Multicore Semantics and Programming (Lecture 2)

Peter Sewell Tim Harris
Mark Batty

University of Cambridge Oracle

October – November, 2012

Caution: Not *All* of x86

Coherent write-back memory (almost all user and OS code), but assumed

- no exceptions
- no interrupts
- no misaligned or mixed-size accesses
- no ‘non-temporal’ operations
- no device memory
- no self-modifying code
- no page-table changes

For example

Open Question: is every memory write guaranteed to eventually propagate from store buffer to shared memory?

We tentatively assume so (with a progress condition on machine traces).

AMD: yes

Intel: unclear

ARM: yes

What is a Mutex?

Extending the Tiny Language

$location,\ x,\ m$			address
$integer,\ n$			integer
$thread\_id,\ t$			thread id
$expression,\ e$	$::=$		expression
	$\mid$	n	integer literal
	$\mid$	x	read from address x
	$\mid$	$x = e$	write value of e to address x
	$\mid$	$e; e'$	sequential composition
	$\mid$	$e + e'$	plus
	$\mid$	lock x	lock mutex at address x
	$\mid$	unlock x	unlock mutex at address x

Extending the Semantics

Don’t mix addresses used for locks and other addresses.

Say lock is *free* if it holds 1, *taken* otherwise.

Informal semantics: $\mathsf{lock}\,x$ has to *atomically*

1. check the mutex is currently free,
2. change its state to taken, and
3. let the thread proceed.

$\mathsf{unlock}\,x$ has to change its state to free.

Extending the Semantics

$\boxed{M \xrightarrow{t:l} M'}$ M does $t : l$ to become M'

$\boxed{e \xrightarrow{l} e'}$ e does l to become e'

$$\frac{M(x) = n}{M \xrightarrow{t:\mathsf{R}\,x=n} M} \textsc{MRead}$$

$$\frac{}{\mathsf{lock}\,x \xrightarrow{\mathsf{LOCK}\,x} 0} \textsc{Lock}$$

$$\frac{}{M \xrightarrow{t:\mathsf{W}\,x=n} M \oplus (x \mapsto n)} \textsc{MWrite}$$

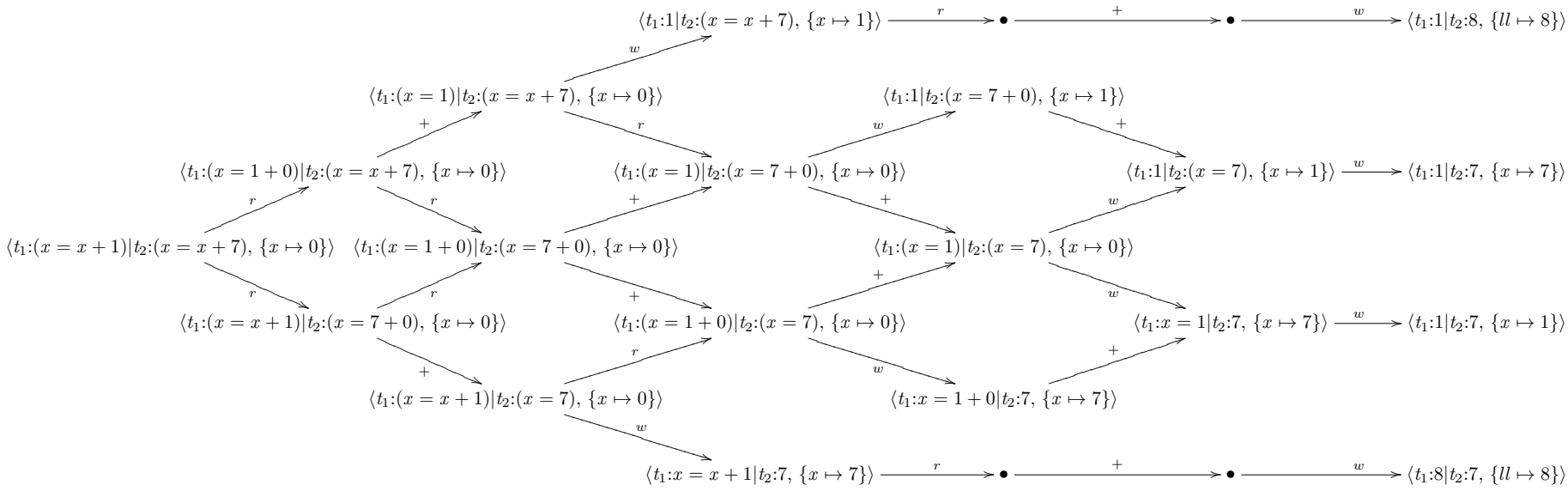
$$\frac{}{\mathsf{unlock}\,x \xrightarrow{\mathsf{UNLOCK}\,x} 0} \textsc{Unlock}$$

$$\frac{M(x) = 1}{M \xrightarrow{t:\mathsf{LOCK}\,x} M \oplus (x \mapsto 0)} \textsc{MLock}$$

$$\frac{}{M \xrightarrow{t:\mathsf{UNLOCK}\,x} M \oplus (x \mapsto 1)} \textsc{MUnlock}$$

Using a Mutex

Recall the behaviour of $t_1{:}x = x + 1 \,|\, t_2{:}x = x + 7$ for the initial store $\{x \mapsto 0\}$:

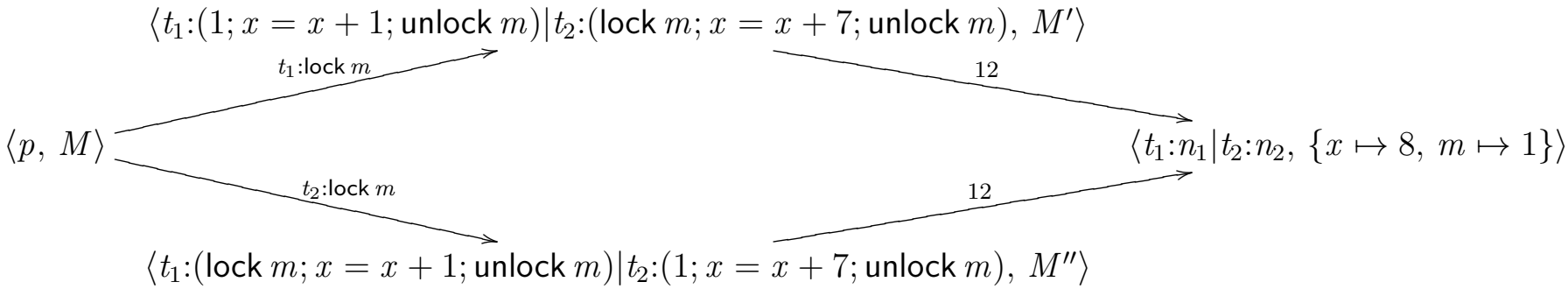


NB: the labels $+$, w and r in this picture are just informal hints as to how those transitions were derived

Using a Mutex

Consider $p =$
$t_1{:}(\mathsf{lock}\ m; x = x + 1; \mathsf{unlock}\ m) | t_2{:}(\mathsf{lock}\ m; x = x + 7; \mathsf{unlock}\ m)$

in the initial store $M = \{x \mapsto 0,\ m \mapsto 1\}$:



(where $M' = M \oplus (m \mapsto 0)$)

Deadlock

lock m can block (that's the point, after all).

Hence, you can *deadlock*.

$$\begin{aligned} p = \quad & t_1{:}(\mathsf{lock}\, m_1; \mathsf{lock}\, m_2; x = 1; \mathsf{unlock}\, m_1; \mathsf{unlock}\, m_2) \\ \mid \quad & t_2{:}(\mathsf{lock}\, m_2; \mathsf{lock}\, m_1; x = 2; \mathsf{unlock}\, m_1; \mathsf{unlock}\, m_2) \end{aligned}$$

Lock Design

Record of which thread is holding a locked lock?

Re-entrancy?

Fairness?

Performance under contention? (backoff?)

Implementing Mutexes with x86 Spinlocks

Suppose register eax holds the address x, which holds 1 if the lock is free or ≤ 0 if taken.

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

         critical section

unlock:  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

Spinlock Example (SC)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		

Spinlock Example (SC)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	

Spinlock Example (SC)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = 0	critical	

Spinlock Example (SC)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = 0	critical	
x = -1	critical	lock

Spinlock Example (SC)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = 0	critical	
x = -1	critical	lock
x = -1	critical	spin, reading x

Spinlock Example (SC)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = 0	critical	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = 1	unlock, writing x	

Spinlock Example (SC)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = 0	critical	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = 1	unlock, writing x	
x = 1		read x

Spinlock Example (SC)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = 0	critical	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = 1	unlock, writing x	
x = 1		read x
x = 0		lock

Spinlock SC Data Race

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = 0	critical	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = 1	unlock, writing x	

Spinlock Example (x86-TSO)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		

Spinlock Example (x86-TSO)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	

Spinlock Example (x86-TSO)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = -1	critical	lock

Spinlock Example (x86-TSO)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = -1	critical	lock
x = -1	critical	spin, reading x

Spinlock Example (x86-TSO)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = -1	unlock, writing x to buffer	

Spinlock Example (x86-TSO)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = -1	unlock, writing x to buffer	
x = -1	...	spin, reading x

Spinlock Example (x86-TSO)

```
lock:LOCK DEC [eax]; JNS enter
spin:CMP [eax],0; JLE spin; JMP lock
enter: ...critical section...
unlock:MOV [eax]←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = -1	unlock, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	

Spinlock Example (x86-TSO)

lock:**LOCK DEC** [eax]; **JNS** enter
spin:**CMP** [eax],0; JLE spin; **JMP** lock
enter: *...critical section...*
unlock:**MOV** [eax]←1

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = -1	unlock, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	
x = 1		read x

Spinlock Example (x86-TSO)

```
lock:LOCK DEC [eax]; JNS enter
spin:CMP [eax],0; JLE spin; JMP lock
enter: ...critical section...
unlock:MOV [eax]←1
```

Shared Memory	Thread 0	Thread 1
x = 1		
x = 0	lock	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = -1	unlock, writing x to buffer	
x = -1	...	spin, reading x
x = 1	write x from buffer	
x = 1		read x
x = 0		lock

Data Race Freedom (DRF)

If all shared-memory accesses in a program are ‘properly protected’ by locks, then it should be ‘race-free’.

Basic Principle (you’d hope):

If a program has no data races in any sequentially consistent (SC) execution, then any relaxed-memory execution is equivalent to some sequentially consistent execution.

NB: premise only involves SC execution.

Data Race Freedom (DRF)

If all shared-memory accesses in a program are ‘properly protected’ by locks, then it should be ‘race-free’.

Basic Principle (you’d hope):

If a program has no data races in any sequentially consistent (SC) execution, then any relaxed-memory execution is equivalent to some sequentially consistent execution.

NB: premise only involves SC execution.

But what *is* a data race?
what does *equivalent* mean?

What is a data race — first attempt

Suppose SC executions are traces of events

- t:R $x{=}n$ for thread t reading value n from address x
- t:W $x{=}n$ for thread t writing value n to address x

(erase τ's, and ignore mutexes and x86 locked instructions and mfence for a moment)

Then say an SC execution has a data race if it contains a pair of adjacent accesses, by different threads, to the same location, that are not both reads:

- $\ldots, t_1$:R $x{=}n_1$, t_2:W $x{=}n_2, \ldots$
- $\ldots, t_1$:W $x{=}n_1$, t_2:R $x{=}n_2, \ldots$
- $\ldots, t_1$:W $x{=}n_1$, t_2:W $x{=}n_2, \ldots$

What is a data race — for x86

1. Need not consider write/write pairs to be races
2. Have to consider SC semantics for LOCK'd instructions (and MFENCE), with events:
   - t:L at the start of a LOCK'd instruction by t
   - t:U at the end of a LOCK'd instruction by t
   - t:B for an MFENCE by thread t
3. Need not consider a LOCK'd read/any write pair to be a race

Say an *x86 data race* is an execution of one of these shapes:

- $\ldots$, t_1:R $x{=}n_1$, t_2:W $x{=}n_2$, $\ldots$
- $\ldots$, t_1:R $x{=}n_1$, t_2:L,$\ldots$,t_2:W $x{=}n_2$, $\ldots$

(or v.v. No t_2:U between the t_2:L and t_2:W $x{=}n_2$)

DRF Principle for x86-TSO

Say an x86 program is *data race free* (DRF) if no SC execution contains an x86 data race.

Theorem 1 (DRF) *If a program is DRF then any x86-TSO execution is equivalent to some SC execution.*

(where *equivalent* means that there is an SC execution with the same subsequence of writes and in which each read reads from the corresponding write)

Proof: via the x86-TSO axiomatic model

Scott Owens, ECOOP 2010

Triangular Races (Owens)

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	W y = v_2
⋮	⋮
W x = v_1	R x
⋮	⋮

Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	W y = v_2
⋮	⋮
W x = v_1	R x
⋮	⋮

Not triangular race

⋮	W y = v_2
⋮	⋮
W x = v_1	W x = w
⋮	⋮

Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	W y = v_2
⋮	⋮
W x = v_1	R x
⋮	⋮

Not triangular race

⋮	W y = v_2
⋮	**mfence**
W x = v_1	R x
⋮	⋮

Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

⋮	W y = v_2
⋮	⋮
W x = v_1	R x
⋮	⋮

Not triangular race

⋮	W y = v_2
⋮	⋮
W x = v_1	**locked** R x
⋮	⋮

Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read

Triangular race

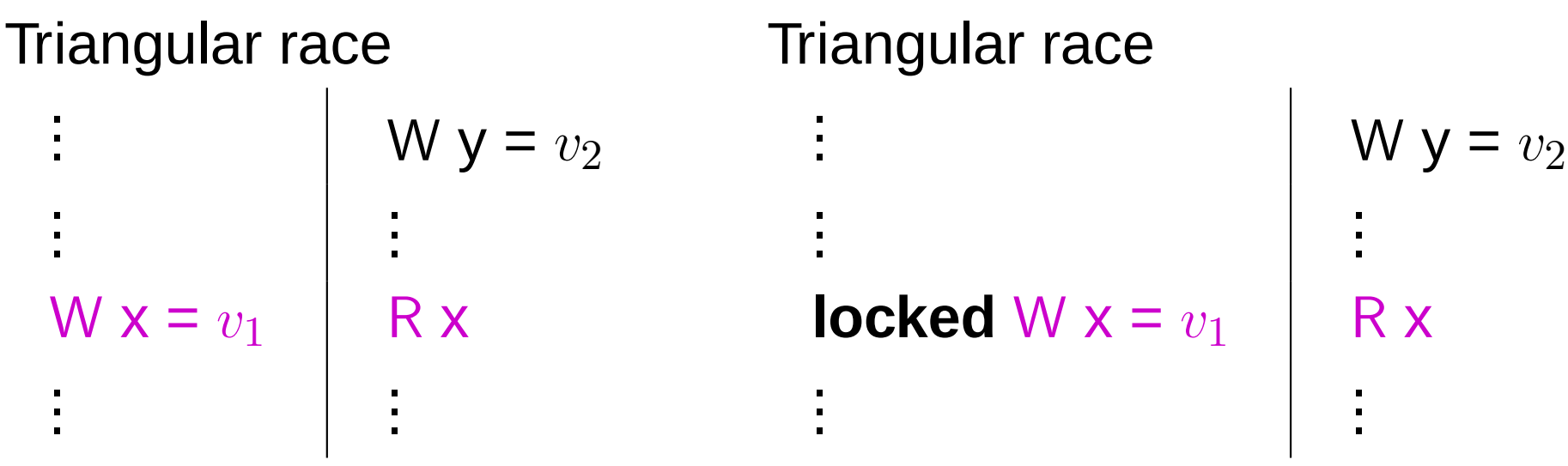
⋮	W y = v_2
⋮	⋮
W x = v_1	R x
⋮	⋮

Not triangular race

⋮	**locked** W y = v_2
⋮	⋮
W x = v_1	R x
⋮	⋮

Triangular Races

- Read/write data race
- Only if there is a bufferable write preceding the read



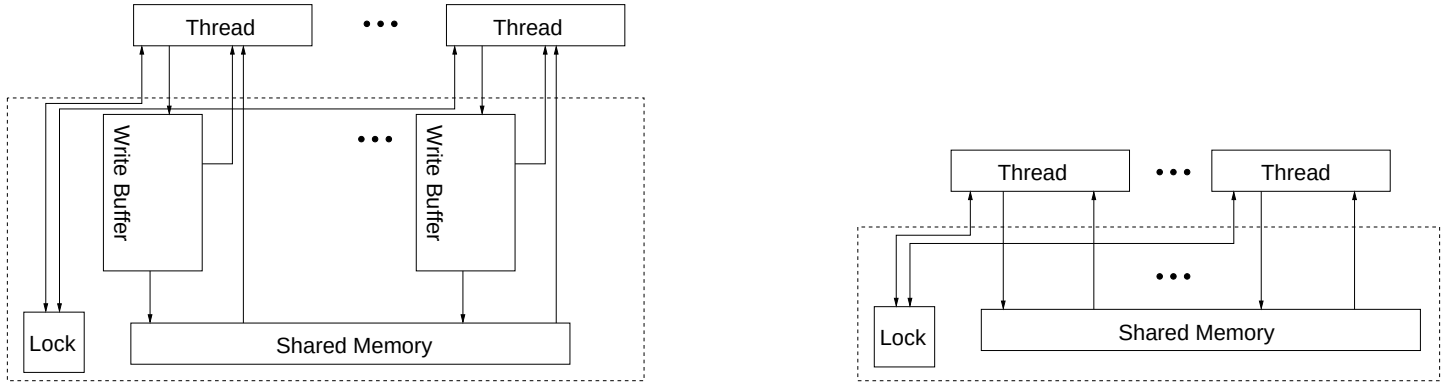
TRF Principle for x86-TSO

Say a program is *triangular race free (TRF)* if no SC execution has a triangular race.

Theorem 2 (TRF) *If a program is TRF then any x86-TSO execution is equivalent to some SC execution.*

If a program has no triangular races when run on a sequentially consistent memory, then

x86-TSO = SC



Spinlock Data Race

```
lock:LOCK DEC [eax]; JNS enter
spin:CMP [eax],0; JLE spin; JMP lock
enter: ...critical section...
unlock:MOV [eax]←1
```

---

x = 1		
x = 0	lock	
x = -1	critical	lock
x = -1	critical	spin, reading x
x = 1	unlock, writing x	

- lock's writes are LOCK'd

Program Correctness

Theorem 3 *Any well-synchronized program that uses the spinlock correctly is TRF.*

Theorem 4 *Spinlock-enforced critical sections provide mutual exclusion.*

Other Applications

A concurrency bug in the HotSpot JVM

- Found by Dave Dice (Sun) in Nov. 2009
- `java.util.concurrent.LockSupport` (‘Parker’)
- Platform specific C++
- Rare hung thread
- Since “day-one” (missing MFENCE)
- Simple explanation in terms of TRF

Also: Ticketed spinlock, Linux SeqLocks, Double-checked locking

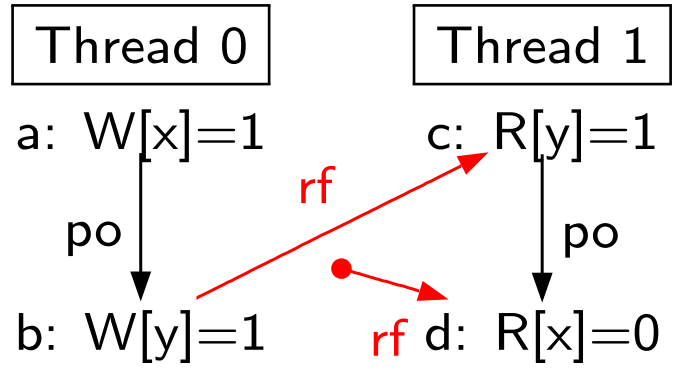
POWER and ARM

Susmit Sarkar, Luc Maranget, Jade Alglave,
Derek Williams, Peter Sewell

Message Passing (MP) Again

MP Pseudocode

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

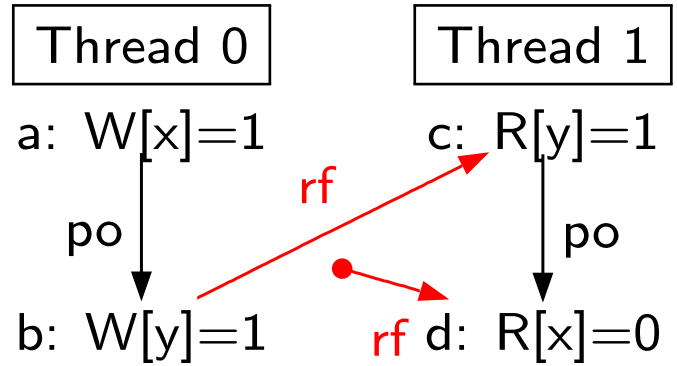


Test MP: Allowed

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	



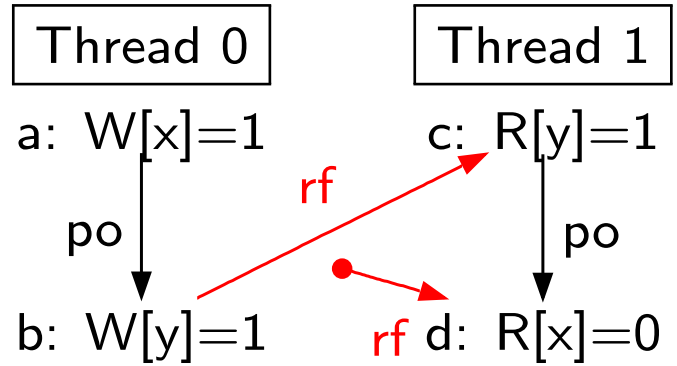
Test MP: Allowed

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP	Allow	10M/4.9G	6.5M/29G	1.7G/167G	40M/3.8G	138k/16M	61k/552M	437k/185M

Message Passing (MP) Again

MP Pseudocode

Thread 0	Thread 1
x=1 y=1	r1=y r2=x
Initial state: x=0 $\wedge$ y=0	
Allowed: 1:r1=1 $\wedge$ 1:r2=0	



Test MP: Allowed

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 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

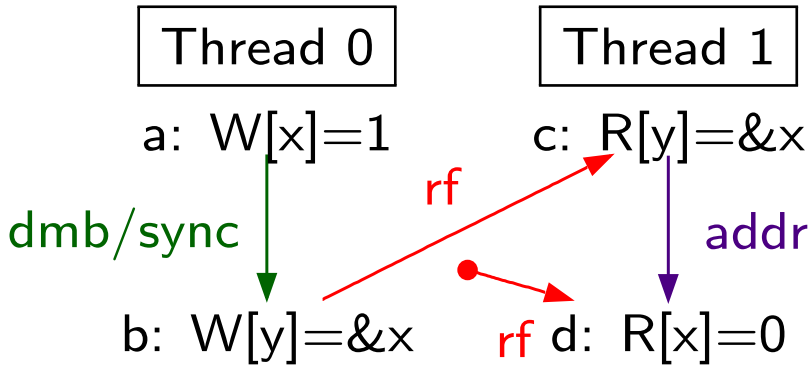
Thread 0	Thread 1

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	

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP	Allow	10M/4.9G	6.5M/29G	1.7G/167G	40M/3.8G	138k/16M	61k/552M	437k/185M
MP+dmbs/syncs	Forbid	0/6.9G	0/40G	0/252G	0/24G	0/39G	0/26G	0/2.2G
MP+lwsyncs	Forbid	0/6.9G	0/40G	0/220G	—	—	—	—

Enforcing Order with Dependencies

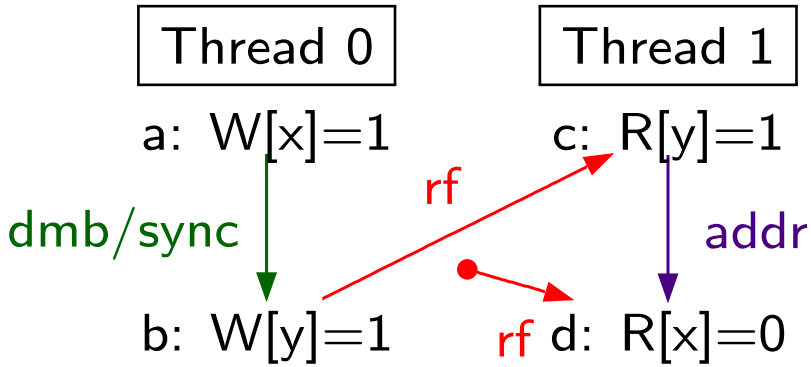


Test MP+dmb/sync+addr': Forbidden

MP+dmb/sync+addr′ Pseudocode

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

Enforcing Order with Dependencies



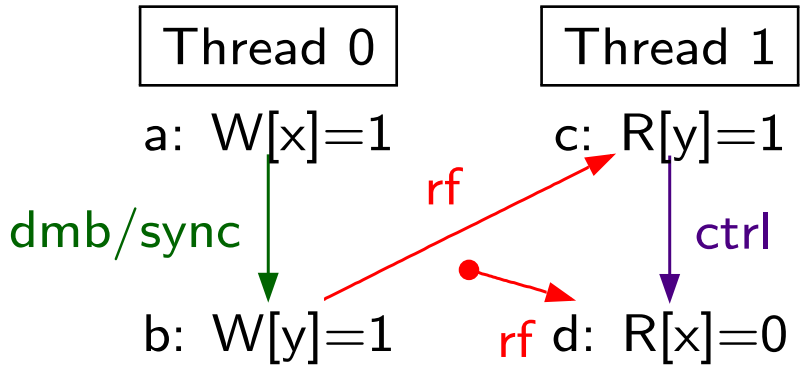
Test MP+dmb/sync+addr: Forbidden

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 dmb/sync y=1	r1=y if (r1 == 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

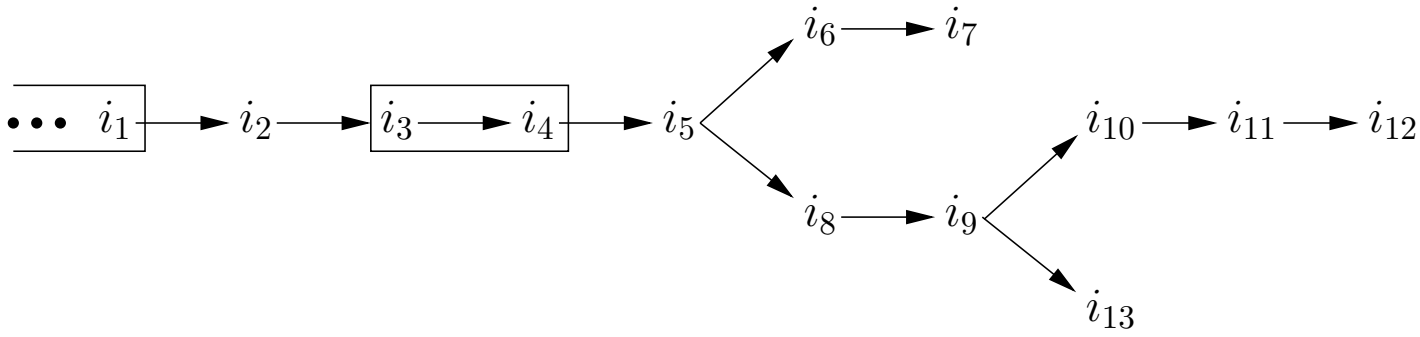
Read-to-Read: address and control-isb/control-isync dependencies respected; control dependencies *not* respected

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

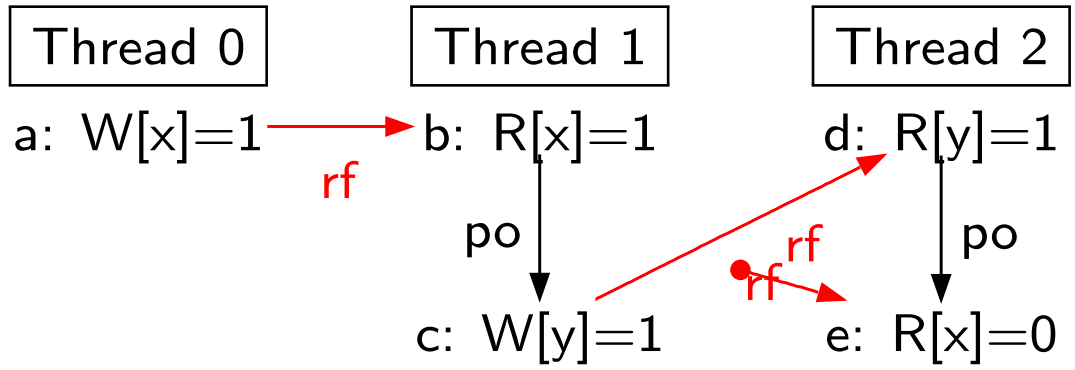


Iterated Message Passing and Cumulative Barriers

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 $\wedge$ y=0		
Forbidden?: 2:r3=0		

Iterated Message Passing and Cumulative Barriers

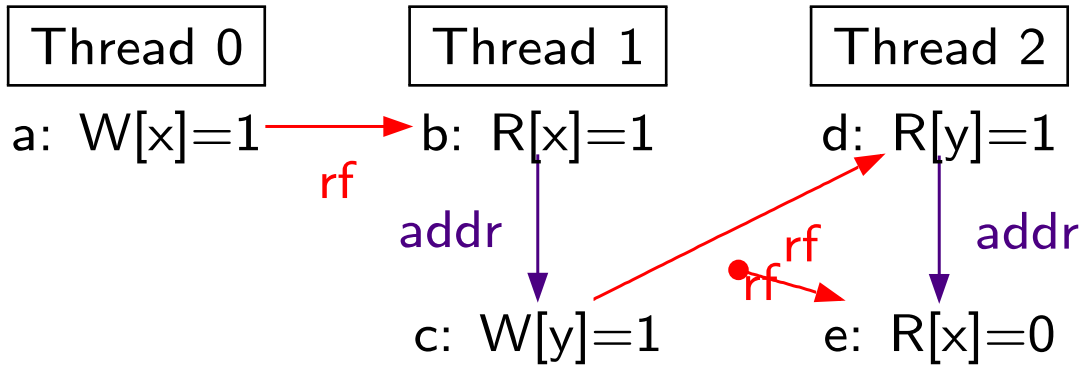


Test WRC: Allowed

WRC Pseudocode

Thread 0	Thread 1	Thread 2
x=1	r1=x y=1	r2=y r3=x
Initial state: x=0 $\wedge$ y=0		
Allowed: 1:r1=1 $\wedge$ 2:r2=1 $\wedge$ 2:r3=0		

Iterated Message Passing and Cumulative Barriers

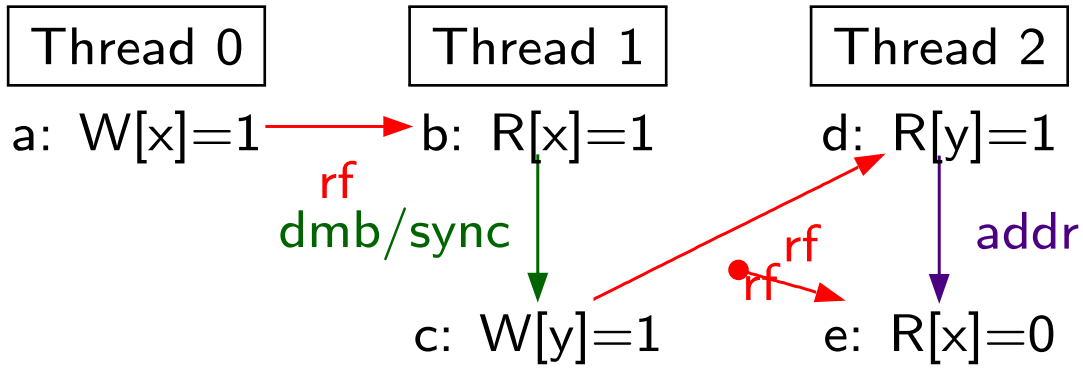


Test WRC+addrs: Allowed

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

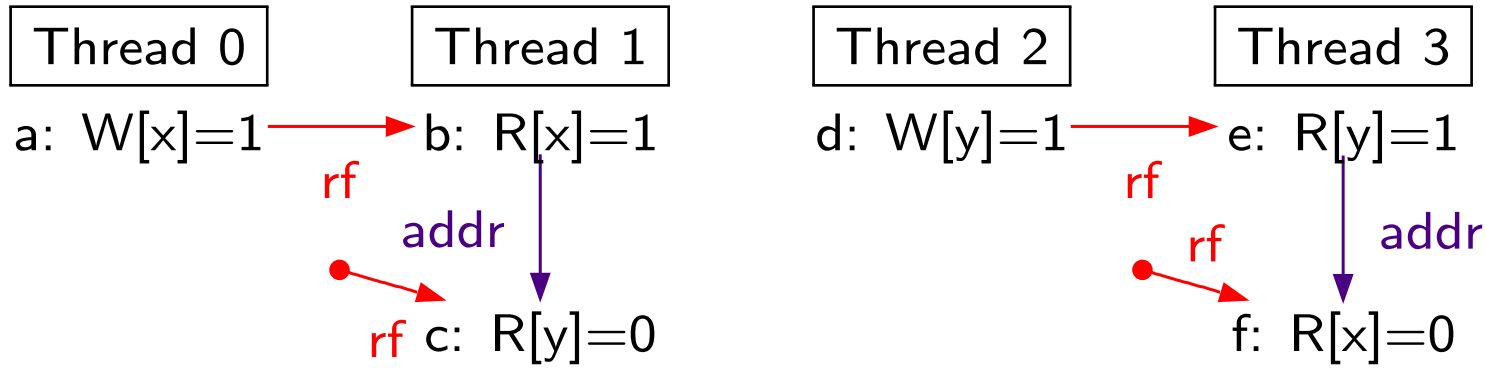
WRC+dmb/sync+addr Pseudocode

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	8.6k/8.2M
WRC+addrs	Allow	0/2.4G	225k/4.3G	104k/25G	0/20G
WRC+dmb/sync+addr	Forbid	0/3.5G	0/21G	0/158G	0/20G
WRC+lwsync+addr	Forbid	0/3.5G	0/21G	0/138G	—
ISA2	Allow	3/91M	73/30M	1.0k/3.8M	6.7k/2.0M
ISA2+dmb/sync+addr+addr	Forbid	0/2.3G	0/12G	0/55G	0/20G
ISA2+lwsync+addr+addr	Forbid	0/2.3G	0/12G	0/55G	—

Independent Reads of Independent Writes



Test IRIW+addrs: Allowed

IRIW+addrs Pseudocode

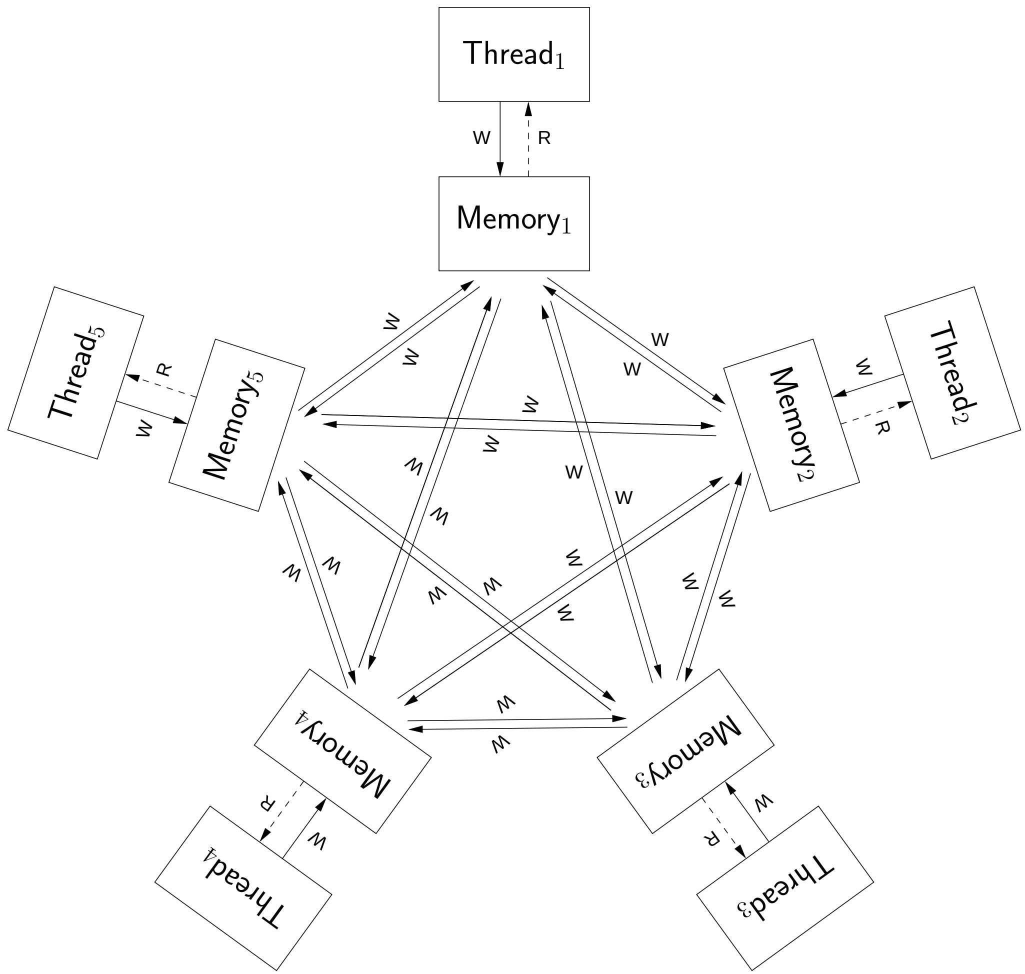
Thread 0	Thread 1	Thread 2	Thread 3
x=1	r1=x r2=*(&y+r1-r1)	y=1	r3=y r4=*(&x+r3-r3)
Initial state: x=0 ∧ y=0 ∧ z=0			
Allowed: 1:r1=1 ∧ 1:r2=0 ∧ 3:r3=1 ∧ 3:r4=0			

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

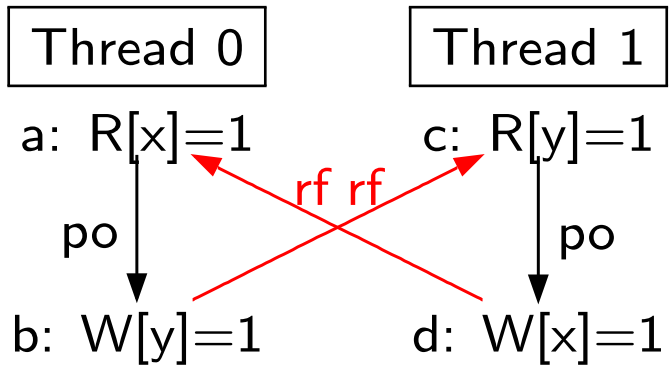
Storage Subsystem Semantics

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

No global memory



Load Buffering (LB)



Test LB: Allowed

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	

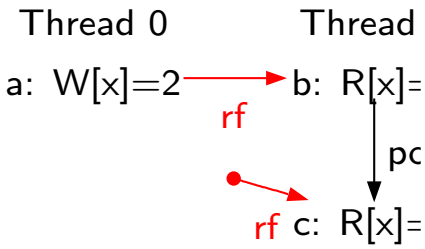
Fix with address or data dependencies:

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
LB	Allow	0/7.4G	0/43G	0/258G	1.5M/3.9G	124k/16M	58/1.6G	1.3M/185M
LB+addrs	Forbid	0/6.9G	0/40G	0/216G	0/24G	0/39G	0/26G	0/2.2G
LB+datas	Forbid	0/6.9G	0/40G	0/252G	0/16G	0/23G	0/18G	0/2.2G
LB+ctrls	Forbid	0/4.5G	0/16G	0/88G	0/8.1G	0/7.5G	0/1.6G	0/2.2G

Coherence

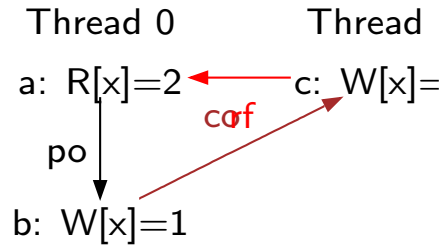
Reads and writes to each location in isolation behave SC

CoRR1: rf,po,fr forbidden



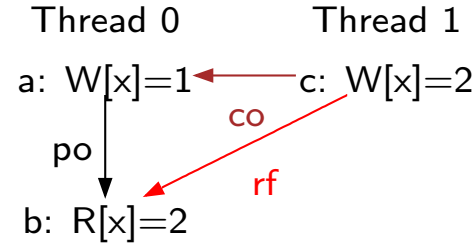
Test CoRR1

CoRW: rf,po,co forbidden



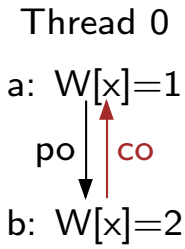
Test CoRW

CoWR: co,fr forbidden



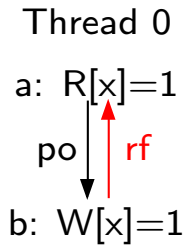
Test CoWR

CoWW: po,co forbidden



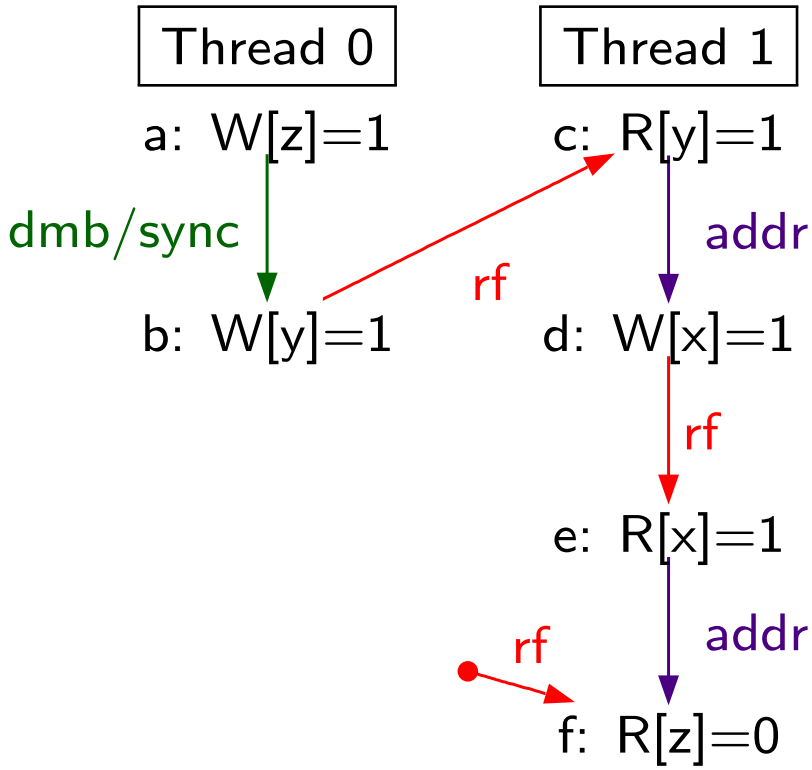
Test CoWW: Forbidden

CoRW1: po,rf forbidden



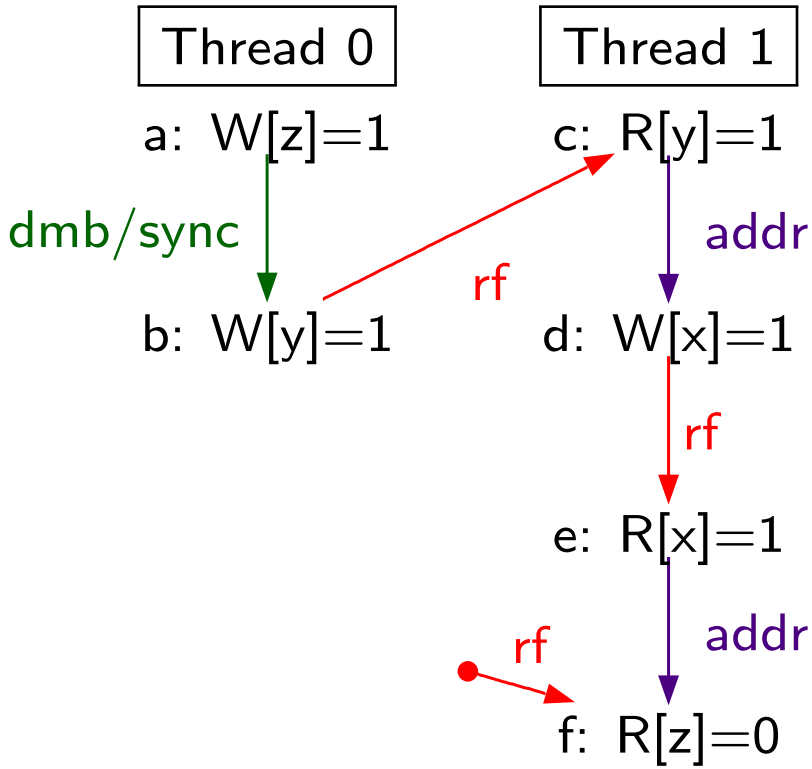
Test CoRW1: Forbidden

Another Cautionary Tale: PPOAA/PPOCA



Test PPOAA: Forbidden

Another Cautionary Tale: PPOAA/PPOCA



Test PPOAA: Forbidden

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
PPOCA	Allow	1.1k/3.4G	0/49G	175k/157G	0/24G	0/39G	233/743M	0/2.2G
PPOAA	Forbid	0/3.4G	0/46G	0/209G	0/24G	0/39G	0/26G	0/2.2G

Basic Question

What *is* the concurrency semantics of Power/ARM processors?

We've built a model...

[Susmit Sarkar, Jade Alglave, Luc Maranget, Derek Williams, PS]

...by a long process of

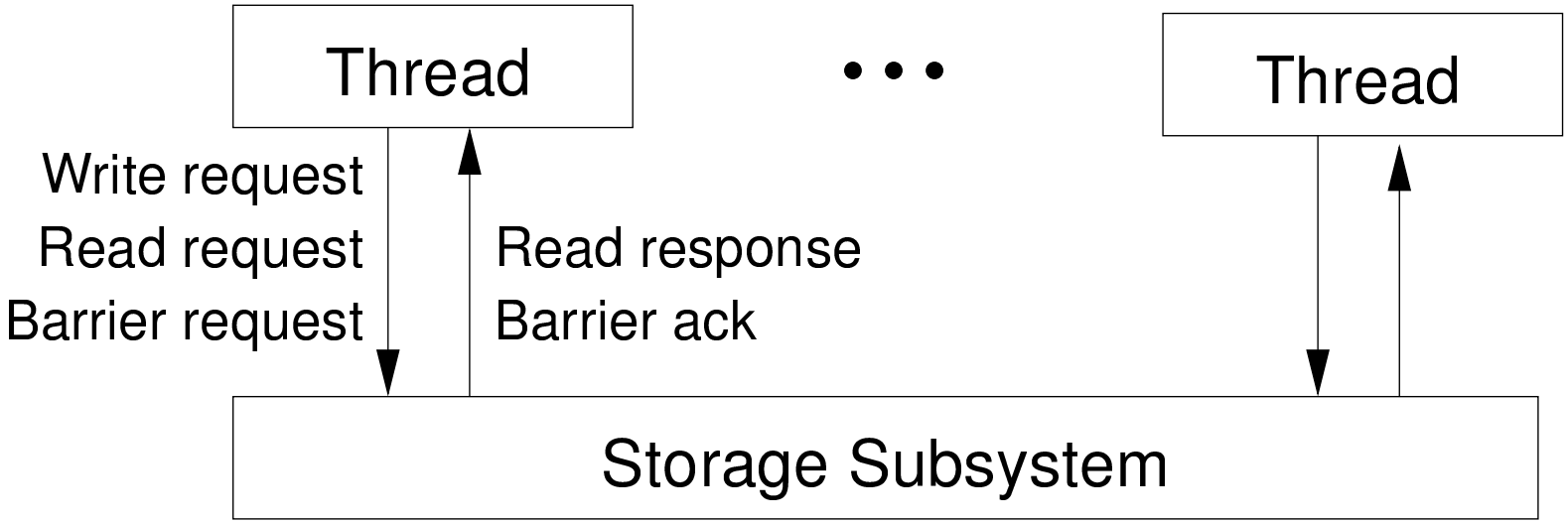
- generating test cases
- experimental testing of hardware
- talking with IBM and ARM architects
- checking candidate models

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

Operational Model

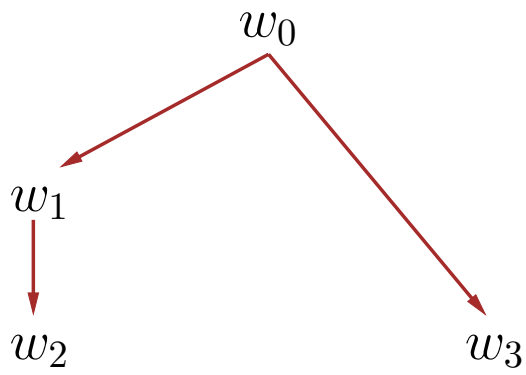
Operational abstract-machine model:

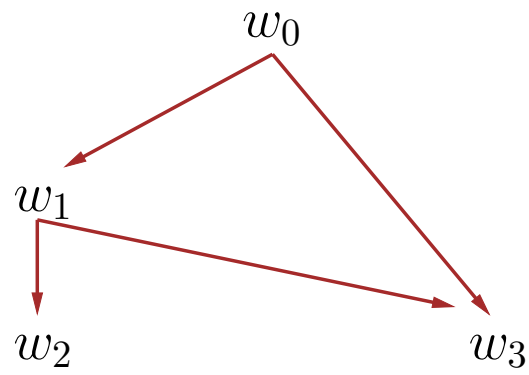
- thread-local LTS *(speculation)*
- storage subsystem LTS *(propagation)*
- top-level LTS parallel composition of those



Coherence by Fiat

Suppose the storage subsystem has seen 4 writes to x:





Suppose just $[w_1]$ has propagated to tid and then tid reads x.

- it cannot be sent w_0, as w_0 is coherence-before the w_1 write that (because it is in the writes-propagated list) it might have read from;
- it could re-read from w_1, leaving the coherence constraint unchanged;
- it could be sent w_2, again leaving the coherence constraint unchanged, in which case w_2 must be appended to the events propagated to *tid*; or
- it could be sent w_3, again appending this to the events propagated to *tid*, which moreover entails committing to w_3 being coherence-after w_1, as in the coherence constraint on the right above. Note that this still leaves the relative order of w_2 and w_3 unconstrained, so another thread could be sent w_2 then w_3 or (in a different run) the other way around (or indeed just one, or neither).

Model States

Storage subsystem:

- thread ids (set)
- writes seen (set)
- coherence (strict partial order over writes, per-address)
- writes past coherence point (set)
- events propagated to each thread (list of writes and barriers)

Thread:

- initial register state
- tree of committed and in-flight instructions
- unacknowledged sync/dmb barriers

Sample Transition Rule

Propagate write to another thread (a τ transition)

The storage subsystem can propagate a write w (by thread tid) that it has seen to another thread tid', if:

- the write has not yet been propagated to tid';
- w is coherence-after any write to the same address that has already been propagated to tid'; and
- all barriers that were propagated to tid before w (in $s.events\_propagated\_to\ (tid)$) have already been propagated to tid'.

Action: append w to $s.events\_propagated\_to\ (tid')$.

Explanation: This rule advances the thread tid' view of the coherence order to w, which is needed before tid' can read from w, and is also needed before any barrier that has w in its "Group A" can be propagated to tid'.

DEMO

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

...periodic table