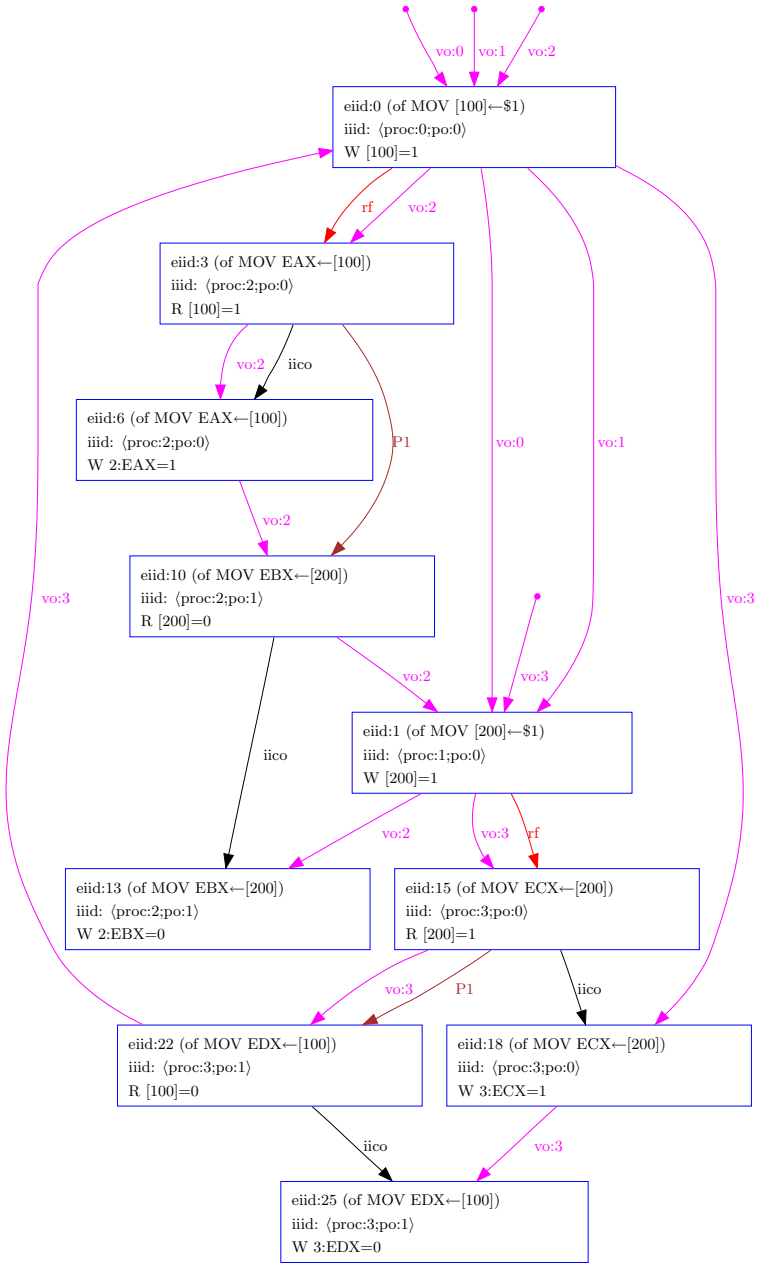
This test, from the Intel white paper, shows that all observers must agree on the order of writes to the same location. If processor 2 sees the write of processor 0 before the write of processor 1, processor 3 is not allowed to see them in a different order.

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				

3.13 No serialisation of independent writes: amd6

This test, from the AMD documentation, varies test 3.12 by making the writes on processors 0 and 1 be to different locations. Now it becomes possible to observe the writes in different orders on processors 2 and 3.

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				



amd6: Litmus Test (event structure 51)

3.14 Locked instructions are serialised: iwp2.7/amd7

This test, from both the Intel white paper and the AAMD documentation, shows that there is a globally imposed serialisation of all locked instructions. Processors 2 and 3 are not allowed to observe the effects of the two XCHG instructions (locked implicitly) from processors 0 and 1 in different orders.

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				

3.15 Transitivity through write serialisation: n2

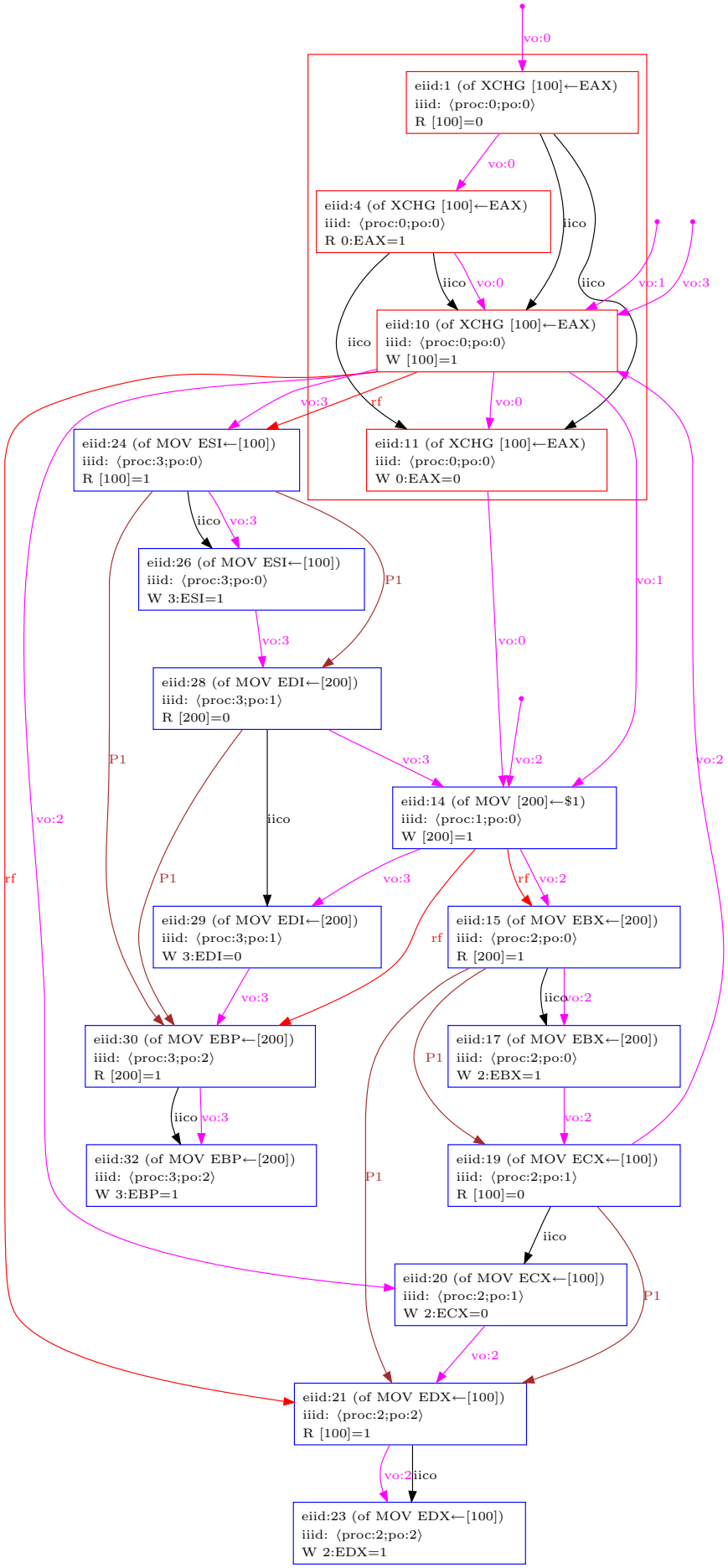
We might wonder how strong the transitivity enforced by the architecture actually is. Consider the new test below. Here processor 0 performs two independent writes, and processor 1 does two independent writes, such that the first write is to the same location as the second write of proocessor 0, and the second write is to a fresh location. Absent any observations on processor 2, the two reads on processor 3 are allowed to see the writes in any order whatsoever. Now consider processor 2 seeing the two writes to the common location in a particular order. Since writes to a location is serialised, any other processor must observe these writes in that order. The question is whether transitivity goes through this write serialisation on a processor (such as processor 3) that does not ever perform the read to the location serialised on. Discussions with Intel architects indicates that this is still enforced, and hence we include write serialisation edges when we calculate the transitively closed `happens-before` relation.

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				

3.16 No order between locked writes and writes: n3

While two locked instructions are globally serialised by a lock serialisation order, and writes to the same location are globally serialised by a write serialisation order, there is no constraint between an unlocked write and a locked write to independent locations. We illustrate this fact by our new test below, where processor 2 sees the unlocked write of processor 1 before the locked write of processor 0, while processor 3 sees them in exactly the reverse order.

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				



n3: Litmus Test (event structure 3)

3.17 Fences: amd5

An example from the AMD documentation, this is a version of test 3.5 with the addition of MFENCEs on both processors. Again, the previously visible behaviour is now forbidden. This can be explained by a weak version of MFENCE which imposes local ordering between the store and the subsequent load on both processors.

amd5	proc:0	proc:1
po:0	MOV [100]←$1	MOV [200]←$1
po:1	MFENCE	MFENCE
po:2	MOV EAX←[200]	MOV EBX←[100]
Forbidden: 0:EAX=0 ∧ 1:EBX=0		

3.18 Fences: amd10

The final test, also from the AMD documentation, inserts a MFENCE instruction on each processor in test 3.10. On the AMD, this prevents the previously allowed behaviour of each processor seeing its own write but not the other processor's write to be forbidden. Notice that this result implies MFENCEs are stronger than imposing ordering between events of the same processor, since test 3.10 already has ordering locally.

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		

References

[AMD07] *AMD64 Architecture Programmer's Manual*. Advanced Micro Devices, September 2007. (3 vols).

[Int07] Intel. Intel 64 architecture memory ordering white paper, 2007. SKU 318147-001.

[SSZN+09] S. Sarkar, P. Sewell, F. Zappa Nardelli, S. Owens, T. Ridge, T. Braibant, M. Myreen, and J. Alglave. The semantics of x86 multiprocessor machine code. In *Proc. POPL 2009*, January 2009. To appear.