Multiprocessor Architectures Don't Really Exist (But They Should)

Peter Sewell Susmit Sarkar

University of Cambridge

with:

Mark Batty, Anthony Fox, Magnus Myreen, Scott Owens, Tom Ridge
University of Cambridge

Jade Alglave, Luc Maranget, Francesco Zappa Nardelli
INRIA

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

MTV 7 December 2009

What is a processor architecture anyway?

- interface between h/w and s/w people
- assumption for software verification
- criterion for verification of processor design

Lots of them

Intel 64 and IA-32 Architectures Software Developer's Manual

AMD64 Architecture Programmer's Manual

Power ISA specification

ARM Architecture Reference Manual

. . .


intel
Intel® 64 and IA-32 Architectures
Software Developer's Manual
VOLUME 3A: System Programming Guide
Part 1

Subject to Conflicting Requirements

They must:

1. reveal enough processor behaviour for effective programming;
2. not reveal sensitive IP;
3. be sound with respect to a range of previous specific processor implementations; and
4. not unduly constrain future processor design;
5. be accessible to programmers and hardware designers (not overwhelmingly complex).

. . . so have to be *loose* specifications

The IA-32 AAA instruction (ASCII Adjust After Addition):

> *The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0. The OF, SF, ZF, and PF flags are undefined.*

. . . so have to be *loose* specifications

The IA-32 AAA instruction (ASCII Adjust After Addition):

> *The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0. The OF, SF, ZF, and PF flags are undefined.*

Can lead to problems:

- s/w tested above actual processors, not liberal architecture simulators

. . . so have to be *loose* specifications

The IA-32 AAA instruction (ASCII Adjust After Addition):

> *The AF and CF flags are set to 1 if the adjustment results in a decimal carry; otherwise they are set to 0. The OF, SF, ZF, and PF flags are undefined.*

Can lead to problems:

- s/w tested above actual processors, not liberal architecture simulators

But

- clear what programmer can depend on
- h/w testing can use (almost) deterministic golden reference model

But what about Multiprocessors?

Traditional naive assumption:

> multiprocessors are *sequentially consistent*: accesses by multiple threads to a shared memory occur in a global-time linear order.

But what about Multiprocessors?

Traditional naive assumption:

> multiprocessors are *sequentially consistent*:
> accesses by multiple threads to a shared memory
> occur in a global-time linear order.

Obviously False!

Multiprocessors are built of optimisations:
local store buffers, shadow register files, cache hierarchies,...

- unobservable by single-threaded programs;
- sometimes observable by concurrent code.

Only a *relaxed* (or *weakly consistent*) view of the memory.
(compilers too)

A Simple Example

Initial: [x]=0 ∧ [y]=0	
proc 0	proc 1
MOV [x]←1 MOV EAX←[y] (0)	MOV [y]←1 MOV EBX←[x] (0)
Allow: EAX=0 ∧ EBX=0	

- One can't view the execution in global time
  (at this level of abstraction)
- Observable on dual core Intel Core2 (630 / 100,000)
- Plausible microarchitectural explanation
- If that's allowed, then what *else* might be?

In practice

Architectures described by *informal prose*:

> *In a multiprocessor system, maintenance of cache consistency may, in rare circumstances, require intervention by system software.*
>
> (Intel SDM, Nov. 2006, vol 3a, 10-5)

In practice

Architectures described by *informal prose*:

> *In a multiprocessor system, maintenance of cache consistency may, in rare circumstances, require intervention by system software.*

(Intel SDM, Nov. 2006, vol 3a, 10-5)

As we shall see, such descriptions sometimes are:

1) vague; 2) incomplete; 3) unsound.

Also, they cannot be used to *test programs* or to *test processor implementations*.

In practice

Architectures described by *informal prose*:

> *In a multiprocessor system, maintenance of cache consistency may, in rare circumstances, require intervention by system software.*
>
> (Intel SDM, Nov. 2006, vol 3a, 10-5)

As we shall see, such descriptions sometimes are:

1) vague; 2) incomplete; 3) unsound.

Also, they cannot be used to *test programs* or to *test processor implementations*.

In this sense, multiprocessor architectures **don't really exist**.

Our Plan

We've been looking at the memory models of x86, Power, ARM, and C++ (and Ševčík and Aspinall looked at Java).

Our Plan

We've been looking at the memory models of x86, Power, ARM, and C++ (and Ševčík and Aspinall looked at Java).

The vendor specs and language standards are **all** flawed

Our Plan

We've been looking at the memory models of x86, Power, ARM, and C++ (and Ševčík and Aspinall looked at Java).

The vendor specs and language standards are **all** flawed

We'll show some problems and talk about specific and general solutions

Era of Vagueness (before Aug. 2007, e.g. Intel SDM rev. 22)

- Linux kernel mailing list (Nov. 1999):

  "Can we remove a barrier from our spinlock implementation?"

- Simple programming question, highly conjectural debate
- Resolved only by appeal to an oracle

Era of Causality: Ambiguity

IWP/SDM rev. 26/AMD 3.14/x86-CC (Aug 2007 – Oct. 2008)

- 10 *litmus tests*, e.g.:

proc:0	proc:1	proc:2
MOV [x]←1	MOV EAX←[x] MOV [y]←1	MOV EBX←[y] MOV ECX←[x]
Forbid: 1:EAX=1 ∧ 2:EBX=1 ∧ 2:ECX=0		

- 8 IWP *'principles'*, e.g.:

  "Intel 64 memory ordering ensures transitive visibility of stores — i.e. stores that are causally related appear to execute in an order consistent with the causal relation"

Era of Causality: Weakness

Independent reads of independent writes

Initial: [x]=0 ∧ [y]=0			
proc 0	proc 1	proc 2	proc 3
MOV [x]←1	MOV [y]←1	MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV ECX←[y] (1) MOV EDX←[x] (0)
Final: 2:EAX=1 ∧ 2:EBX=0 ∧ 3:ECX=1 ∧ 3:EDX=0			

- proc 2: see write x before write y
- proc 3: see write y before write x
- AMD manual (3.14): yes!
- Intel manual: ? (implicitly allowed by principles)
- real hardware: unobserved

Weakness: adding MFENCEs does not recover SC (which was assumed in a Sun implementation of the JMM)

Era of Causality: Unsoundness

Initial: [x]=0 ∧ [y]=0	
proc 0	proc 1
MOV [x]←1 MOV EAX←[x] (1) MOV EBX←[y] (0)	MOV [y]←2 MOV [x]←2
Final: 0:EAX=1 ∧ 0:EBX=0 ∧ [x]=1	

(Thanks to Paul Loewenstein)

Observed on hardware, but *not* allowed by IWP principles:

"Stores are not reordered with other stores"

and

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

Second Era of Causality

Intel SDM rev. 29–32 (Nov. 2008 – Nov. 2009)

- Not unsound in the previous sense
- Not weak in the IRIW sense

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

- But... still a bit ambiguous, and *the view by those processors is left entirely unspecified!*

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

AMD 3.15 (Nov. 2009): IRIW explicitly forbidden

Modern Age (x86-TSO)

- Folk wisdom: x86 has a relatively strong architecture.
- Litmus tests consistent with total store ordering (TSO)
- Suggestive rev.29–32
- Suggestive comments from Intel and AMD staff

Simple Plan: x86-TSO

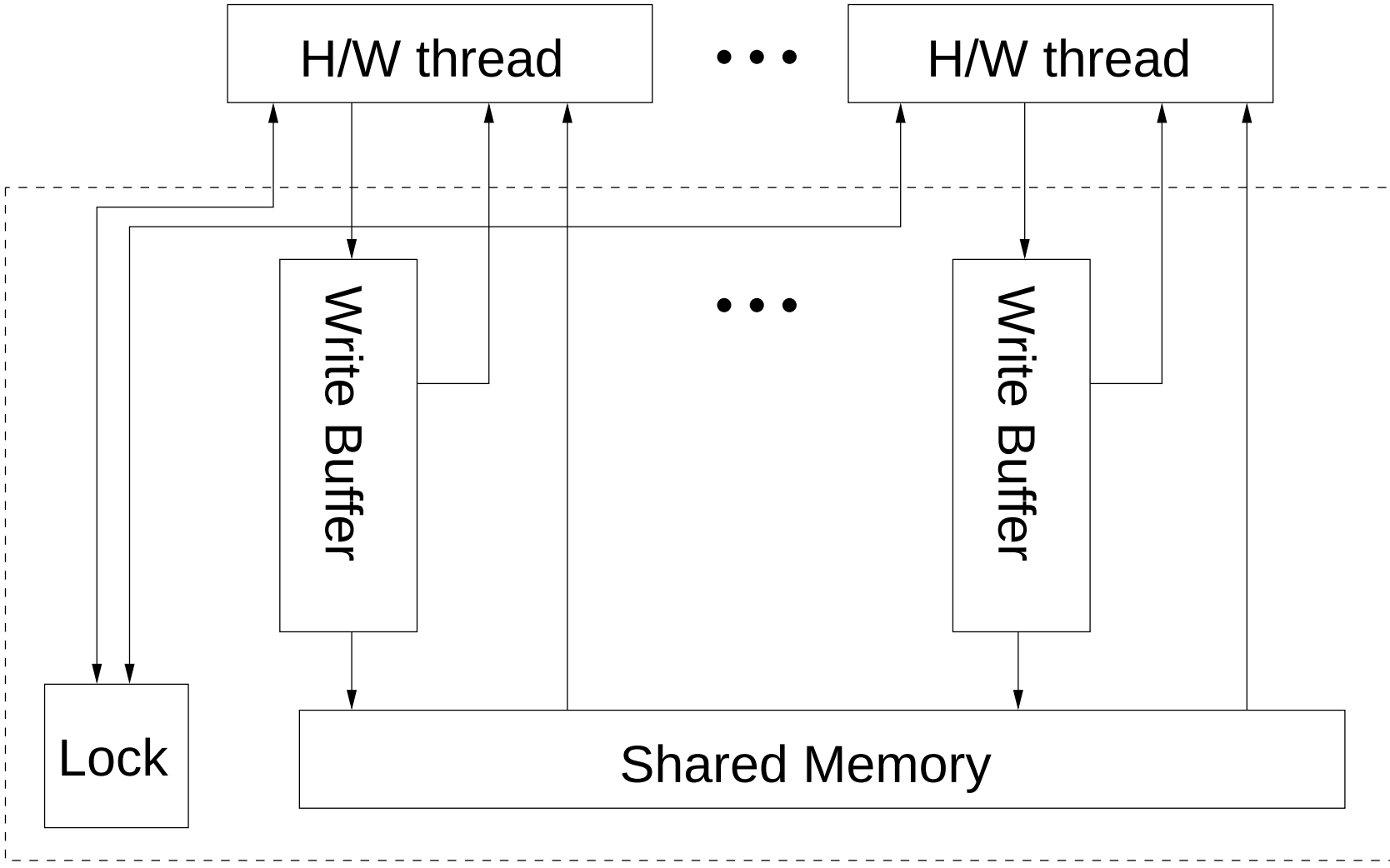
Specify a TSO-based programmer-visible architecture
(adapting to x86 as appropriate)
(and considering only common-case aligned write-back accesses)

Integrate memory model with instruction semantics spec.
(specify and test 30-odd instructions, with all addressing modes, including instruction decoding)

Define both abstract machine and axiomatic versions of MM

In HOL (mechanized logic — expressive and precise)

Abstract Machine



Abstract Machine

A tool to specify the *programmer-visible behaviour* only.

The internal structure should be easy to understand, but need not be much like the actual h/w.

Force of the model: programmers can assume that nothing (of the internal optimisations of processors) *except* FIFO write buffers is *visible to them*

Tools

- MEMEVENTS, exhaustively finding *all results* allowed by the model for a litmus test program
- LITMUS, running litmus tests on real h/w
- X86SEM, testing instruction semantics against real h/w
- ARCHITECTURAL EMULATOR, find, randomly, *some result* allowed by the model for a larger program. WIP.

Power ISA 2.06 and ARM v7

Visible behaviour is much weaker & more subtle than x86

Key concept: actions being *performed* or *observed*.

> A *load* by a processor (P1) *is performed* with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.

Used to define the semantics of dependencies and barriers.

This style of definition goes back to the work of Dubois et al. (1986).

Power ISA 2.06 and ARM v7

Visible behaviour is much weaker & more subtle than x86

Key concept: actions being *performed* or *observed*.

> A *load* by a processor (P1) *is performed* with respect to any processor (P2) when the value to be returned by the load can no longer be changed by a store by P2.

Used to define the semantics of dependencies and barriers.

This style of definition goes back to the work of Dubois et al. (1986).

But it's *subjunctive*: it refers to a hypothetical store by P2.

Underlying Problems

These specs are written by smart people, but:

- confusion between *loose specs* and *vague specs*
- informal prose is a *terrible* medium for writing precise but loose specs

Conclusion

If we're

- to have any hope of building reliable multiprocessor software
- or of testing processor designs against an architecture

we need to replace loose-specification-by-ambiguous prose by loose specification in a rigorous language.

It's not that hard.

A precise model makes possible:

- correctness proofs of architecture
- testing program behaviour

The End

Java

Goal 1: data-race free programs are sequentially consistent;

Goal 2: all programs satisfy memory safety/security properties;

Goal 3: common compiler optimisations are sound.

Java

Goal 1: data-race free programs are sequentially consistent;

Goal 2: all programs satisfy memory safety/security properties;

Goal 3: common compiler optimisations are not all sound
[Ševčík and Aspinall]

Java

Goal 1: data-race free programs are sequentially consistent;

Goal 2: all programs satisfy memory safety/security properties;

Goal 3: common compiler optimisations are not all sound
[Ševčík and Aspinall]

C++0x

Goals 1 and 3 only

Prose draft standard. Formalisation in progress [Mark Batty]

Low-level atomics driven by relaxed-memory processors