How do multicore machines actually behave?

(x86, ARM/POWER, Java, and C/C++11)

Peter Sewell

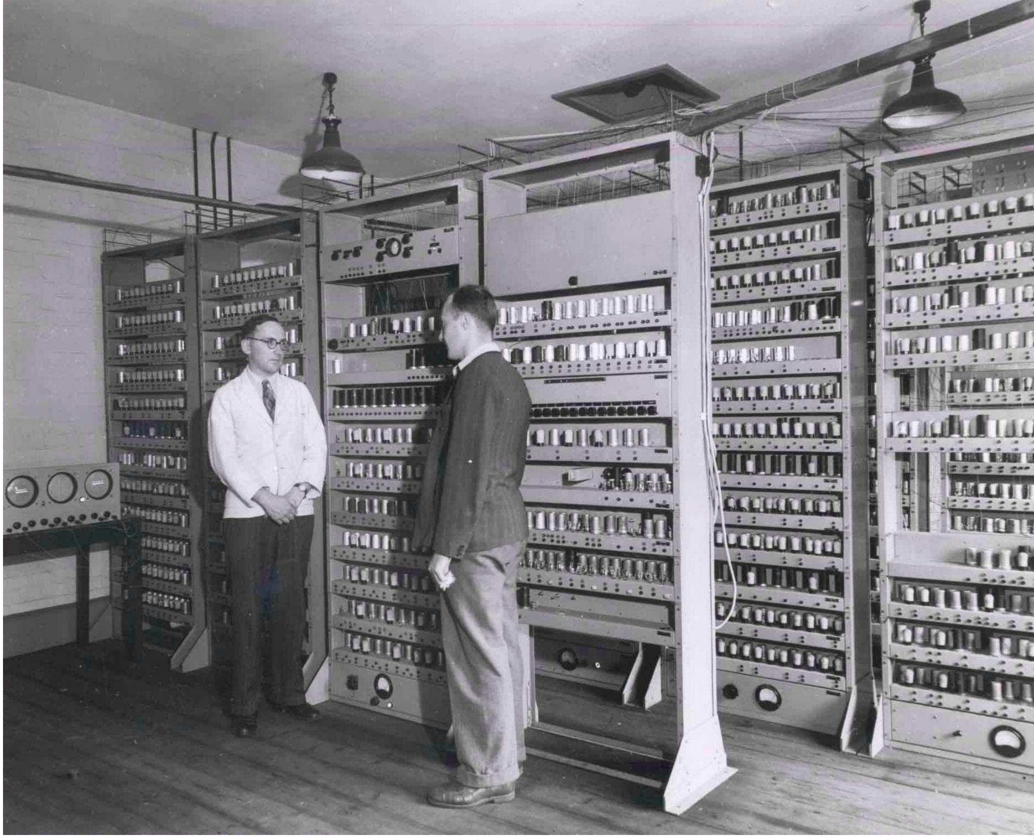
University of Cambridge

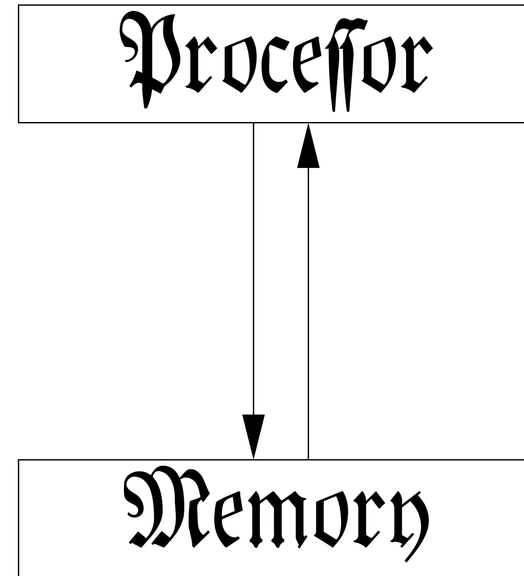
TRANSFORM Summer School, MSR Cambridge, July 2012

Who Needs to Know?

1. processor designers
2. concurrency library authors
3. compiler writers
4. programming language designers
5. verification tool builders
6. semanticists
7. mainstream programmers?
8. you?

The Golden Age, 1945–1959





Programs

Memory locations x, y,... hold *values* (numbers 0 – 255)

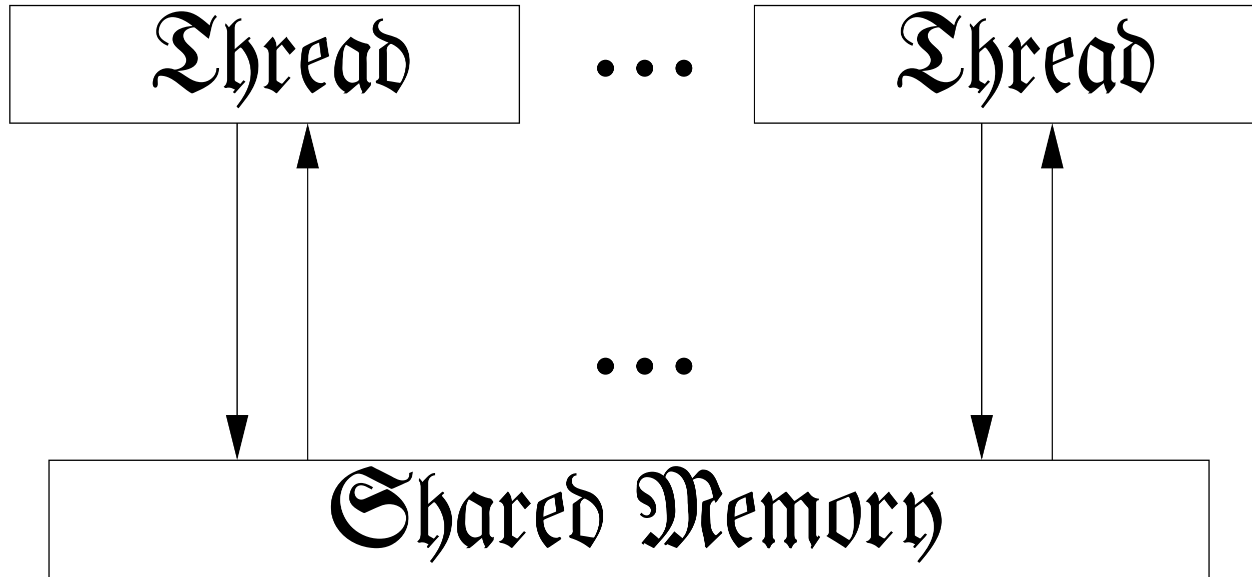
Programs are lists of simple instructions:

```
start:  x = 17
        y = 1
label:  y = 2 × y
        x = x - 1
        if x > 0 goto label
        print y
```

...that are executed in order and that sometimes read (and sometimes change) the values held in memory

...any read reads the most recent value written

Multiprocessors



Multiple hardware threads operating on the *same* memory

The Ghost of Multiprocessors Past

BURROUGHS D825, 1962

''Outstanding features include truly modular hardware with parallel processing throughout''

FUTURE PLANS

The complement of compiling languages is to be expanded.

The Ghost of Multiprocessors Present

ARM®

MOBILE PHONES NEWS

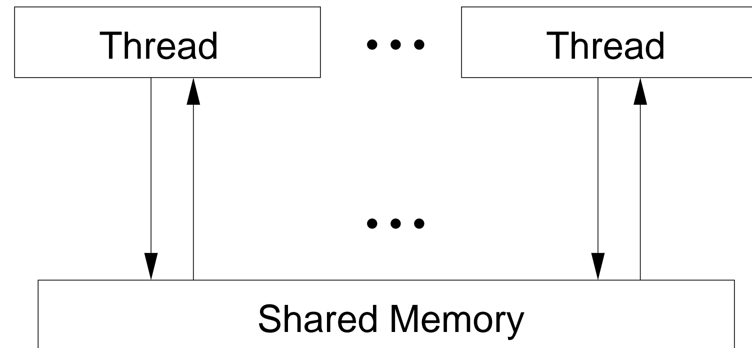
Best quad core phone: 4 contenders examined

Early View: HTC One X vs ZTE Era vs LG Optimus 4X HD vs Huawei Ascend D Quad

Intel Xeon E7
(up to 20 hardware threads)

IBM Power 795 server
(up to 1024 hardware threads)

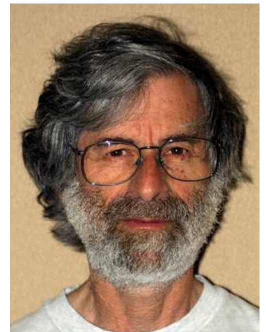
Multiprocessors — with SC Shared Memory?



Multiple threads, but acting on a *sequentially consistent* (SC) shared memory:

> *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, respecting the order specified by the program*

Leslie Lamport, 1979



A Simple Hardware Example (SB)

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y, initially 0

Thread 0		Thread 1	
MOV [x]$\leftarrow$1	(write x=1)	MOV [y]$\leftarrow$1	(write y=1)
MOV EAX$\leftarrow$[y]	(read y)	MOV EBX$\leftarrow$[x]	(read x)

What final states are allowed?

What are the possible sequential orders?

A Simple Hardware Example (SB)

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y, initially 0

Thread 0	Thread 1
MOV [x]←1 (write x=1)	
MOV EAX←[y] (read y=0)	
	MOV [y]←1 (write y=1)
	MOV EBX←[x] (read x=1)

Thread 0:EAX = 0 Thread 1:EBX=1

A Simple Hardware Example (SB)

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y, initially 0

Thread 0	Thread 1
MOV [x]$\leftarrow$1 (write x=1)	
	MOV [y]$\leftarrow$1 (write y=1)
MOV EAX$\leftarrow$[y] (read y=1)	
	MOV EBX$\leftarrow$[x] (read x=1)

Thread 0:EAX = 1 Thread 1:EBX=1

A Simple Hardware Example (SB)

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y, initially 0

Thread 0	Thread 1
MOV [x]$\leftarrow$1 (write x=1)	
	MOV [y]$\leftarrow$1 (write y=1)
	MOV EBX$\leftarrow$[x] (read x=1)
MOV EAX$\leftarrow$[y] (read y=1)	

Thread 0:EAX $= 1$ Thread 1:EBX$=1$

A Simple Hardware Example (SB)

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y, initially 0

Thread 0	Thread 1
	MOV [y]$\leftarrow$1 (write y=1)
MOV [x]$\leftarrow$1 (write x=1)	
MOV EAX$\leftarrow$[y] (read y=1)	
	MOV EBX$\leftarrow$[x] (read x=1)

Thread 0:EAX $= 1$ Thread 1:EBX$=1$

A Simple Hardware Example (SB)

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y, initially 0

Thread 0	Thread 1
	MOV [y]←1 (write y=1)
MOV [x]←1 (write x=1)	
	MOV EBX←[x] (read x=1)
MOV EAX←[y] (read y=1)	

Thread 0:EAX = 1   Thread 1:EBX=1

A Simple Hardware Example (SB)

At the heart of a mutual exclusion algorithm, e.g. Dekker's, you might find code like this, say on an x86.

Two memory locations x and y, initially 0

Thread 0	Thread 1
	MOV [y]←1 (write y=1)
	MOV EBX←[x] (read x=0)
MOV [x]←1 (write x=1)	
MOV EAX←[y] (read y=1)	

Thread 0:EAX = 1 Thread 1:EBX=0

A Simple Hardware Example (SB)

Conclusion:

0,1 and 1,1 and 1,0 can happen, but 0,0 is impossible

A Simple Hardware Example (SB)

Conclusion:

0,1 and 1,1 and 1,0 can happen, but 0,0 is impossible

In fact, in the real world:

we observe 0,0 every 630/100000 runs
(on an Intel Core Duo x86)

(and so Dekker's algorithm will fail)

A Simple Compiler Optimisation Example (MP)

Thread 1	Thread 2
`data = 1`	
`ready = 1`	`while (ready != 1) {};`
	`print data`

A Simple Compiler Optimisation Example (MP)

In SC, message passing should work as expected:

Thread 1	Thread 2
`data = 1`	
`ready = 1`	`if (ready == 1)`
	`print data`

In SC, the program should only print 1.

A Simple Compiler Optimisation Example (MP)

Thread 1	Thread 2
`data = 1`	`int r1 = data`
`ready = 1`	`if (ready == 1)`
	`   print data`

In SC, the program should only print 1.

Regardless of other reads.

A Simple Compiler Optimisation Example (MP)

Thread 1	Thread 2
`data = 1`	`int r1 = data`
`ready = 1`	`if (ready == 1)`
	`print data`

In SC, the program should only print 1.

But common subexpression elimination (as in `gcc -O1` and `HotSpot`) will rewrite

`print data` $\Longrightarrow$ `print r1`

A Simple Compiler Optimisation Example (MP)

Thread 1	Thread 2

`data = 1`
`ready = 1`
	`int r1 = data`
	`if (ready == 1)`
	`   print r1`

In SC, the program should only print 1.

But common subexpression elimination (as in `gcc -O1` and `HotSpot`) will rewrite

`print data` $\Longrightarrow$ `print r1`

So the compiled program can print 0

Relaxed Memory

Multiprocessors and compilers incorporate many performance optimisations

(hierarchies of cache, load and store buffers, speculative execution, cache protocols, common subexpression elimination, etc., etc.)

These are:

- unobservable by single-threaded code
- sometimes observable by concurrent code

Upshot: they provide only various *relaxed* (or *weakly consistent*) memory models, not sequentially consistent memory.

What About the Specs?

Hardware manufacturers document *architectures*:

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

AMD64 Architecture Programmer's Manual

Power ISA specification

ARM Architecture Reference Manual

and programming languages (at best) are defined by *standards*:

ISO/IEC 9899:1999 Programming languages – C

J2SE 5.0 (September 30, 2004)

- loose specifications,
- claimed to cover a wide range of past and future implementations.

What About the Specs?

Hardware manufacturers document *architectures*:

Intel 64 and IA-32 Architectures Software Developer's Manual
AMD64 Architecture Programmer's Manual
Power ISA specification
ARM Architecture Reference Manual

and programming languages (at best) are defined by *standards*:

ISO/IEC 9899:1999 Programming languages – C
J2SE 5.0 (September 30, 2004)

- loose specifications,
- claimed to cover a wide range of past and future implementations.

Flawed. Always confusing, sometimes wrong.

What About the Specs?

“all that horrible horribly incomprehensible and confusing [...] text that no-one can parse or reason with — not even the people who wrote it”

Anonymous Processor Architect, 2011

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)

x86

Intel/AMD/VIA

Scott Owens, Susmit Sarkar, Francesco Zappa Nardelli, ...

A Cautionary Tale

Intel 64/IA32 and AMD64 - before August 2007 (Era of Vagueness)

A model called *Processor Ordering*, informal prose

Example: Linux Kernel mailing list, 20 Nov 1999 - 7 Dec 1999 (143 posts)

Keywords: speculation, ordering, cache, retire, causality

A one-instruction programming question, a microarchitectural debate!

1. spin_unlock() Optimization On Intel

20 Nov 1999 - 7 Dec 1999 (143 posts) Archive Link: "sp
optimization(i386)"

Topics: BSD: FreeBSD, SMP

People: Linus Torvalds, Jeff V. Merkey, Erich Boleyn,
Spraul, Peter Samuelson, Ingo Molnar

Manfred Spraul thought he'd found a way to shave spir
down from about 22 ticks for the "`lock; btrl $0,%0`" a
to 1 tick for a simple "`movl $0,%0`" instruction, a huge ga
he reported that Ingo Molnar noticed a 4% speed-up in
mark test, making the optimization very valuable.
added that the same optimization cropped up in the
mailing list a few days previously. But Linus Torvalds po
water on the whole thing, saying:

> It does NOT WORK!
>
> Let the FreeBSD people use it, and let them get
> timings. They will crash, eventually.
>
> The window may be small, but if you do this, the
> denly spinlocks aren't reliable any more.

that the piplines are no longer invalid and the b
should be blown out.

I have seen the behavior Linus describes on a
ware analyzer, BUT ONLY ON SYSTEMS
WERE PPRO AND ABOVE. I guess the BSD p
must still be on older Pentium hardware and that
they don't know this can bite in some cases.

Resolved only by appeal to an oracle:

Erich Boleyn, an Architect in an IA32 development grou
also replied to Linus, pointing out a possible miscon
his proposed exploit. Regarding the code Linus post
replied:

It will always return 0. You don't need "
`unlock()`" to be serializing.

The only thing you need is to make sure ther
store in "`spin_unlock()`", and that is kind of tr
the fact that you're changing something to be ob
able on other processors.

The reason for this is that stores can only po
be observed when all prior instructions have r
(i.e. the store is not sent outside of the processo
it is committed state, and the earlier instructio
already committed by that time), so the any
stores, etc absolutely have to have complete
cache-miss or not.

He went on:

Since the instructions for the store in the spin u

IWP and AMD64, Aug. 2007/Oct. 2008 (Era of Causality)

Intel published a white paper (IWP) defining 8 informal-prose principles, e.g.

P1. Loads are not reordered with older loads
P2. Stores are not reordered with older stores

supported by 10 *litmus tests* illustrating allowed or forbidden behaviours, e.g.

Message Passing (MP)

Thread 0	Thread 1
MOV [x]←1 (write x=1)	MOV EAX←[y] (read y=1)
MOV [y]←1 (write y=1)	MOV EBX←[x] (read x=0)
Forbidden Final State: Thread 1:EAX=1 ∧ Thread 1:EBX=0	

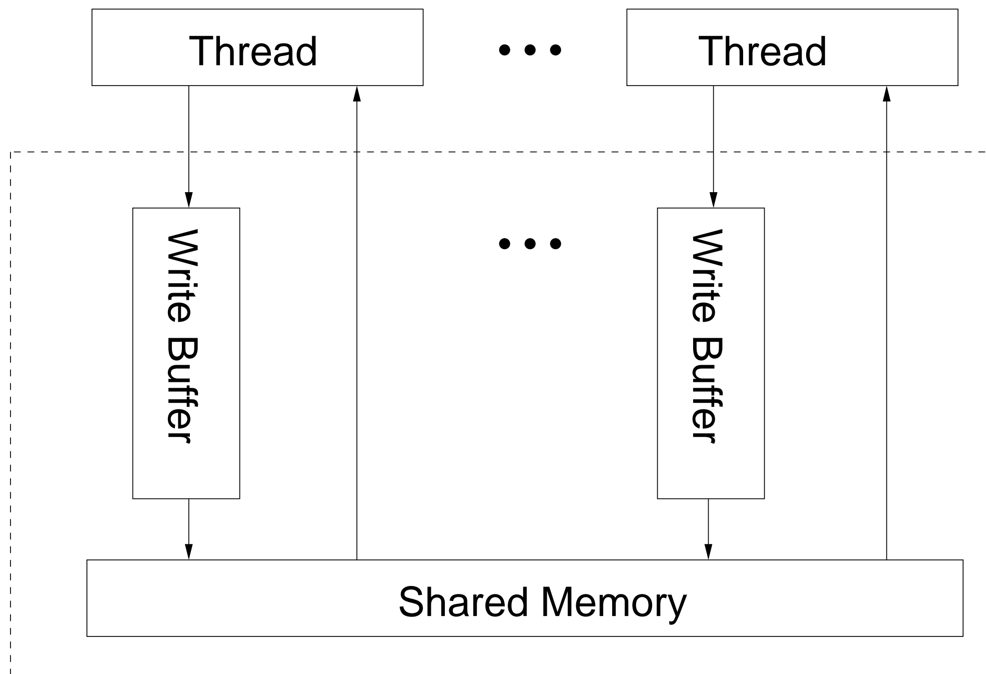
P3. Loads may be reordered with older stores to different locations but not with older stores to the same location

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[y]	(read y=0)	MOV EBX←[x]	(read x=0)
Allowed Final State: Thread 0:EAX=0 $\wedge$ Thread 1:EBX=0			

P3. Loads may be reordered with older stores to different locations but not with older stores to the same location

Store Buffer (SB)

Thread 0		Thread 1	
MOV [x]$\leftarrow$1	(write x=1)	MOV [y]$\leftarrow$1	(write y=1)
MOV EAX$\leftarrow$[y]	(read y=0)	MOV EBX$\leftarrow$[x]	(read x=0)
Allowed Final State: Thread 0:EAX$=$0 $\wedge$ Thread 1:EBX$=$0			

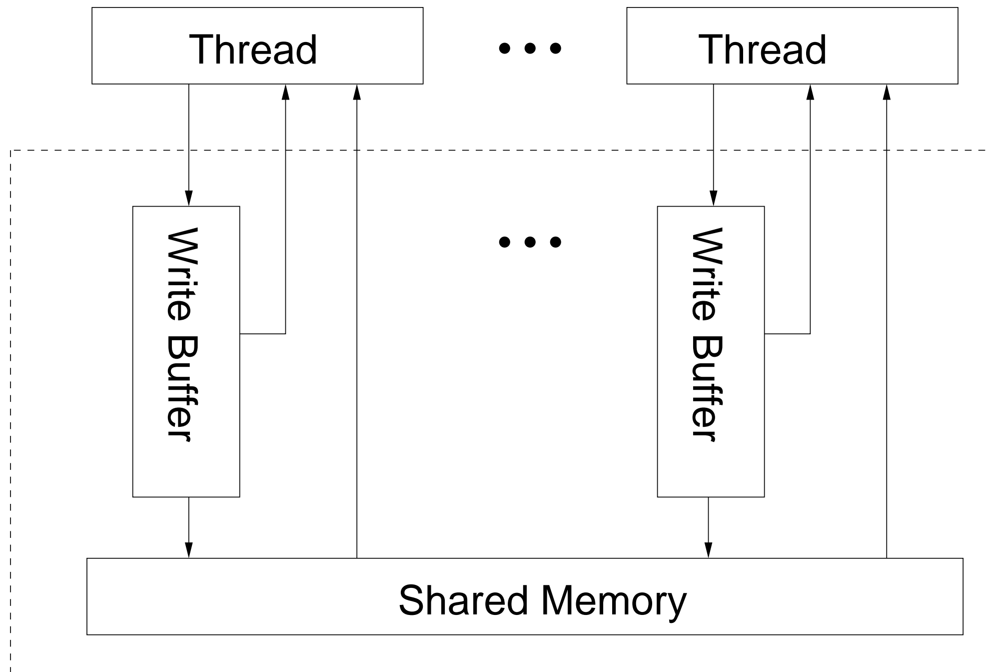


Litmus Test 2.4. Intra-processor forwarding is allowed

Thread 0		Thread 1	
MOV [x]$\leftarrow$1	(write x=1)	MOV [y]$\leftarrow$1	(write y=1)
MOV EAX$\leftarrow$[x]	(read x=1)	MOV ECX$\leftarrow$[y]	(read y=1)
MOV EBX$\leftarrow$[y]	(read y=0)	MOV EDX$\leftarrow$[x]	(read x=0)
Allowed Final State: Thread 0:EBX=0 $\wedge$ Thread 1:EDX=0 Thread 0:EAX=1 $\wedge$ Thread 1:ECX=1			

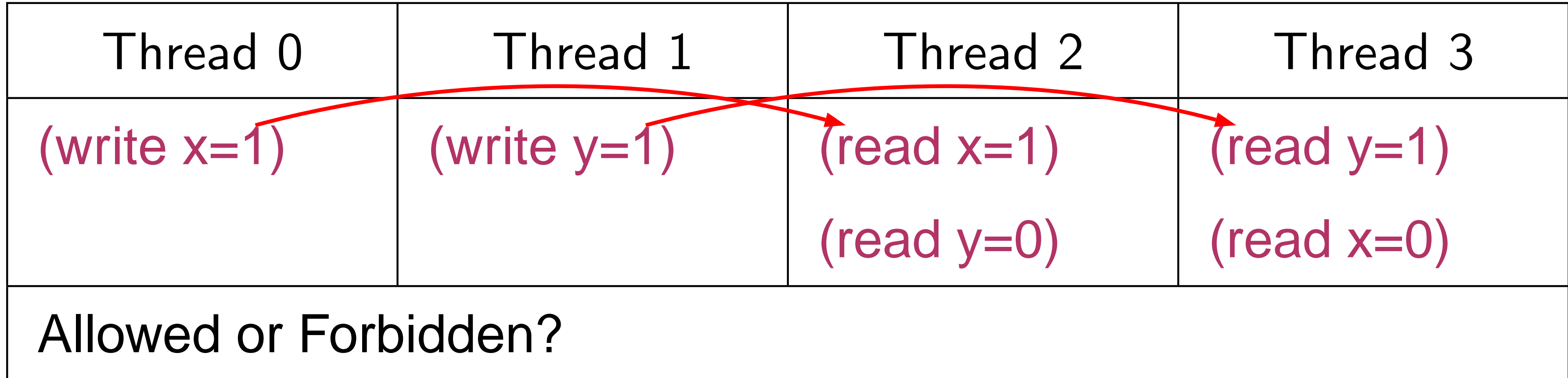
Litmus Test 2.4. Intra-processor forwarding is allowed

Thread 0		Thread 1	
MOV [x]←1	(write x=1)	MOV [y]←1	(write y=1)
MOV EAX←[x]	(read x=1)	MOV ECX←[y]	(read y=1)
MOV EBX←[y]	(read y=0)	MOV EDX←[x]	(read x=0)
Allowed Final State: Thread 0:EBX=0 ∧ Thread 1:EDX=0 Thread 0:EAX=1 ∧ Thread 1:ECX=1			



Problem 1: Weakness

Independent Reads of Independent Writes (IRIW)



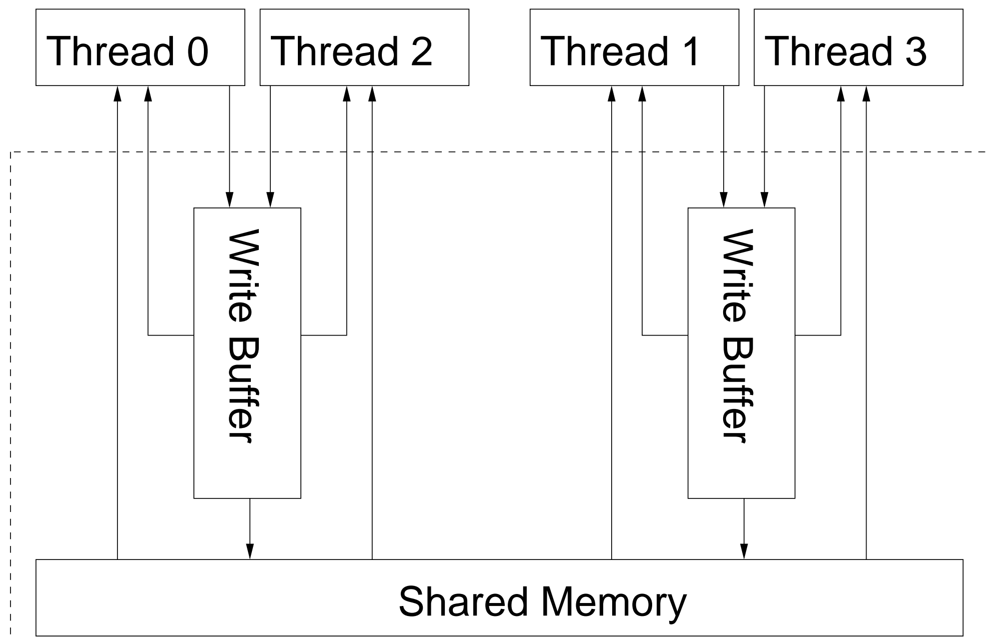
Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1) (read y=0)	(read y=1) (read x=0)
Allowed or Forbidden?			

Problem 1: Weakness

Independent Reads of Independent Writes (IRIW)

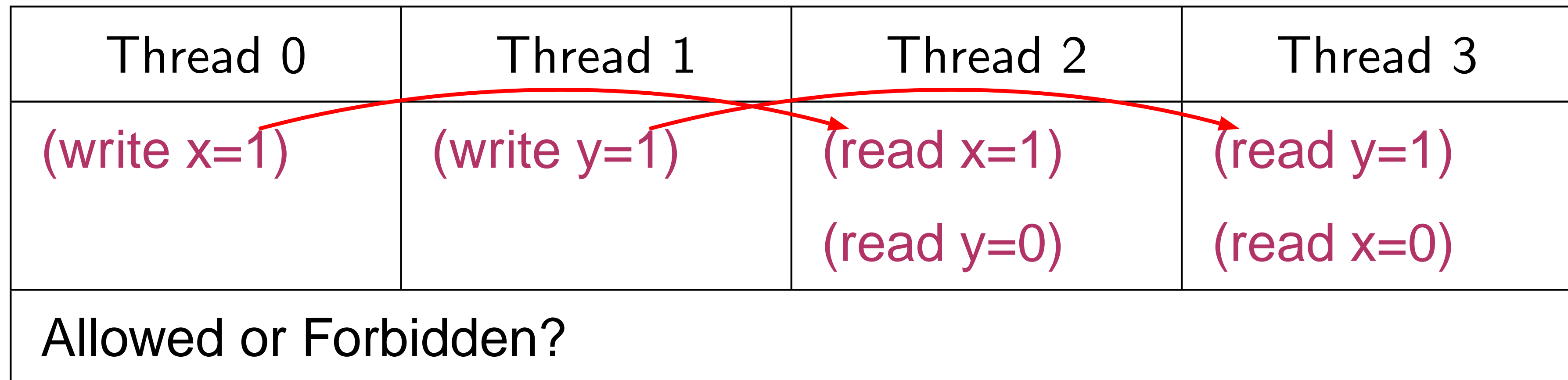
Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1) (read y=0)	(read y=1) (read x=0)
Allowed or Forbidden?			

Microarchitecturally plausible? yes, e.g. with shared store buffers



Problem 1: Weakness

Independent Reads of Independent Writes (IRIW)



Thread 0	Thread 1	Thread 2	Thread 3
(write x=1)	(write y=1)	(read x=1) (read y=0)	(read y=1) (read x=0)
Allowed or Forbidden?			

- AMD3.14: Allowed
- IWP: ???
- Real hardware: unobserved
- Problem for normal programming: ?

Weakness: adding memory barriers does not recover SC, which was assumed in a Sun implementation of the JMM

Problem 2: Ambiguity

P1–4. ...may be reordered with...

P5. 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

Write-to-Read Causality (WRC) (Litmus Test 2.5)

Thread 0	Thread 1	Thread 2
MOV [x]←1 (W x=1)	MOV EAX←[x] (R x=1)	MOV EBX←[y] (R y=1)
	MOV [y]←1 (W y=1)	MOV ECX←[x] (R x=0)
Forbidden Final State: Thread 1:EAX=1 ∧ Thread 2:EBX=1 ∧ Thread 2:ECX=0		

Problem 3: Unsoundness!

Example from Paul Loewenstein:

n6

Thread 0		Thread 1	
MOV [x]←1	(a:W x=1)	MOV [y]←2	(d:W y=2)
MOV EAX←[x]	(b:R x=1)	MOV [x]←2	(e:W x=2)
MOV EBX←[y]	(c:R y=0)		
Allowed Final State: Thread 0:EAX=1 ∧ Thread 0:EBX=0 ∧ x=1			

Observed on real hardware, but not allowed by (any interpretation we can make of) the IWP ‘principles’.

(can see allowed in store-buffer microarchitecture)

Problem 3: Unsoundness!

Example from Paul Loewenstein:

n6

Thread 0		Thread 1	
MOV [x]$\leftarrow$1	(a:W x=1)	MOV [y]$\leftarrow$2	(d:W y=2)
MOV EAX$\leftarrow$[x]	(b:R x=1)	MOV [x]$\leftarrow$2	(e:W x=2)
MOV EBX$\leftarrow$[y]	(c:R y=0)		
Allowed Final State: Thread 0:EAX=1 $\wedge$ Thread 0:EBX=0 $\wedge$ x=1			

In the view of Thread 0:

a$\rightarrow$b by *P4: Reads may [...] not be reordered with older writes to the same location.*

b$\rightarrow$c by *P1: Reads are not reordered with other reads.*

c$\rightarrow$d, otherwise c would read 2 from d

d$\rightarrow$e by *P3. Writes are not reordered with older reads.*

so a:Wx=1 $\rightarrow$ e:Wx=2

But then that should be respected in the final state, by *P6: In a multiprocessor system, stores to the same location have a total order*, and it isn't.

(can see allowed in store-buffer microarchitecture)

Problem 3: Unsoundness!

Example from Paul Loewenstein:

n6

Thread 0		Thread 1	
MOV [x]←1	(a:W x=1)	MOV [y]←2	(d:W y=2)
MOV EAX←[x]	(b:R x=1)	MOV [x]←2	(e:W x=2)
MOV EBX←[y]	(c:R y=0)		
Allowed Final State: Thread 0:EAX=1 ∧ Thread 0:EBX=0 ∧ x=1			

Observed on real hardware, but not allowed by (any interpretation we can make of) the IWP 'principles'.

(can see allowed in store-buffer microarchitecture)

So spec unsound (and also our POPL09 model based on it).

Intel SDM and AMD64, Nov. 2008 –

Intel SDM rev. 29–35 and AMD3.17

Not unsound in the previous sense

Explicitly exclude IRIW, so not weak in that sense. New principle:

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

But, still ambiguous, and the *view by those processors is left entirely unspecified*

Why all these problems?

Recall that the vendor *architectures* are:

- loose specifications;
- claimed to cover a wide range of past and future processor implementations.

Architectures should:

- reveal enough for effective programming;
- without revealing sensitive IP; and
- without unduly constraining future processor design.

There's a big tension between these, compounded by internal politics and inertia.

Fundamental Problem

Architecture texts: *informal prose* attempts at subtle loose specifications

Fundamental problem: prose specifications cannot be used

- to *test programs against*, or
- to *test processor implementations*, or
- to *prove* properties of either, or even
- to *communicate precisely*.

Inventing a Usable Abstraction

Have to be:

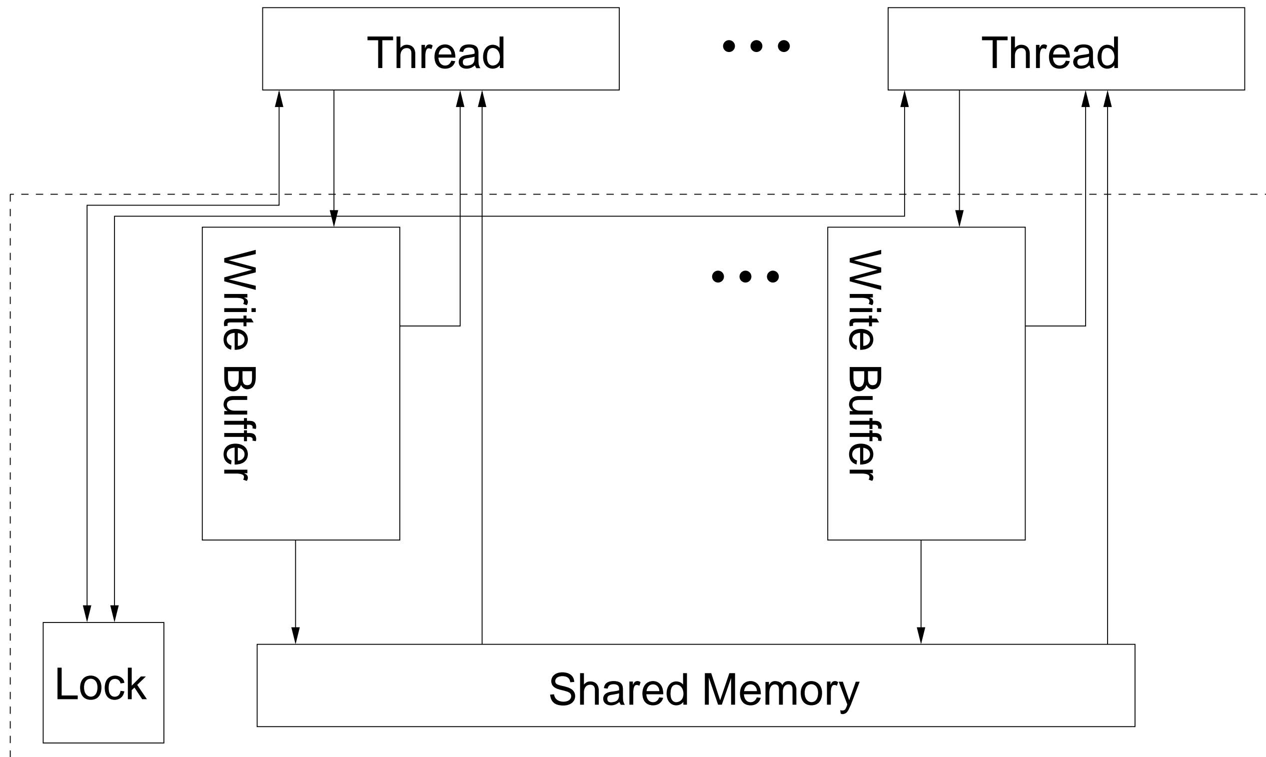
- Unambiguous
- Sound w.r.t. experimentally observable behaviour
- Easy to understand
- Consistent with what we know of vendors intentions
- Consistent with expert-programmer reasoning

Key facts:

- Store buffering (with forwarding) is observable
- IRIW is not observable, and is forbidden by the recent docs
- Various other reorderings are not observable and are forbidden

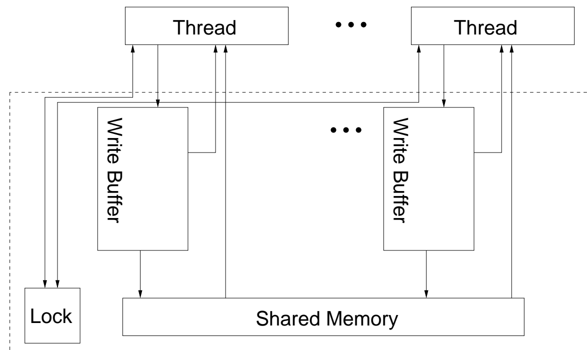
These suggest that x86 is, in practice, like SPARC TSO.

x86-TSO



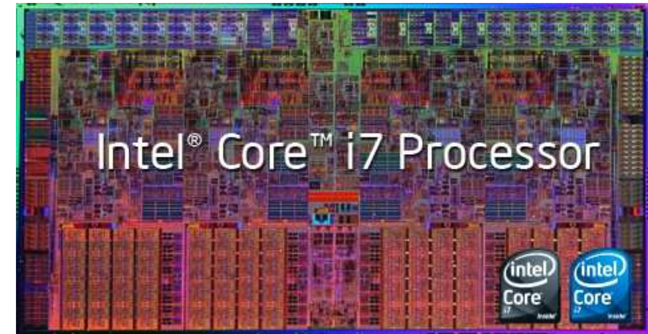
TPHOLs 2009, Scott Owens, Susmit Sarkar, and Peter Sewell
C. ACM 2010, Sewell, Sarkar, Owens, Zappa Nardelli, Myreen

Contrast this *Abstract* Model with the Real Design



$\supseteq_{beh}$

$\neq_{hw}$



Force: Of the internal optimizations of x86 processors, *only* per-thread FIFO write buffers are visible to programmers.

Still quite a loose spec: unbounded buffers, nondeterministic unbuffering, arbitrary interleaving

x86 ISA: Locked Instructions

Thread 0	Thread 1
INC x	INC x

x86 ISA: Locked Instructions

Thread 0	Thread 1
INC x (read x=0; write x=1)	INC x (read x=0; write x=1)
Allowed Final State: [x]=1	

Non-atomic (even in SC semantics)

x86 ISA: Locked Instructions

Thread 0	Thread 1
INC x (read x=0; write x=1)	INC x (read x=0; write x=1)
Allowed Final State: [x]=1	

Non-atomic (even in SC semantics)

Thread 0	Thread 1
LOCK;INC x	LOCK;INC x
Forbidden Final State: [x]=1	

x86 ISA: Locked Instructions

Thread 0	Thread 1
INC x (read x=0; write x=1)	INC x (read x=0; write x=1)
Allowed Final State: [x]=1	

Non-atomic (even in SC semantics)

Thread 0	Thread 1
LOCK;INC x	LOCK;INC x
Forbidden Final State: [x]=1	

Also LOCK'd ADD, SUB, XCHG, etc., and CMPXCHG

x86 ISA: Locked Instructions

Compare-and-swap (CAS):

CMPXCHG dest$\leftarrow$src

compares EAX with dest, then:

- if equal, set ZF=1 and load src into dest,
- otherwise, clear ZF=0 and load dest into EAX

All this is one *atomic* step.

Can use to solve *consensus* problem...

x86 ISA: Memory Barriers

MFENCE memory barrier

(also SFENCE and LFENCE)

Simple x86 Spinlock

The address of x is stored in register eax.

```
acquire:  LOCK DEC [eax]
          JNS enter
spin:     CMP [eax],0
          JLE spin
          JMP acquire
enter:

               critical section

release:  MOV [eax]←1
```

From Linux v2.6.24.7

NB: don't confuse levels — we're using x86 LOCK'd instructions in implementations of Linux spinlocks.

Reasoning above x86-TSO

Theorem 1 *Any program that uses the spinlock correctly (and is otherwise race-free) will behave as if executed on an SC machine*

Proof: via the x86-TSO axiomatic model

Scott Owens, ECOOP 2010

Only the Common-Case Story

What about

- mixed-size accesses
- non-aligned accesses
- self-modifying code
- string instructions and non-temporal instructions
- other memory types
- interactions with virtual memory
- interactions with interrupts
- ...

and hardware transaction support?

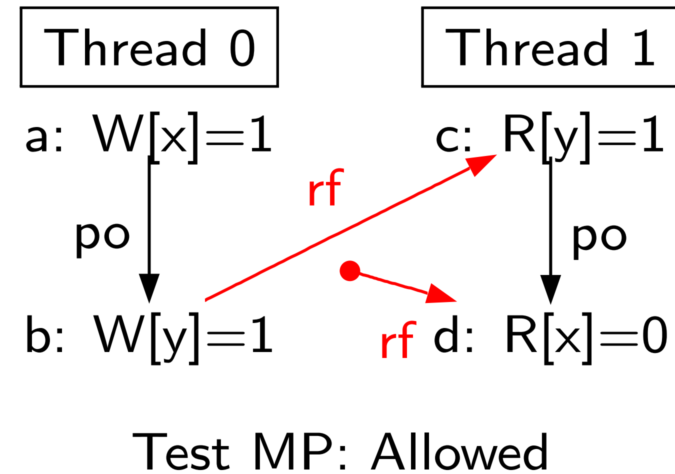
POWER and ARM

Susmit Sarkar, Luc Maranget, Jade Alglave, Derek Williams

Message Passing (MP) Again

MP Pseudocode

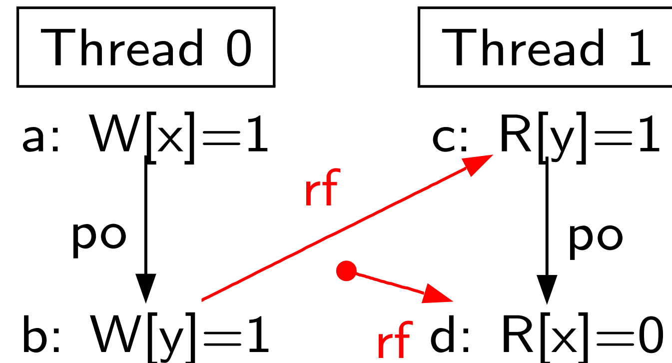
Thread 0	Thread 1
x=1 y=1	r1=y r2=x
Initial state: x=0 ∧ y=0	
Allowed?: 1:r1=1 ∧ 1:r2=0	



Message Passing (MP) Again

MP Pseudocode

Thread 0	Thread 1
x=1 y=1	r1=y r2=x
Initial state: x=0 ∧ y=0	
Allowed: 1:r1=1 ∧ 1:r2=0	

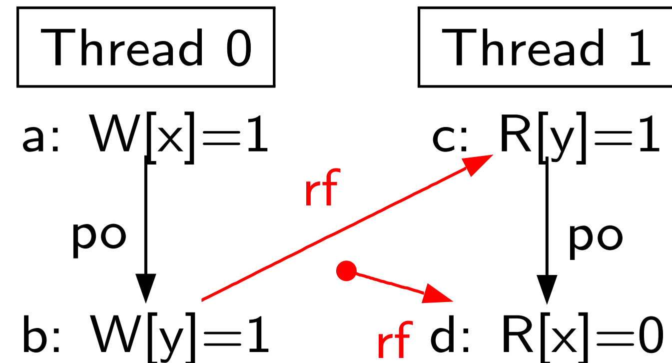


		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP	Allow	10M/4.9G	6.5M/29G	1.7G/167G	40M/3.6G	96k/14M	61k/152M	437k/185M

Message Passing (MP) Again

MP Pseudocode

Thread 0	Thread 1
x=1 y=1	r1=y r2=x
Initial state: x=0 ∧ y=0	
Allowed: 1:r1=1 ∧ 1:r2=0	



Microarchitecturally: writes committed, writes propagated, and/or reads satisfied out-of-order

Enforcing Order with Barriers

MP+dmb/syncs Pseudocode

Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y dmb/sync r2=x
Initial state: x=0 ∧ y=0	
Forbidden: 1:r1=1 ∧ 1:r2=0	

MP+dmbs ARM

Thread 0	Thread 1

MOV R0,#1
STR R0,[R2]
DMB
MOV R1,#1
STR R1,[R3]
	LDR R0,[R3]
	DMB
	LDR R1,[R2]

Initial state: 0:R2=x ∧ 0:R3=y ∧ 1:R2=x ∧ 1:R3=y	
Forbidden: 1:R0=1 ∧ 1:R1=0	

MP+syncs POWER

Thread 0	Thread 1

li r1,1
stw r1,0(r2)
sync
li r3,1
stw r3,0(r4)
	lwz r1,0(r2)
	sync
	lwz r3,0(r4)

Initial state: 0:r2=x ∧ 0:r4=y ∧ 1:r2=y ∧ 1:r4=x	
Forbidden: 1:r1=1 ∧ 1:r3=0	

Enforcing Order with Barriers

MP+dmb/syncs — Pseudocode

Thread 0	Thread 1
x=1	r1=y
dmb/sync	dmb/sync
y=1	r2=x
Initial state: $x=0 \wedge y=0$	
Forbidden: $1{:}r1=1 \wedge 1{:}r2=0$	

MP+dmbs — ARM

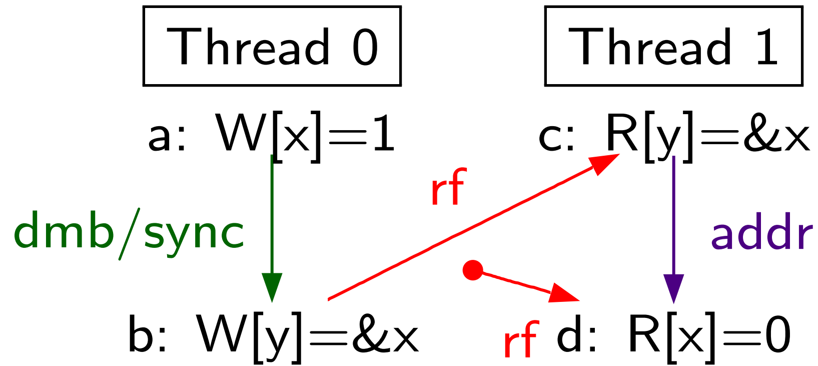
Thread 0	Thread 1
MOV R0,#1	LDR R0,[R3]
STR R0,[R2]	DMB
DMB	LDR R1,[R2]
MOV R1,#1	
STR R1,[R3]	
Initial state: $0{:}R2=x \wedge 0{:}R3=y \wedge 1{:}R2=x \wedge 1{:}R3=y$	
Forbidden: $1{:}R0=1 \wedge 1{:}R1=0$	

MP+syncs — POWER

Thread 0	Thread 1
li r1,1	lwz r1,0(r2)
stw r1,0(r2)	sync
sync	lwz r3,0(r4)
li r3,1	
stw r3,0(r4)	
Initial state: $0{:}r2=x \wedge 0{:}r4=y \wedge 1{:}r2=y \wedge 1{:}r4=x$	
Forbidden: $1{:}r1=1 \wedge 1{:}r3=0$	

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP	Allow	10M/4.9G	6.5M/29G	1.7G/167G	40M/3.6G	96k/14M	61k/152M	437k/185M
MP+dmbs/syncs	Forbid	0/6.9G	0/34G	0/252G	0/12G	0/8.3G	0/10G	0/2.2G
MP+lwsyncs	Forbid	0/6.9G	0/34G	0/220G	—	—	—	—

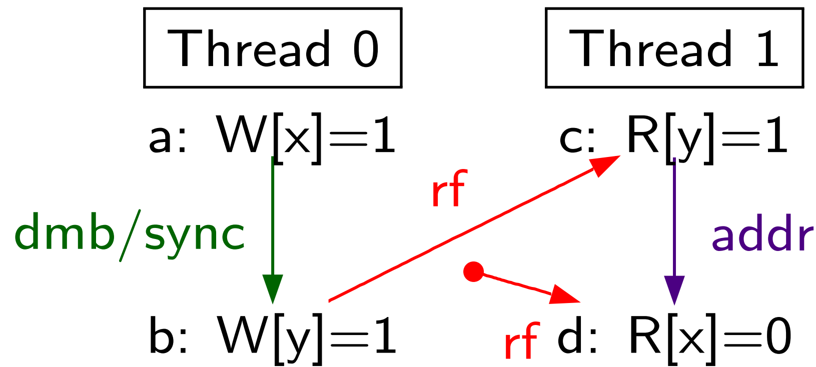
Enforcing Order with Dependencies



MP+dmb/sync+addr′ Pseudocode

Thread 0	Thread 1
x=1	r1=y
dmb/sync	
y=&x	r2=*r1
Initial state: x=0 ∧ y=0	
Forbidden: 1:r1=&x ∧ 1:r2=0	

Enforcing Order with Dependencies



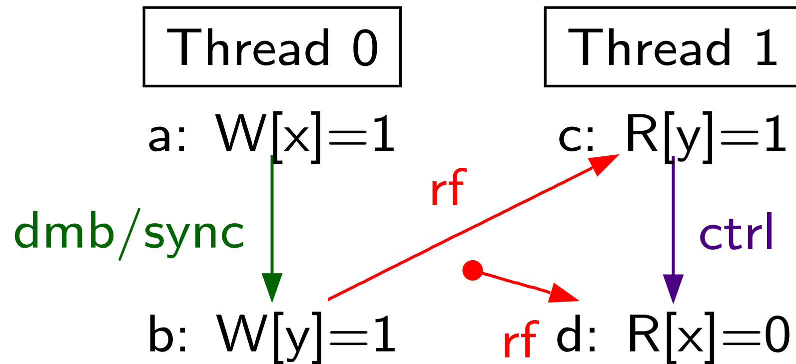
Test MP+dmb/sync+addr: Forbidden

MP+dmb/sync+addr Pseudocode

Thread 0	Thread 1
x=1 dmb/sync y=1	r1=y r3=(r1 xor r1) r2=*(&x + r3)
Initial state: x=0 ∧ y=0	
Forbidden: 1:r1=1 ∧ 1:r2=0	

NB: your compiler will not understand this stuff!

Enforcing Order with Dependencies



Test MP+dmb/sync+ctrl: Allowed

MP+dmb/sync+ctrl

Thread 0	Thread 1
x=1	r1=y
dmb/sync	if (r1 == 1)
y=1	r2=x
Initial state: x=0 ∧ y=0	
Allowed: 1:r1=1 ∧ 1:r2=0	

Fix with ISB/isync instruction between branch and second read

Enforcing Order with Dependencies

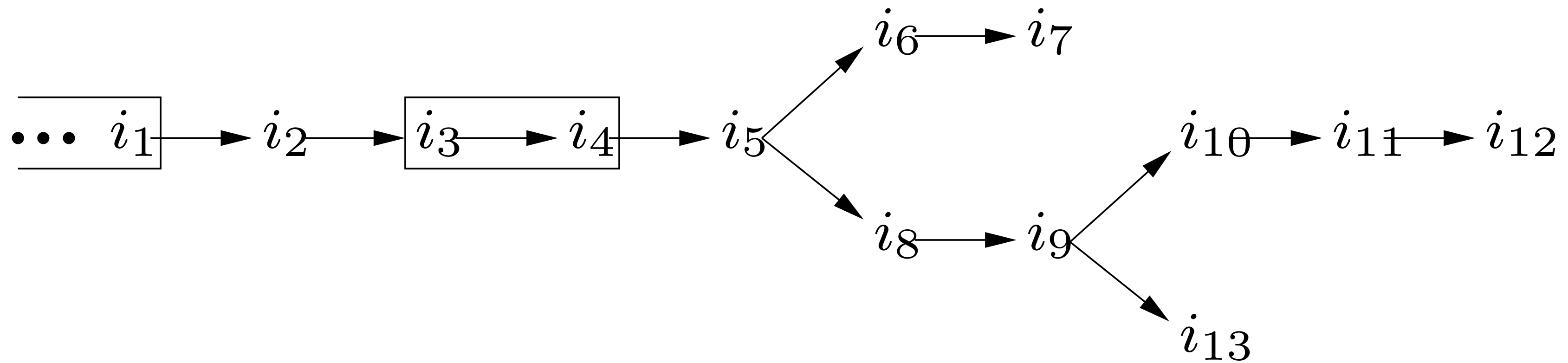
Read-to-Read: address and control-isb/control-isync dependencies respected; control dependencies *not* respected

Read-to-Write: address, data, *and control* dependencies all respected

(all whether natural or artificial)

Core Semantics

Unless constrained, instructions can be executed out-of-order and speculatively

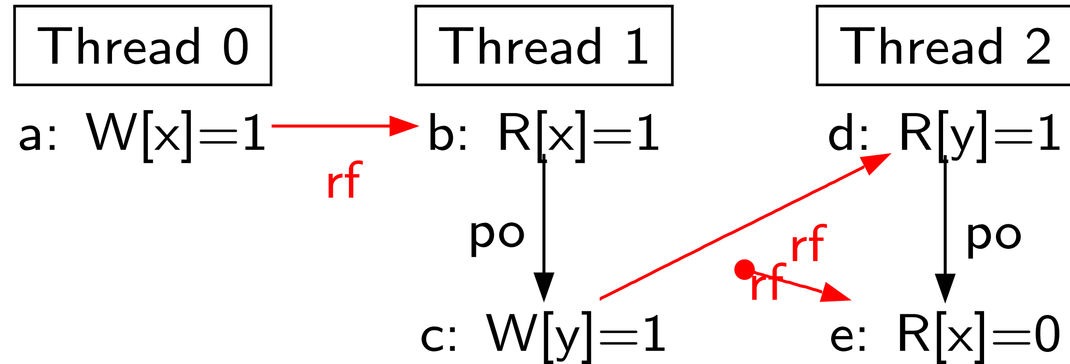


Iterated Message Passing and Cumulative Barriers

WRC-loop Pseudocode

Thread 0	Thread 1	Thread 2
x=1	while (x==0) {} y=1	while (y==0) {} r3=x
Initial state: x=0 ∧ y=0		
Forbidden?: 2:r3=0		

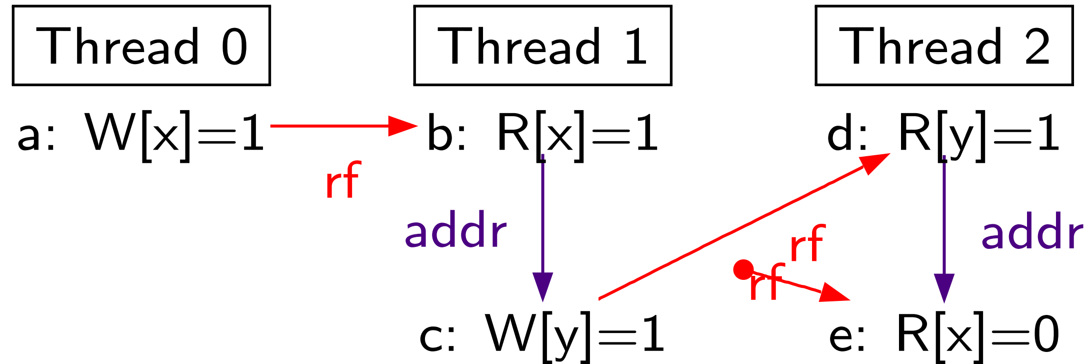
Iterated Message Passing and Cumulative Barriers



WRC Pseudocode

Thread 0	Thread 1	Thread 2
x=1	r1=x y=1	r2=y r3=x
Initial state: x=0 ∧ y=0		
Allowed: 1:r1=1 ∧ 2:r2=1 ∧ 2:r3=0		

Iterated Message Passing and Cumulative Barriers

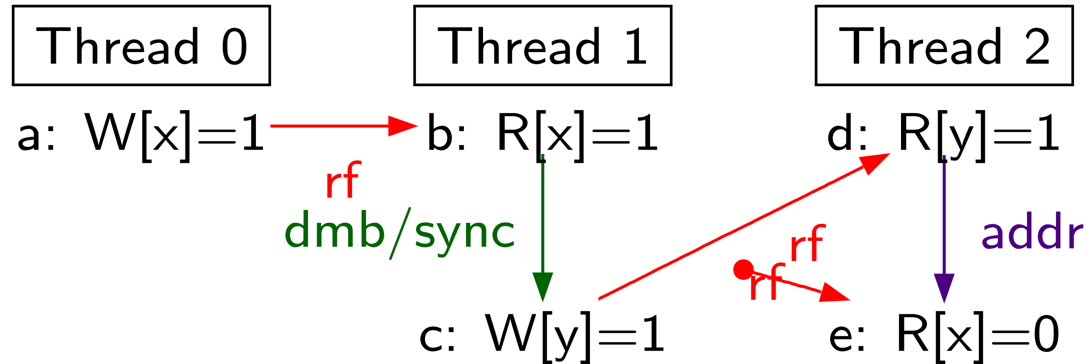


Test WRC+addrs: Allowed

WRC+addrs Pseudocode

Thread 0	Thread 1	Thread 2
x=1	r1=x *(&y+r1-r1) = 1	r2=y r3 = *(&x + r2 - r2)
Initial state: x=0 ∧ y=0		
Allowed: 1:r1=1 ∧ 2:r2=1 ∧ 2:r3=0		

Iterated Message Passing and Cumulative Barriers



Test WRC+dmb/sync+addr: Forbidden

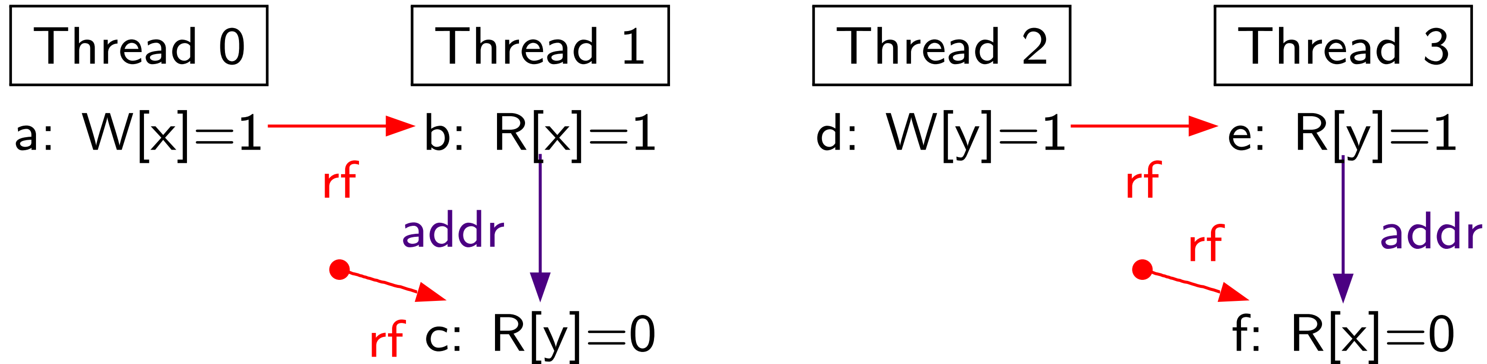
WRC+dmb/sync+addr — Pseudocode

Thread 0	Thread 1	Thread 2
x=1	r1=x dmb/sync y=1	r2=y r3 = *(&x + r2 - r2)
Initial state: x=0 ∧ y=0		
Forbidden: 1:r1=1 ∧ 2:r2=1 ∧ 2:r3=0		

Iterated Message Passing and Cumulative Barriers

		POWER			ARM
	Kind	PowerG5	Power6	Power7	Tegra3
WRC	Allow	44k/2.7G	1.2M/13G	25M/104G	5.9k/7.2M
WRC+addrs	Allow	0/2.4G	225k/4.3G	104k/25G	0/4.0G
WRC+dmb/sync+addr	Forbid	0/3.5G	0/17G	0/158G	0/4.0G
WRC+lwsync+addr	Forbid	0/3.5G	0/17G	0/138G	—
ISA2	Allow	3/91M	72/26M	1.0k/3.8M	4.9k/1.0M
ISA2+dmb/sync+addr+addr	Forbid	0/2.3G	0/8.3G	0/55G	0/4.0G
ISA2+lwsync+addr+addr	Forbid	0/2.3G	0/8.3G	0/55G	—

Independent Reads of Independent Writes



Test IRIW+addrs: Allowed

IRIW+addrs Pseudocode

Thread 0	Thread 1	Thread 2	Thread 3
x=1	r1=x r2=*(&y+r1-r1)	y=1	r3=y r4=*(&x+r3-r3)
Initial state: x=0 ∧ y=0 ∧ z=0			
Allowed: 1:r1=1 ∧ 1:r2=0 ∧ 3:r3=1 ∧ 3:r4=0			

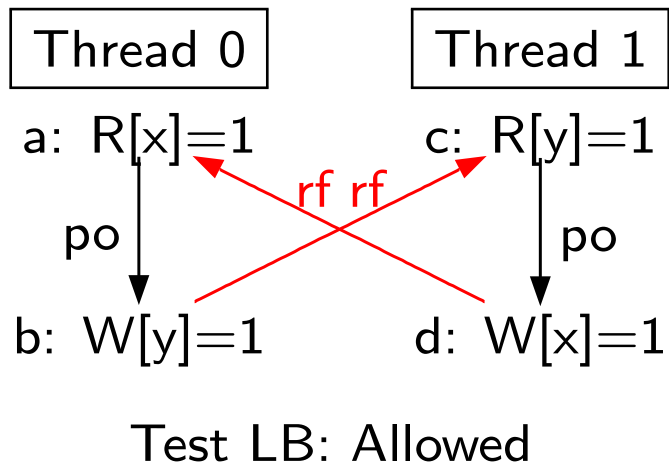
Like SB, this needs two DMBs or syncs (lwsyncs not enough).

Storage Subsystem Semantics

Have to consider writes as *propagating* to *each other thread*

No global memory

Load Buffering (LB)



LB Pseudocode

Thread 0	Thread 1
r1=x y=1	r2=y x=1
Initial state: x=0 ∧ y=0	
Allowed: r1=1 ∧ r2=1	

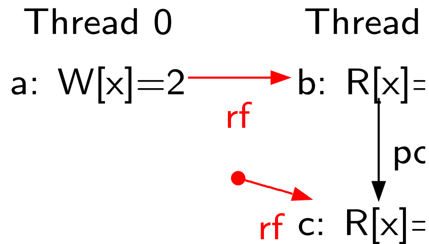
Fix with address or data dependencies:

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
LB	Allow	0/7.4G	0/37G	0/258G	1.5M/3.6G	124k/14M	53/162M	1.3M/185M
LB+addrs	Forbid	0/6.9G	0/34G	0/216G	0/12G	0/8.3G	0/10G	0/2.2G
LB+datas	Forbid	0/6.9G	0/34G	0/252G	0/4.1G	0/3.5G	0/1.6G	0/2.2G

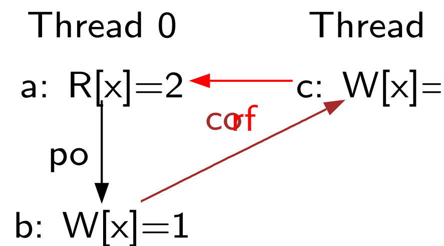
Coherence

Reads and writes to each location in isolation behave SC

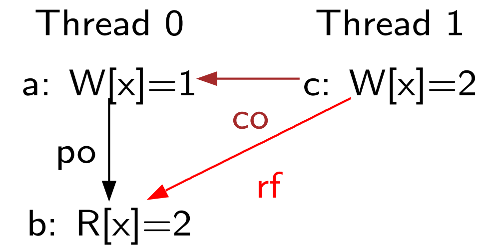
CoRR1: rf,po,fr forbidden



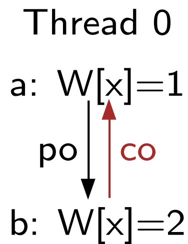
CoRW: rf,po,co forbidden



CoWR: co,fr forbidden

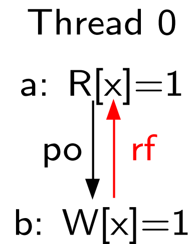


CoWW: po,co forbidden



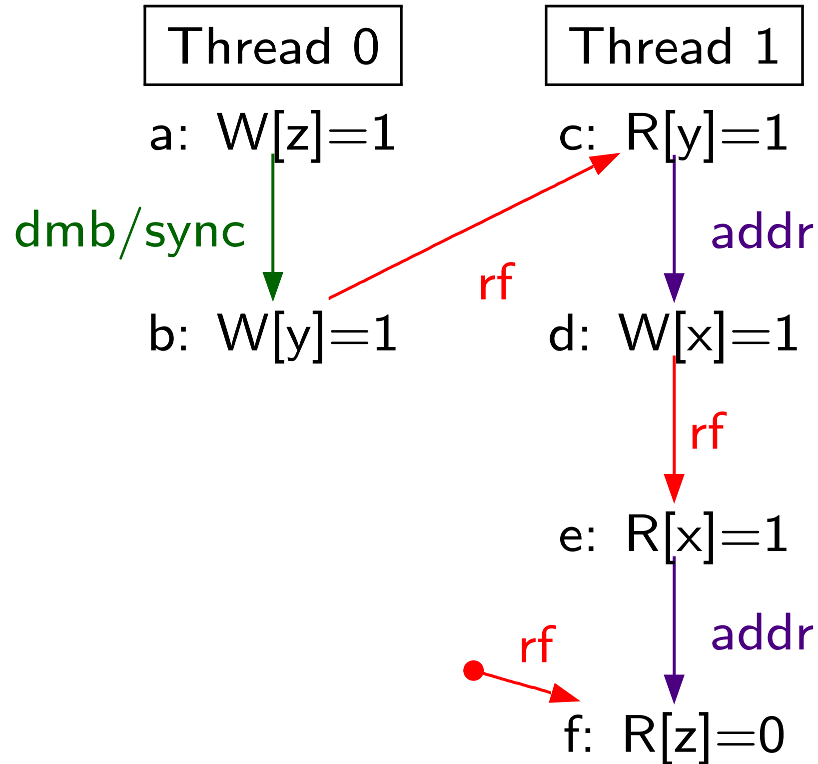
Test CoWW: Forbidden

CoRW1: po,rf forbidden



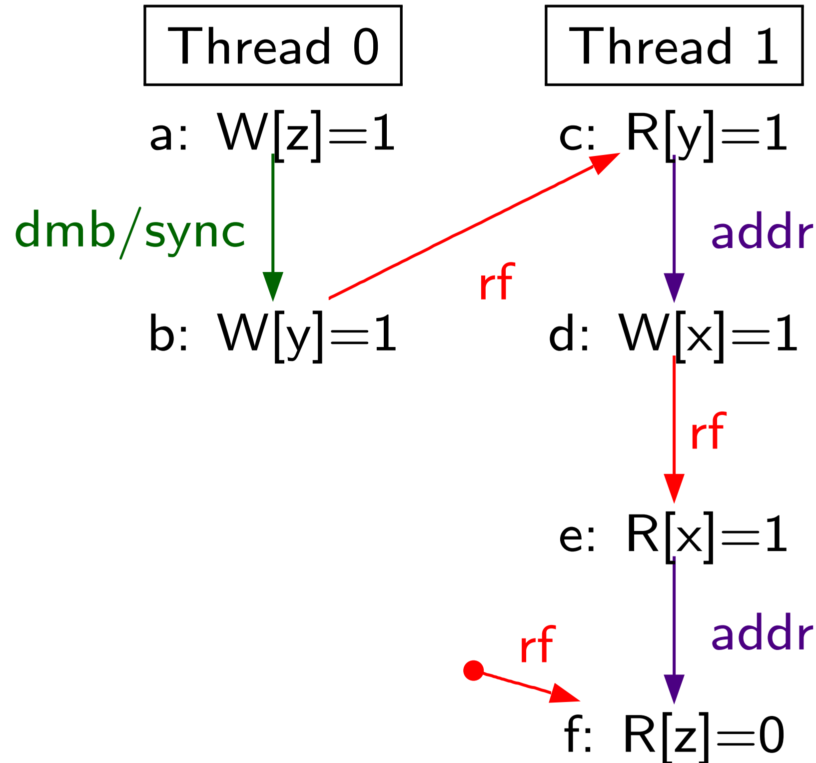
Test CoRW1: Forbidden

Another Cautionary Tale: PPOAA/PPOCA



Test PPOAA: Forbidden

Another Cautionary Tale: PPOAA/PPOCA



		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
PPOCA	Allow	1.1k/3.4G	0/43G	175k/157G	0/12G	0/8.3G	231/159M	0/2.2G
PPOAA	Forbid	0/3.4G	0/40G	0/209G	0/12G	0/8.3G	0/10G	0/2.2G

Under the Hood

1. read docs
2. experiment
3. build formal models
4. tools to compare their predictions vs experiment
5. work with designers
6. prove facts about compilation
7. goto 2

(Papers in POPL09, TPHOLs09, CAV10, POPL11, PLDI11, POPL12, PLDI12, CAV12)

DEMO

Java and C11/C++11

Mark Batty, Suresh Jagannathan, Scott Owens, Susmit Sarkar, Jaroslav Ševčík , Viktor Vafeiadis, Tjark Weber, Francesco Zappa Nardelli

Data-Race Freedom as a Definition

H/W memory models define (albeit loosely) the behaviour of all programs, and we have theorems that race-free programs behave SC. Instead, for PLs can *define*:

- programs that are race-free in SC semantics have SC behaviour
- programs that have a race in some execution in SC semantics can behave in any way at all

Sarita Adve & Mark Hill, 1990

Data-Race Freedom as a Definition

Core of C11 and C++11 [Boehm & Adve, PLDI 2008]. Pro:

- Simple! ‘Programmer-Centric’
- Strong guarantees for most code
- Allows lots of freedom for compiler and hardware optimisations

Con:

- programs that have a race in some execution in SC semantics *can behave in any way at all*
  - Undecidable premise.
  - Imagine debugging: either bug is X ... or there is a potential race in *some* execution
  - No guarantees for untrusted code
- restrictive. Forbids those fancy concurrent algorithms
- need to define exactly what a race is (in libraries?)

Java

Java has integrated multithreading, and it attempts to specify the precise behaviour of concurrent programs.

By the year 2000, the initial specification was shown:

- to allow unexpected behaviours;
- to prohibit common compiler optimisations,
- to be challenging to implement on top of a weakly-consistent multiprocessor.

Superseded around 2004 by the JSR-133 memory model.

The Java Memory Model, Jeremy Manson, Bill Pugh & Sarita Adve, POPL05

Java: JSR-133

- Goal 1: data-race free programs are sequentially consistent;
- Goal 2: all programs satisfy some memory safety and security requirements; (no reads out of thin air)
- Goal 3: common compiler optimisations are sound.

Java: JSR-133 — Unsoundness

The model is intricate, and *fails to meet Goal 3.*: Some optimisations may generate code that exhibits more behaviours than those allowed by the un-optimised source.

As an example, JSR-133 allows `r2=1` in the optimised code below, but forbids `r2=1` in the source code:

x = y = 0	
r1=x y=r1	r2=y x=(r2==1)?y:1

HotSpot optimisation ⟶

x = y = 0	
r1=x y=r1	x=1 r2=y

Jaroslav Ševčík & Dave Aspinall, ECOOP 2008

C11 and C++11

(replacing decades of unfounded reliance on POSIX library spec)

- normal loads and stores
- lock/unlock
- atomic operations (load, store, read-modify-write, ...)
  - `seq_cst`
  - `relaxed, consume, acquire, release, acq_rel`

Idea: if you only use SC atomics, you get DRF guarantee
Non-SC atomics there for experts.

Informal-prose spec., originally flawed in various ways — fixed following formalisation work by Mark Batty

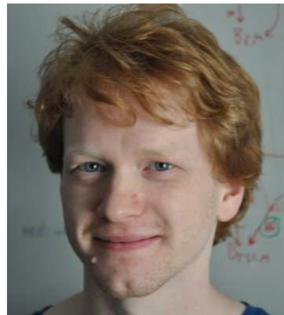
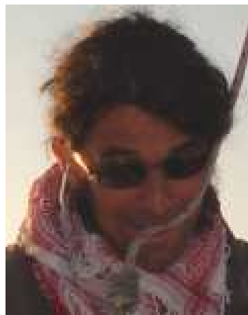
Compiling Down?

- verified compilation scheme from C/C++11 to x86-TSO
- verified compilation scheme from C/C++11 to POWER
- verified compiler (CompCertTSO) from Clight-TSO to x86-TSO

Computer Science?

The End

Thanks!

Jade Alglave, Mark Batty, Luc Maranget, Scott Owens, Susmit Sarkar, Derek Williams, Francesco Zappa Nardelli...