POWER and ARM

IBM POWER: high-end server processor

POWER 8: up to 192 cores, each with up to 8 h/w threads
`https://en.wikipedia.org/wiki/POWER8`

Power7: IBM's Next-Generation Server Processor. Kalla, R.; Sinharoy, B.; Starke, W.J.; Floyd, M.
`http://www.hotchips.org/wp-content/uploads/hc_archives/hc2`

ARMv8-A: 64-bit application-class (vs microcontrollers)

Cores designed by ARM and by others, in various SoCs.
`https://en.wikipedia.org/wiki/Comparison_of_ARMv8-A_cores`

- Samsung Exynos 7420 and Qualcomm Snapdragon 810, containing 4xCortex-A57+4xCortex-A53
- Nvidia Denver
- ...

POWER and ARM

Much weaker than x86-TSO:

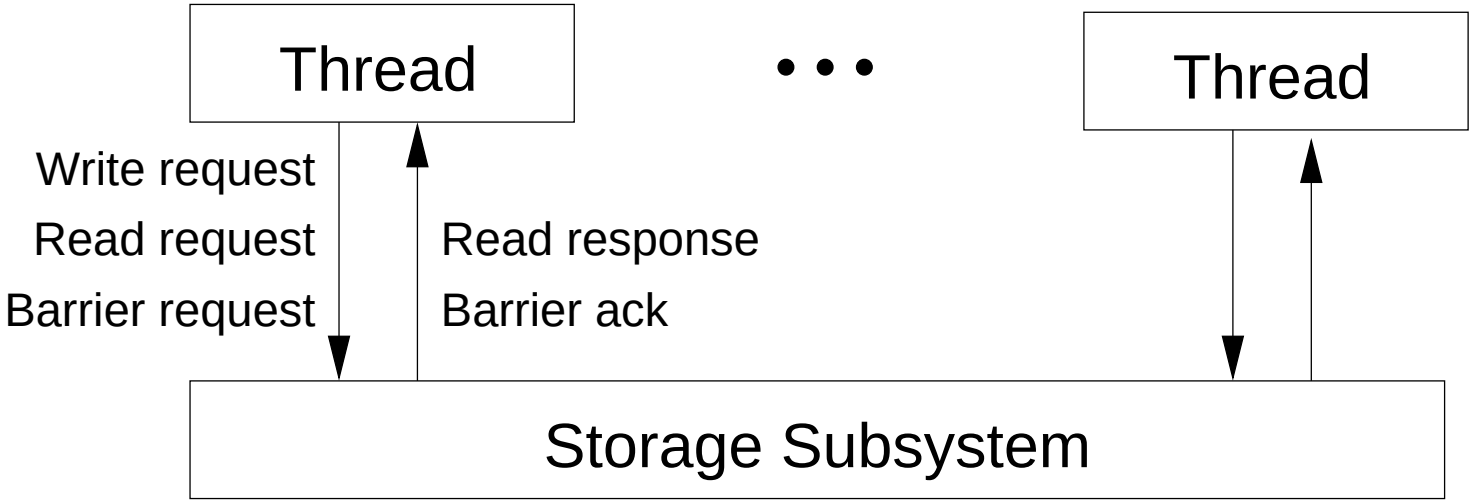
- programmer-visible out-of-order and speculative execution
- non-multi-copy-atomic storage subsystem

Similar but not identical to each other

Operational Models, Overview

Operational abstract-machine models:

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



Broadly corresponding to microarchitecture: to a first approximation this “thread” models the pipeline (and perhaps the L1 store queue); this “storage subsystem” models the remainder of the cache hierarchy and interconnect.

Features

- normal loads and stores (aligned, non-mixed-size, no self-modifying code)
- the (strong) barriers: sync (POWER) and dmb (ARM)
  (aka hwsync and dmb sy)
- dependencies and isync/isb

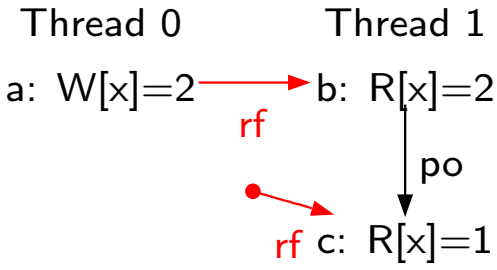
- weaker barriers: lwsync (POWER); dmb ld and dmb st (ARM)
- SC loads and stores: LDAR/STLR (ARM)
- atomic operations: load-linked/store conditional pairs. lwarx/stwcx (POWER), LDREX/STREX (ARM), ...
- misaligned and mixed-size accesses
- ISA semantics and ISA/concurrency integration

- exceptions and interrupts
- virtual memory
- other memory types (device memory, write-combining memory, ...)
- ...

Coherence

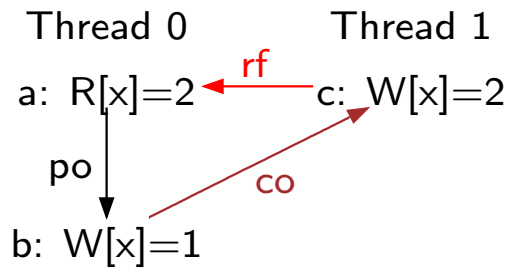
Reads and writes to each location in isolation behave SC

CoRR1: rf,po,fr forbidden



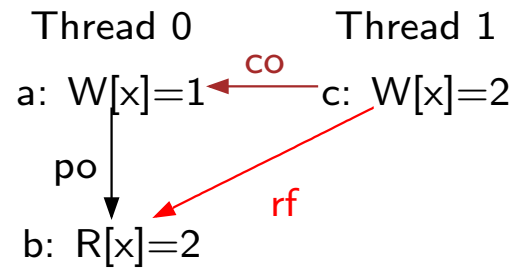
Test CoRR1

CoRW: rf,po,co forbidden



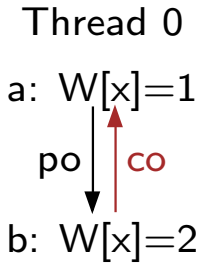
Test CoRW

CoWR: co,fr forbidden



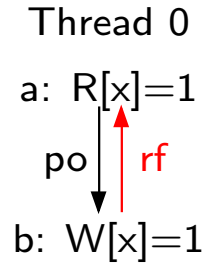
Test CoWR

CoWW: po,co forbidden



Test CoWW: Forbidden

CoRW1: po,rf forbidden



Test CoRW1: Forbidden

(these shapes are in some sense complete...)

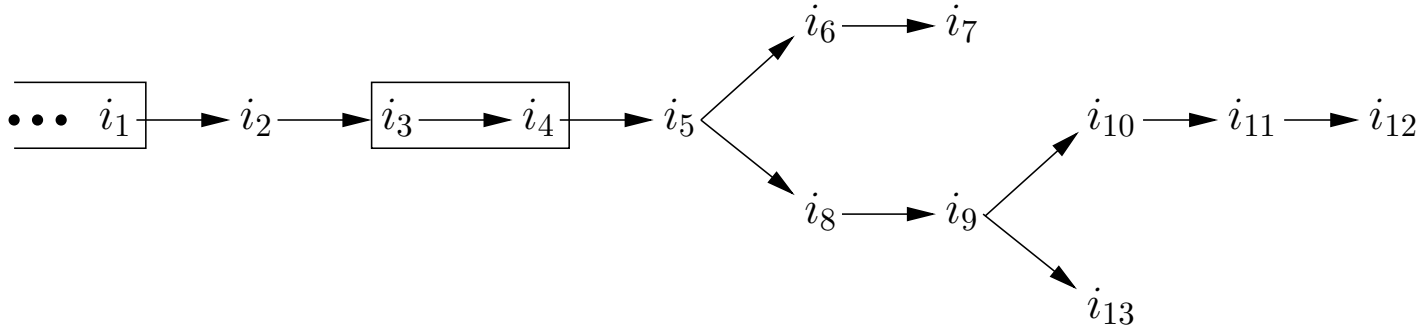
Maintaining Coherence in hardware

- cache protocol (MSI, MESI, MOESI, ...)
- more broadly, the interconnect design
- a bunch of other hazard checks in the pipeline
- ...

Pipeline Aspects: Basics

Thread Semantics

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

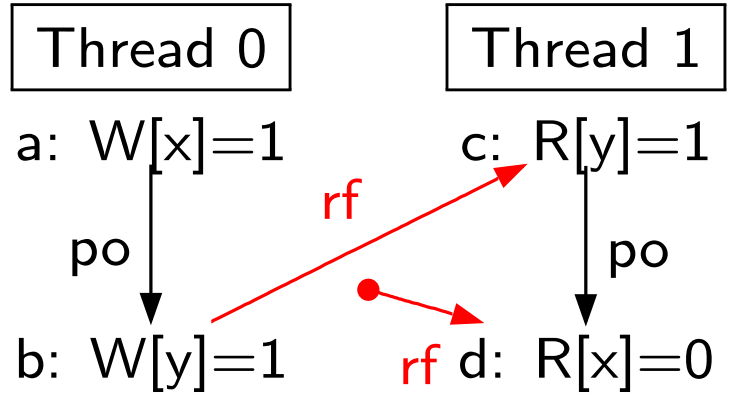


Microarchitecturally: modern pipelines typically do out-of-order execution and speculate past conditional branches

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	

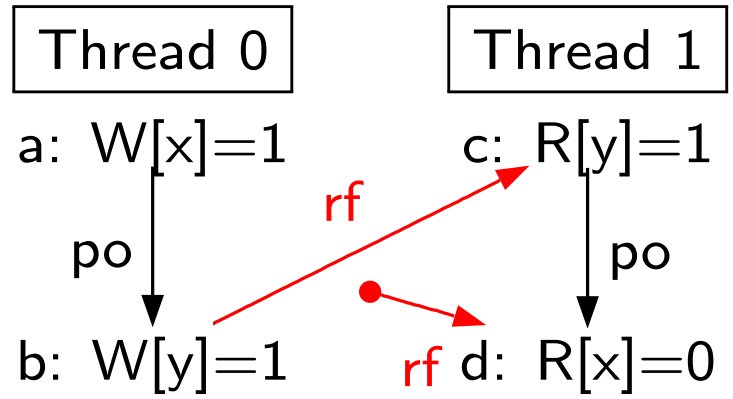


Test MP: Allowed

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	



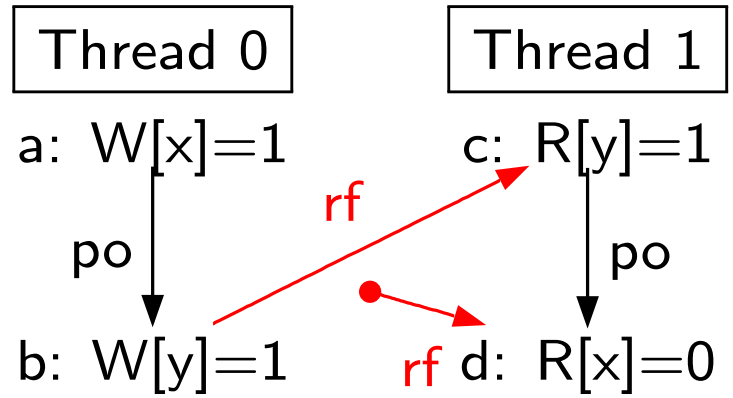
Test MP: Allowed

		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 ∧ y=0	
Allowed: 1:r1=1 ∧ 1:r2=0	



Test MP: Allowed

Microarchitecturally:

- pipeline: out-of-order execution of the writes
- pipeline: out-of-order execution of the reads
- storage subsystem: write *propagation* in either order

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 ∧ y=0	
Forbidden: 1:r1=1 ∧ 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 ∧ 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	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 ∧ 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 ∧ y=0	
Forbidden: 1:r1=1 ∧ 1:r2=0	

MP+dmbs ARM

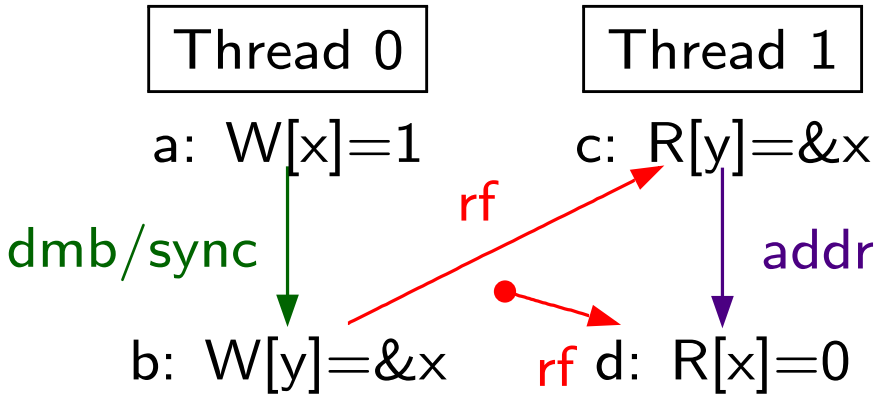
Thread 0	Thread 1
MOV R0,#1	LDR R0,[R3]
STR R0,[R2]	DMB
DMB	LDR R1,[R2]
MOV R1,#1	
STR R1,[R3]	
Initial state: 0:R2=x ∧ 0:R3=y ∧ 1:R2=x ∧ 1:R3=y	
Forbidden: 1:R0=1 ∧ 1:R1=0	

MP+syncs POWER

Thread 0	Thread 1
li r1,1	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 ∧ 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

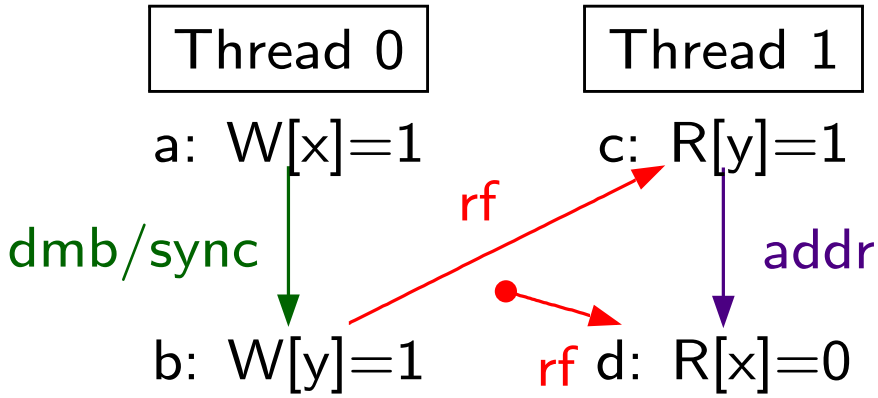
MP+dmb/sync+addr′ Pseudocode

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	

Microarchitecturally: the processor is not (in any programmer-visible way...) speculating the *value* used for the address of the second read.

Enforcing Order with Dependencies

POWER and ARM architecturally guarantee to respect address dependencies even if they are “false” or “artificial”:



Test MP+dmb/sync+addr: Forbidden

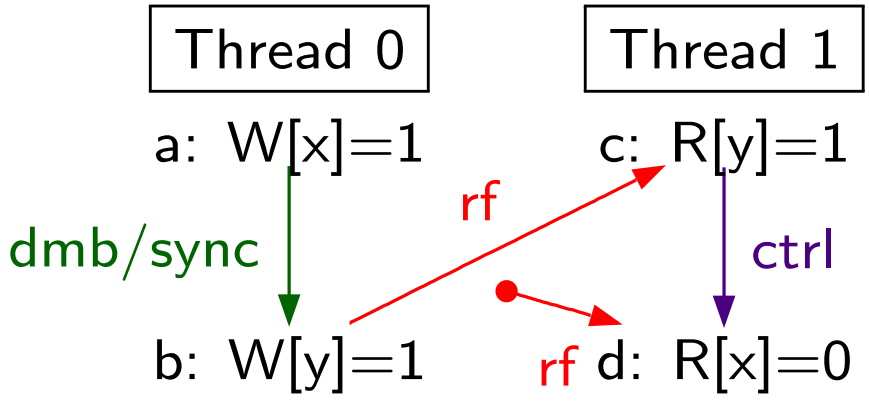
MP+dmb/sync+addr Pseudocode

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

NB: your compiler will not respect this!

Enforcing Order with Dependencies

Microarchitecturally: processors do speculate the outcomes of conditional branches, executing past them before they are resolved:



Test MP+dmb/sync+ctrl: Allowed

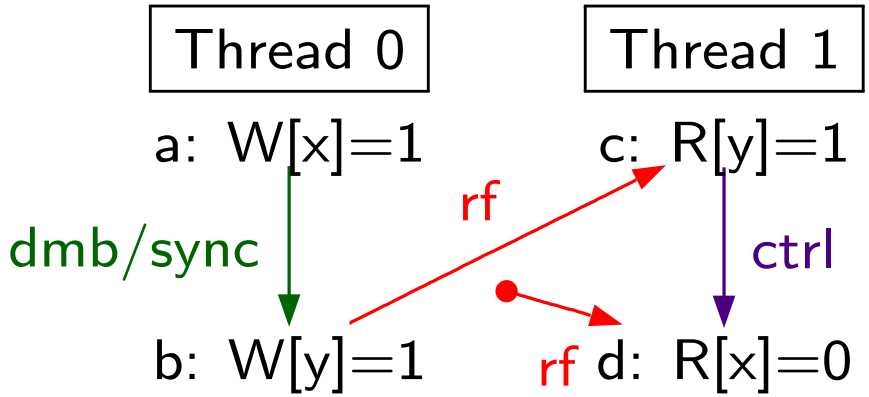
MP+dmb/sync+ctrl

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	

This is a read-to-read *control dependency*

Enforcing Order with Dependencies

Microarchitecturally: processors do speculate the outcomes of conditional branches, executing past them before they are resolved:



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	

Strengthen with ISB/isync instruction between branch and second read:

Thread-local read-to-read ordering is enforced by a conditional branch that is data-dependent on the first read, with an ISB/isync between the branch and the second read – call this a *control-isb/control-isync* dependency

Enforcing Order with Dependencies

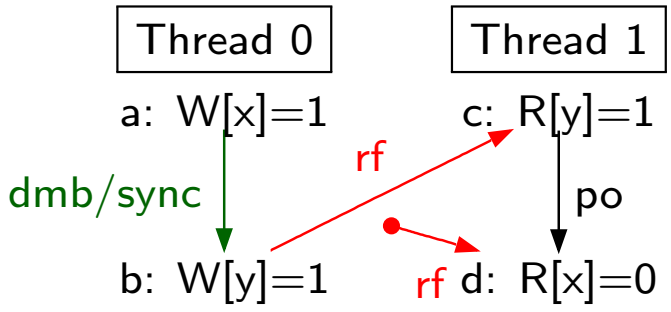
Read-to-Read: address and control-isb/control-isync dependencies respected; control dependencies *not* respected

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

(POWER: all whether natural or artificial. ARM: some debate about artificial data dependencies)

Pipeline Aspects: Further Subtleties

Programmer-visible shadow registers



Test MP+sync+rs (T1 reg reuse): Allowed

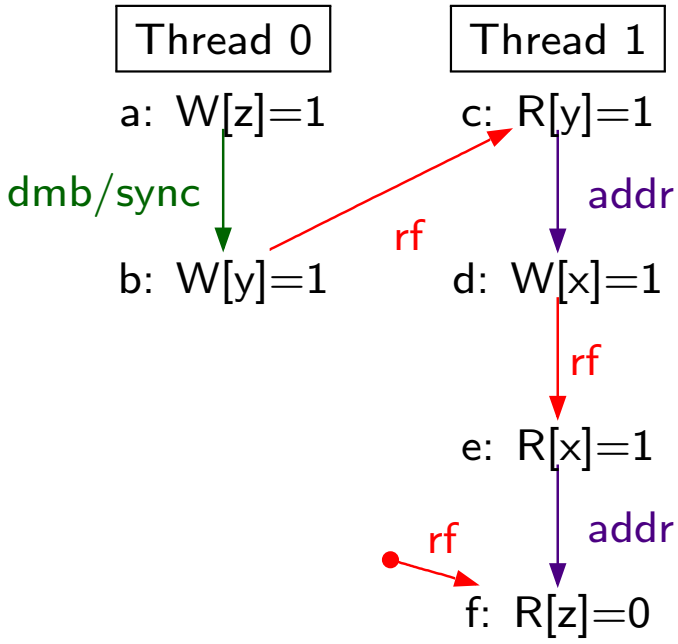
MP+dmb/sync+rs Pseudocode

Thread 0	Thread 1
x=1	r3=y
dmb/sync	r1=r3
y=1	r3 = x
Allowed: 1:r1=1 ∧ 1:r3=0	

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
LB+rs	Allow	0/3.7G	0/26G	0/898G	101k/3.9G	6.4k/89M	0/26G	60k/201M
MP+dmb/sync+rs	Allow	1.8k/3.0G	0/41G	29M/146G	9.0M/3.9G	1.2k/19M	11k/753M	549k/201M

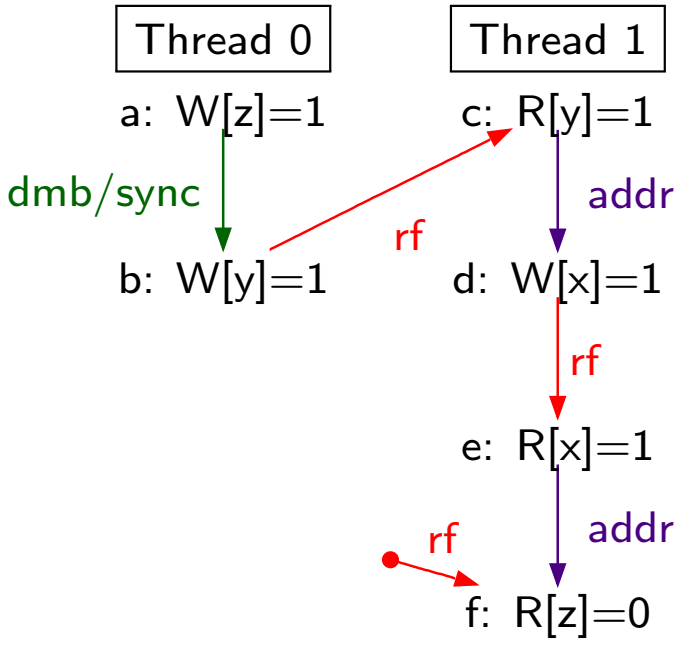
Reuse of the same architected register name does not enforce local reordering. Microarchitecturally: there are shadow registers and register renaming.

Pipeline write forwarding: PPOAA/PPOCA

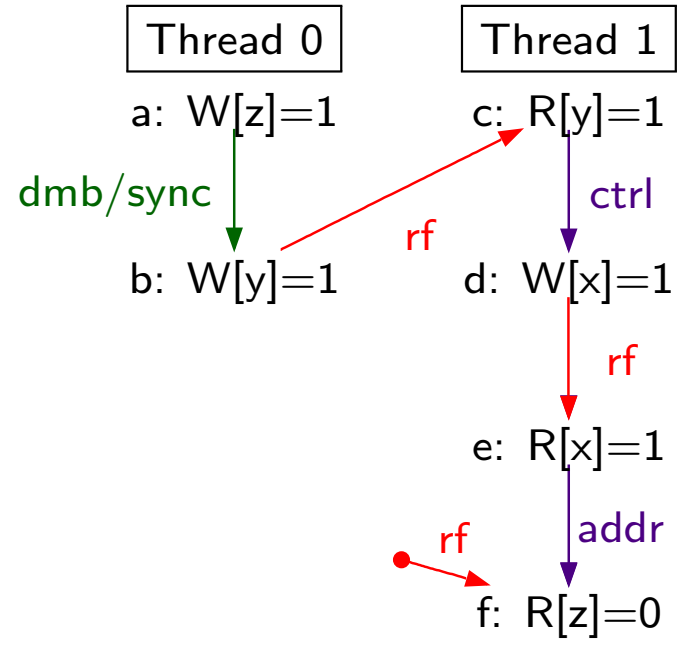


Test PPOAA: Forbidden

Pipeline write forwarding: PPOAA/PPOCA



Test PPOAA: Forbidden



Test PPOCA: Allowed

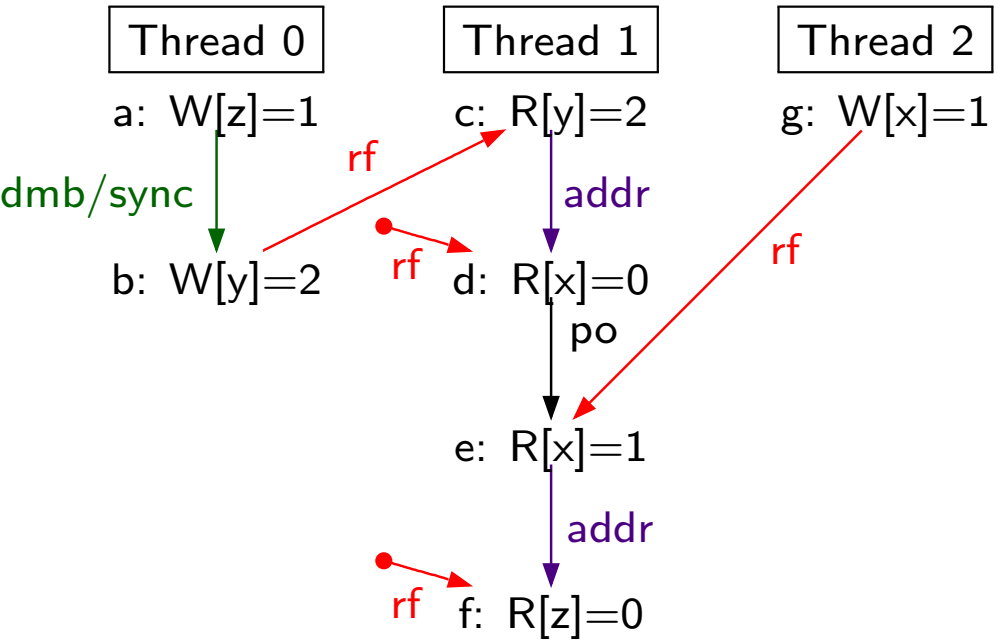
		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
PPOCA	Allow	1.1k/3.4G	0/49G	175k/157G	0/24G	0/39G	233/743M	0/2.2G
PPOAA	Forbid	0/3.4G	0/46G	0/209G	0/24G	0/39G	0/26G	0/2.2G

Writes on speculatively executed branches are not visible to other threads, but can be forwarded to po-later reads on the same thread.

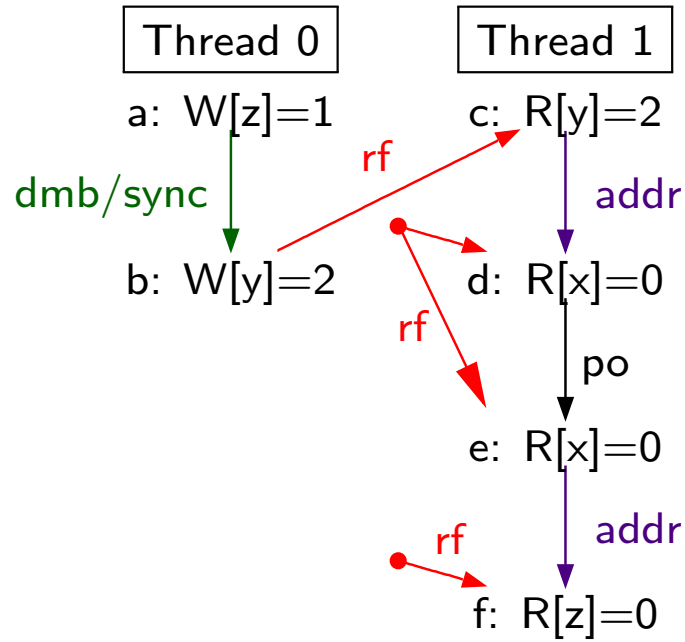
Microarchitecturally: they can be read from an L1 store queue

Aggressively out-of-order reads (RSW/RDW)

Coherence suggests reads from the same address must be satisified in program order, but if they read from the same write event, that's not true.



Test RDW: Forbidden

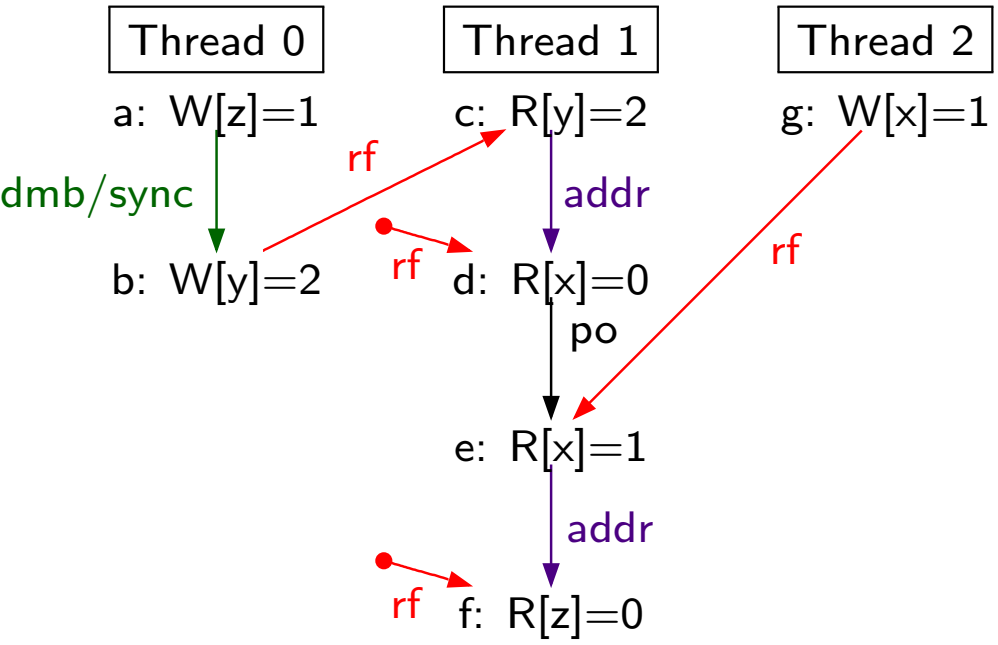


Test RSW: Allowed

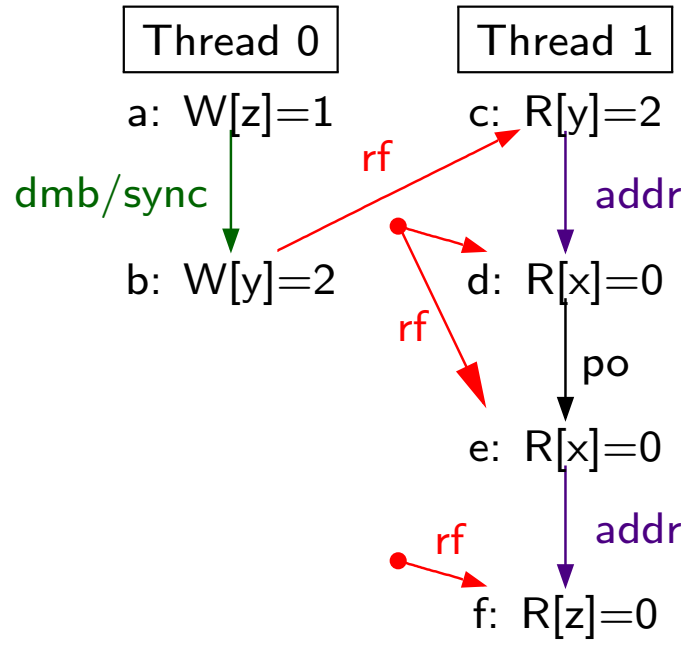
		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
RSW	Allow	1.3k/3.4G	0/33G	33M/144G	0/24G	0/39G	0/26G	0/2.2G
RDW	Forbid	0/1.7G	0/17G	0/125G	—	0/20G	—	—
RDWI	Allow	5.2k/3.0G	0/12G	1.3M/43G	0/24G	0/39G	0/26G	0/2.2G

Aggressively out-of-order reads (RSW/RDW)

Coherence suggests reads from the same address must be satisified in program order, but if they read from the same write event, that's not true.



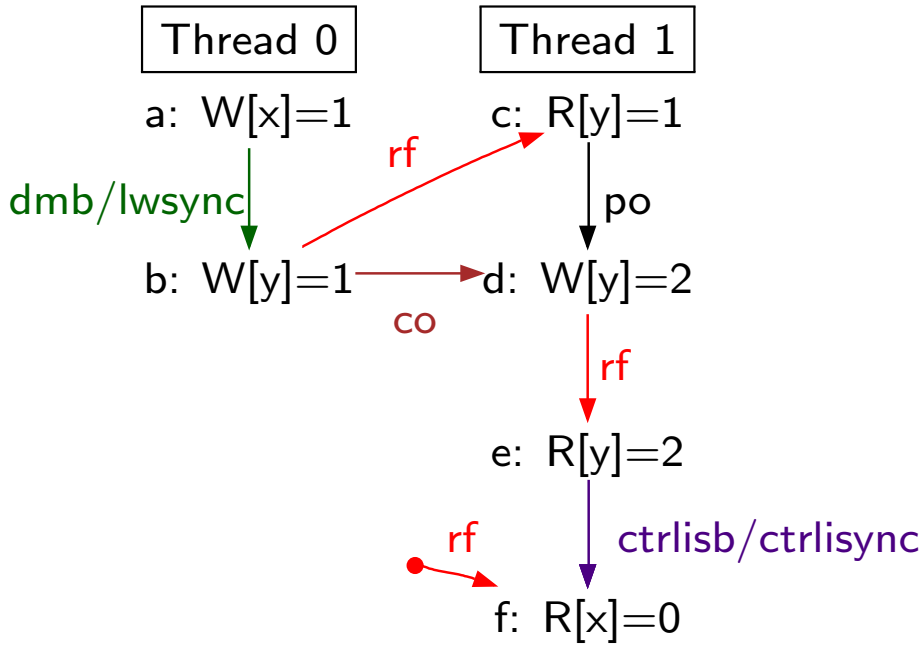
Test RDW: Forbidden



Test RSW: Allowed

Microarchitecturally: one can imagine the reads can in general be satisfied out-of-order, and the coherence hazard checking looks at whether the x *cache line* changes between the two reads.

Observable Read-request Buffering



Test MP+dmb/lwsync+fri-rfi-ctrlisb/ctrlisync

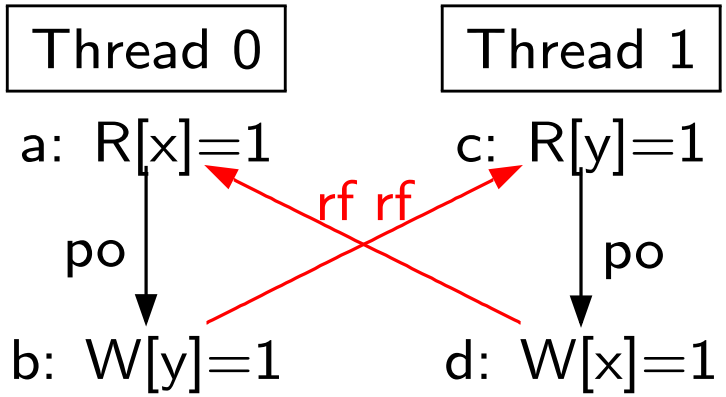
		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
MP+dmb/lwsync+fri-rfi-ctrlisb/isync	Allow	0/26G	0/6.6G	0/80G	0/26G	0/39G	7/1.6G	0/1.9G

PLDI11 POWER model: forbidden

POWER architectural intent: uncommitted

ARM: experimentally observed (on Qualcomm part) and not regarded as h/w bug

Load Buffering (LB)



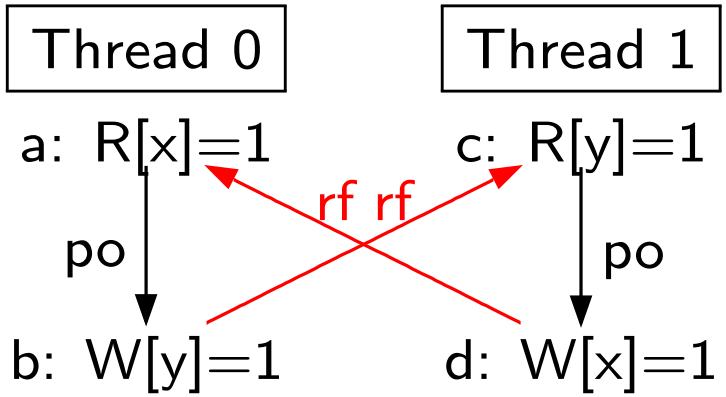
Test LB: Allowed

LB Pseudocode

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

Architecturally allowed on POWER and ARM

Load Buffering (LB)



Test LB: Allowed

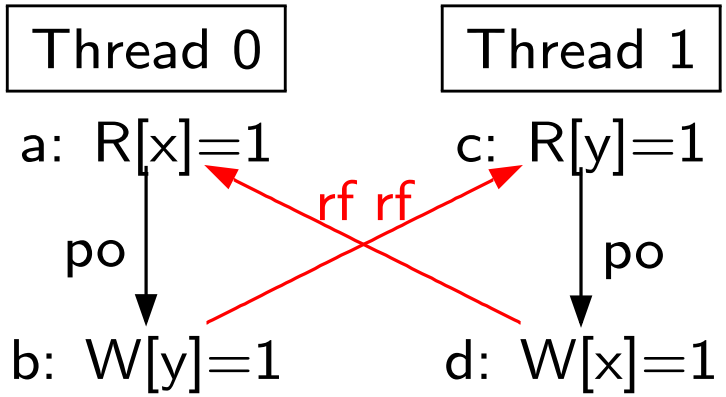
LB Pseudocode

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

Forbid 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

Load Buffering (LB)

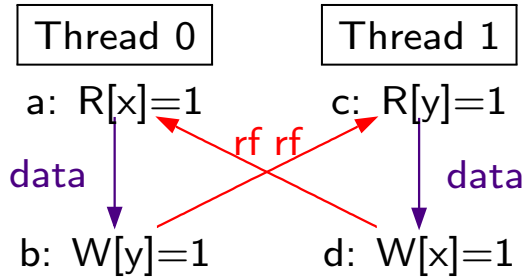


Test LB: Allowed

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	

LB+datas: thin-air values?

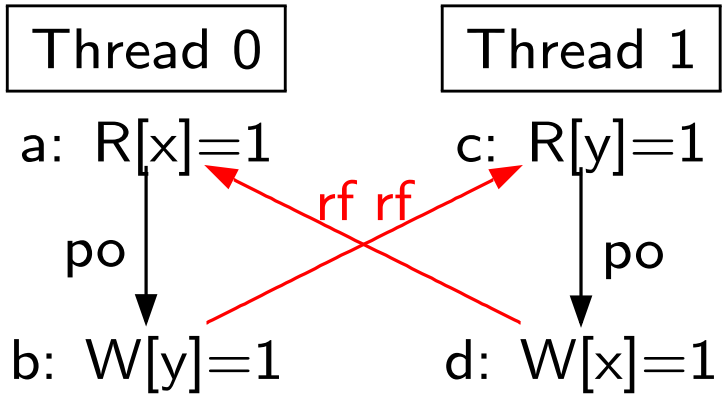


Test LB+datas: Forbidden

LB+datas Pseudocode

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

Load Buffering (LB)



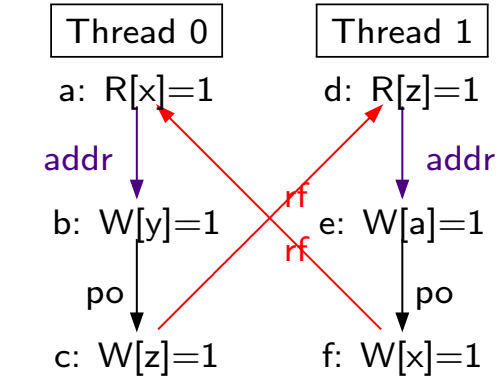
Test LB: Allowed

LB Pseudocode

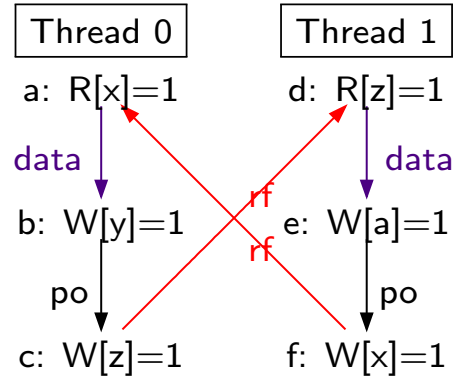
Thread 0	Thread 1
r1=x y=1	r2=y x=1
Initial state: x=0 $\wedge$ y=0	
Allowed: r1=1 $\wedge$ r2=1	

Microarchitecturally: simple out-of-order execution? read-request buffering? think about precise exceptions...

Might-access-same-address



Test LB+addrs+WW: Forbidden



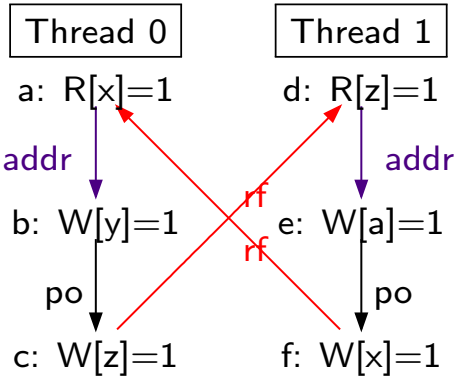
Test LB+datas+WW: Allowed

Address and data dependencies to a write both prevent the write being visible to other threads before the dependent value is fixed. But there is a more sutble effect that distinguishes them: the existence of a address dependency to a write might mean that another program-order-later write cannot proceed until it is known that the first write is not to the same address, whereas the existence of a data dependency to a write has no such effect on program-order-later writes that are statically known to be to different addresses.

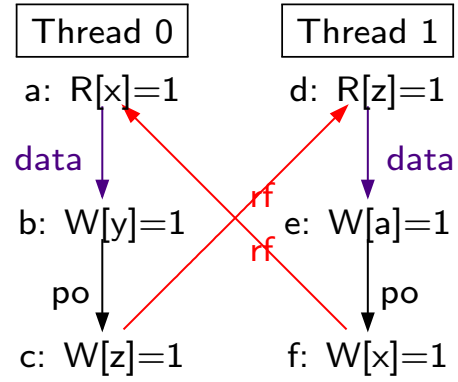
Does it matter?

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
LB+addrs+WW	Forbid	0/30G	0/8.7G	0/208G	0/16G	0/23G	0/18G	0/2.1G
LB+datas+WW	Allow	0/30G	0/9.2G	0/208G	15k/6.3G	224/854M	0/18G	23/1.9G
LB+addrs+RW	Forbid	0/3.6G	0/6.0G	0/128G	0/13G	0/23G	0/16G	—

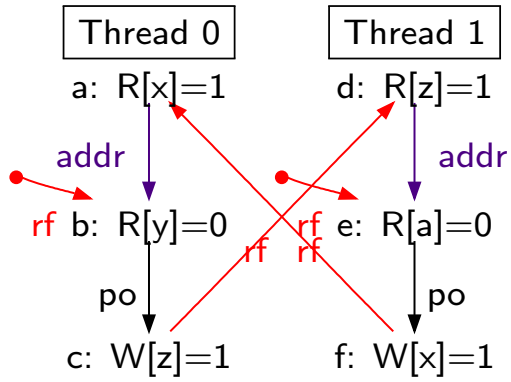
Might-access-same-address



Test LB+addrs+WW: Forbidden



Test LB+datas+WW: Allowed



Test LB+addrs+RW: Forbidden

		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
LB+addrs+WW	Forbid	0/30G	0/8.7G	0/208G	0/16G	0/23G	0/18G	0/2.1G
LB+datas+WW	Allow	0/30G	0/9.2G	0/208G	15k/6.3G	224/854M	0/18G	23/1.9G
LB+addrs+RW	Forbid	0/3.6G	0/6.0G	0/128G	0/13G	0/23G	0/16G	—

Storage Subsystem Aspects

(multi-copy atomicity and cumulative barriers)

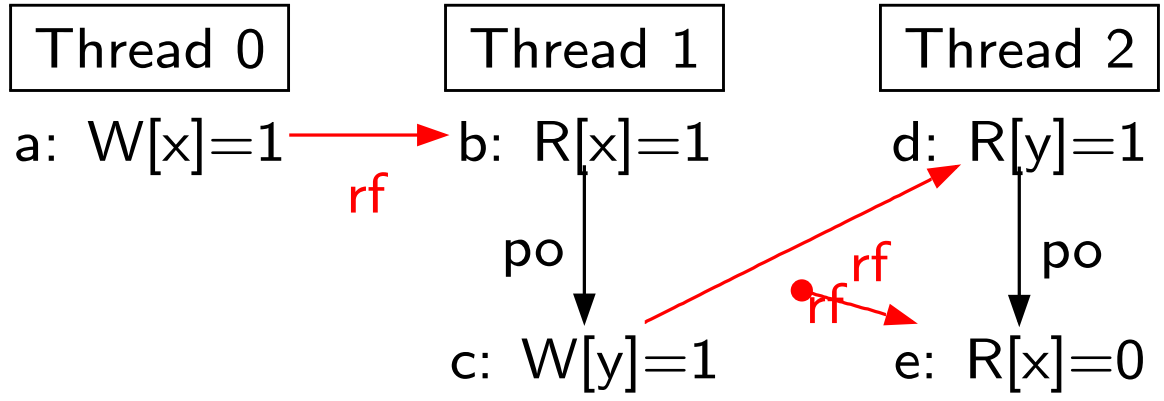
Things get more interesting with more than two hardware threads....

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



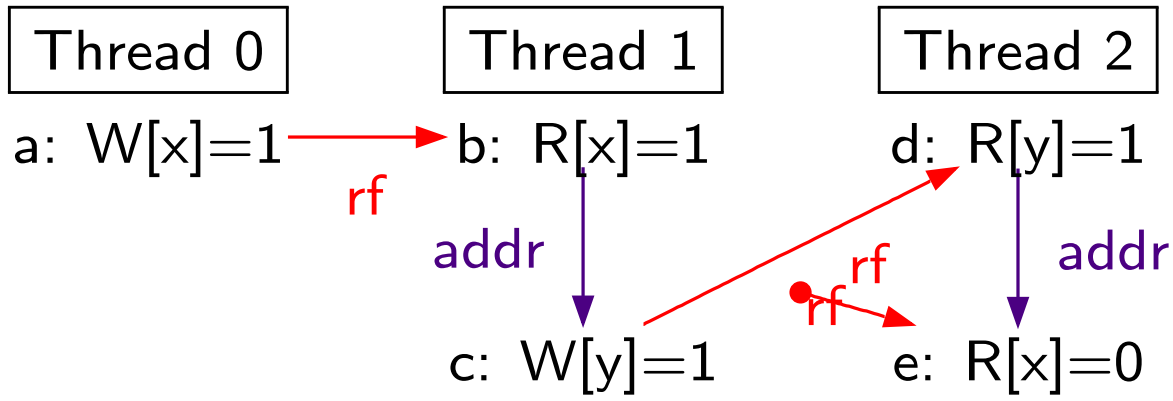
Test WRC: Allowed

WRC Pseudocode

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

That's allowed just by thread-local reordering, so this tells us nothing. Add address dependencies....

Iterated Message Passing and Cumulative Barriers

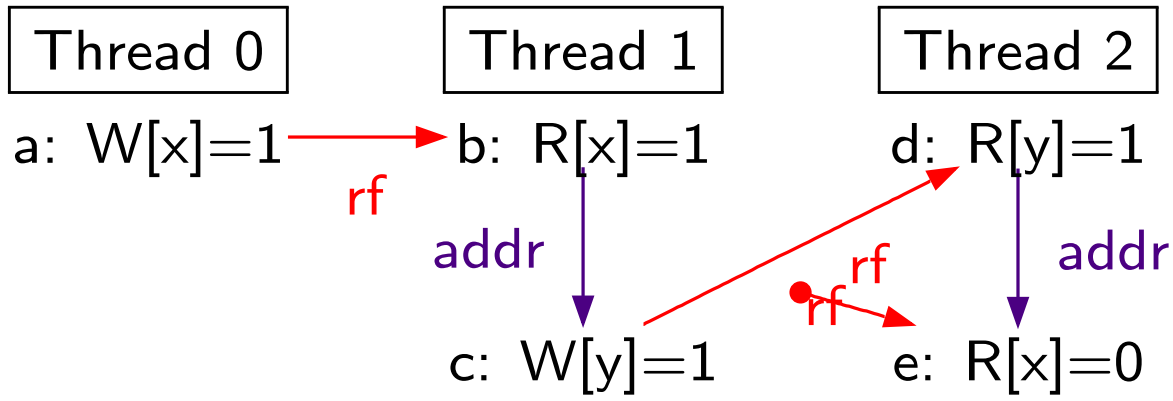


Test WRC+addrs: Allowed

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		

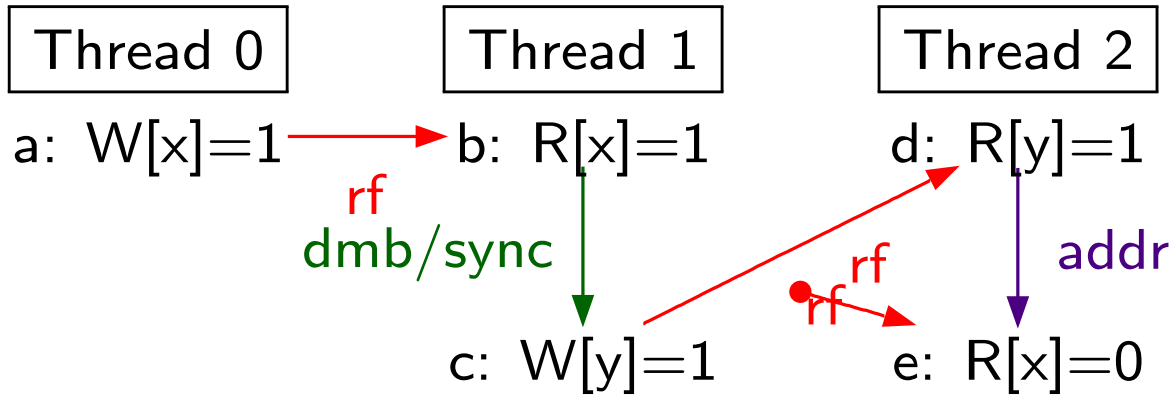
Iterated Message Passing and Cumulative Barriers



Test WRC+addrs: Allowed

ARM and POWER are not *multi-copy-atomic*: the fact that a write has become visible to some other thread does not mean it is visible to all other threads.

Iterated Message Passing and Cumulative Barriers

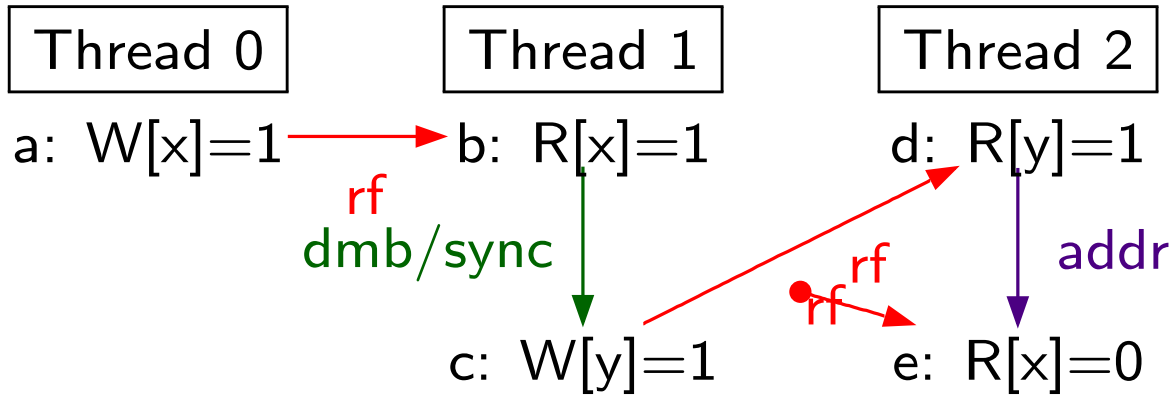


Test WRC+dmb/sync+addr: Forbidden

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

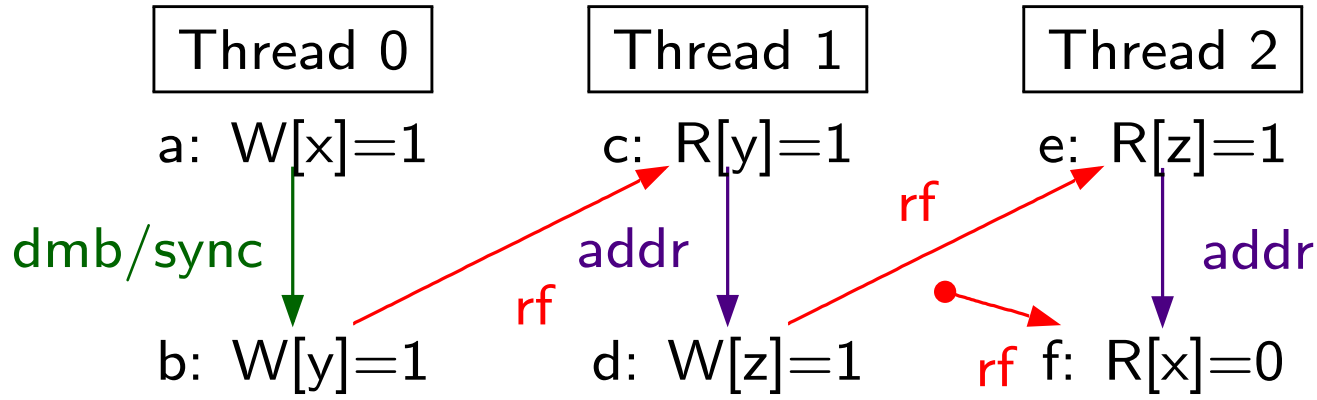


Test WRC+dmb/sync+addr: Forbidden

A dmb/sync keeps writes by the same thread (before and after the barrier) ordered, as far as any single other thread is concerned.

But they also keep any writes propagated to the barrier thread (before the barrier) ordered before writes (by this thread) after the barrier, as far as any other single thread is concerned. A *cumulativity* property. Here (a,c) are ordered, as seen by Thread 2. Microarchitecturally: ...

Iterated Message Passing and Cumulative Barriers



Test ISA2+dmb/sync+addr+addr: Forbidden

And also (a,d) are ordered, w.r.t. visibility by Thread 2.

Explain in terms of write and barrier propagation:

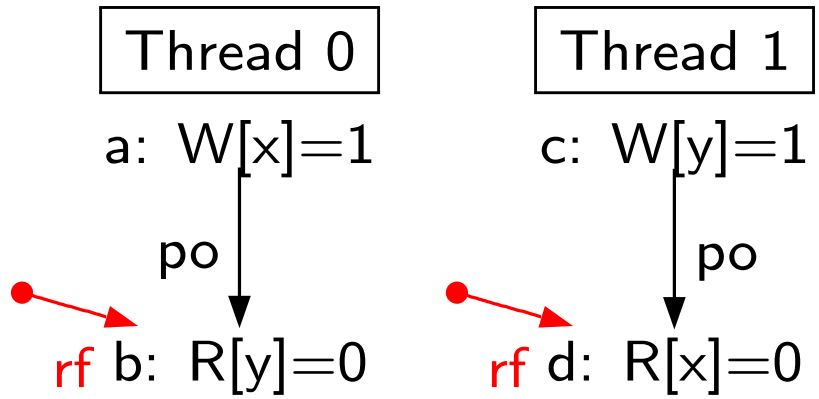
- Writes (a) and (b) are separated by the barrier
- ...so for Thread 1 to read from (b), both (a) and the barrier have to propagate there, in that order
- But now (a) and (d) are separated by the barrier
- ...so before Thread 2 can read from (d), (a) (and the barrier) has to propagate there too
- and hence (f) has to read from (a), instead of the initial state.

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

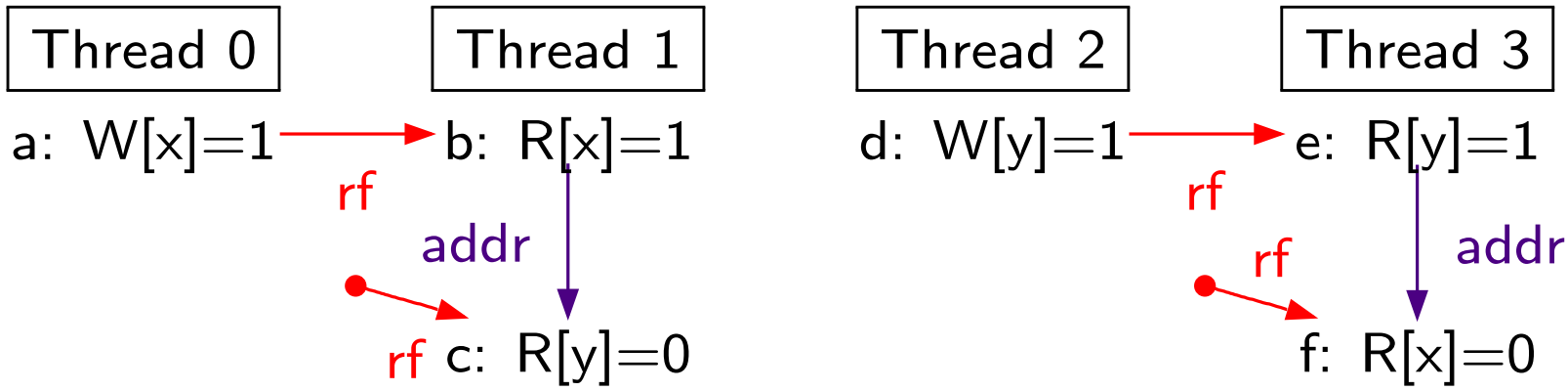
Another illustration of non-multi-copy-atomic behaviour: take SB



Test SB: Allowed

and pull out the initial writes to two other threads (and add address dependencies to prevent local reordering)

Independent Reads of Independent Writes



Test IRIW+addrs: Allowed

IRIW+addrs Pseudocode

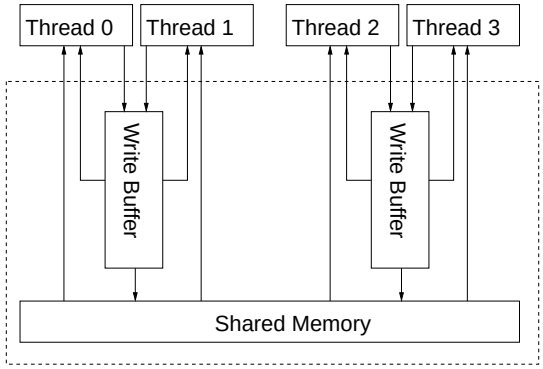
Thread 0	Thread 1	Thread 2	Thread 3
x=1	r1=x r2=*(&y+r1-r1)	y=1	r3=y r4=*(&x+r3-r3)
Initial state: x=0 ∧ y=0 ∧ z=0			
Allowed: 1:r1=1 ∧ 1:r2=0 ∧ 3:r3=1 ∧ 3:r4=0			

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

Independent Reads of Independent Writes

Microarchitecturally:

- Could arise from hierarchical store buffers



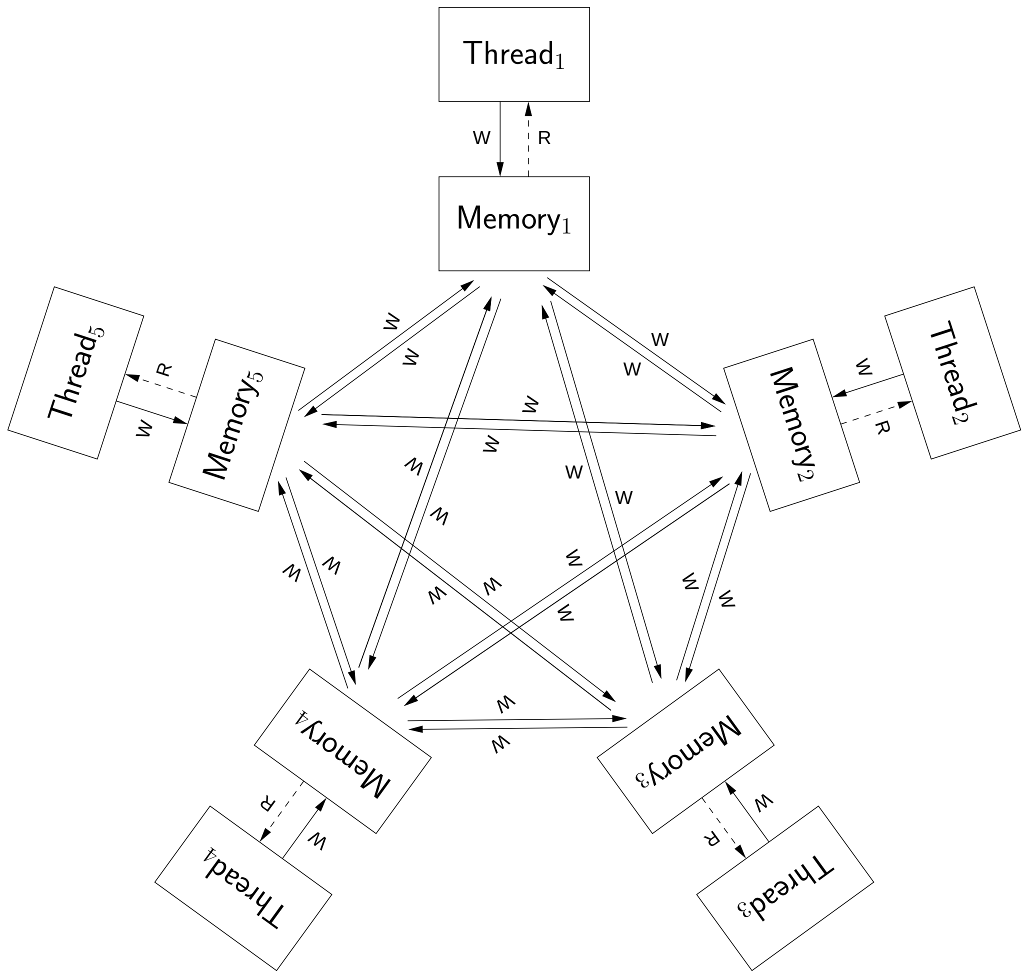
- Or just from the cache protocol

(is there a test that distinguishes?)

Storage Subsystem Semantics

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

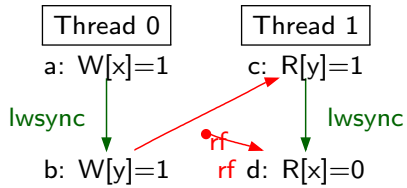
No global memory



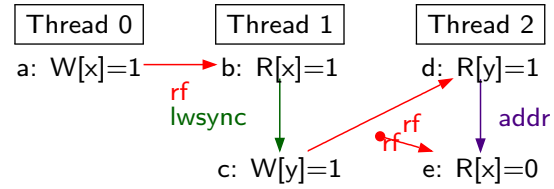
Weaker Barriers and Stronger Operations

lwsync (POWER)

- Cheaper than sync (aka hwsync).
- Locally orders RR, WR, and WW pairs, but not WR
- Similar cumulativity properties as sync, so suffices for message-passing (MP, WRC, ISA2).

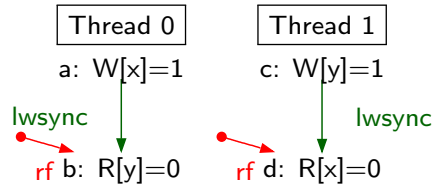


Test MP+lwsyncs: Forbidden

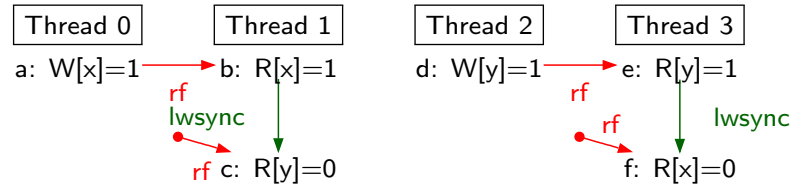


Test WRC+lwsync+addr: Forbidden

- Does *not* suffice to exclude SB, IRIW



Test SB+lwsyncs: Allowed

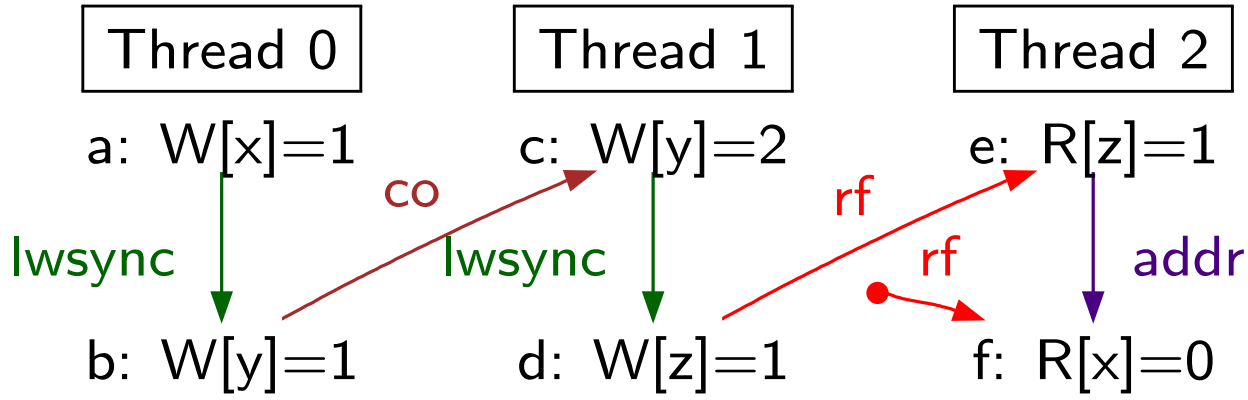


Test IRIW+lwsyncs: Allowed

- Model: think of sync as blocking until all previous (or previously seen) writes have propagated *everywhere*, while lwsync doesn't.

Coherence and lwsync (or not)

The transitive closure of coherence and lwsync edges does not guarantee ordering:



Test Z6.3+lwsync+lwsync+addr: Allowed

The fact that the storage subsystem commits to b before c in the coherence order has no effect on the order in which writes a and d propagate to Thread 2. Thread 1 does not read from either Thread 0 write, so they need not be sent to Thread 1, so no cumulativity is in play. In other words, coherence edges do not bring writes into the "Group A" of a POWER barrier.
Microarchitecturally: the coherence choice may be made later
Contrast with ISA2+lwsync+addr+addr

dmb st and dmb ld (ARM)

Omit for now...

SC loads and stores LDAR/STLR (ARM)

ISA design choice: strength in barriers or in labelled operations?

NB: ARM call these load-acquire and store-release, but this is confusing terminology: they are stronger than the usual release/acquire notions. They guarantee SC — at least when observed with these operations.

Operational Model (POWER)

Basic Question

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

We've built a POWER operational model...

...by a long process of

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

(Also ARM operational models – Flowing and POP – and various axiomatic models; see refs later)

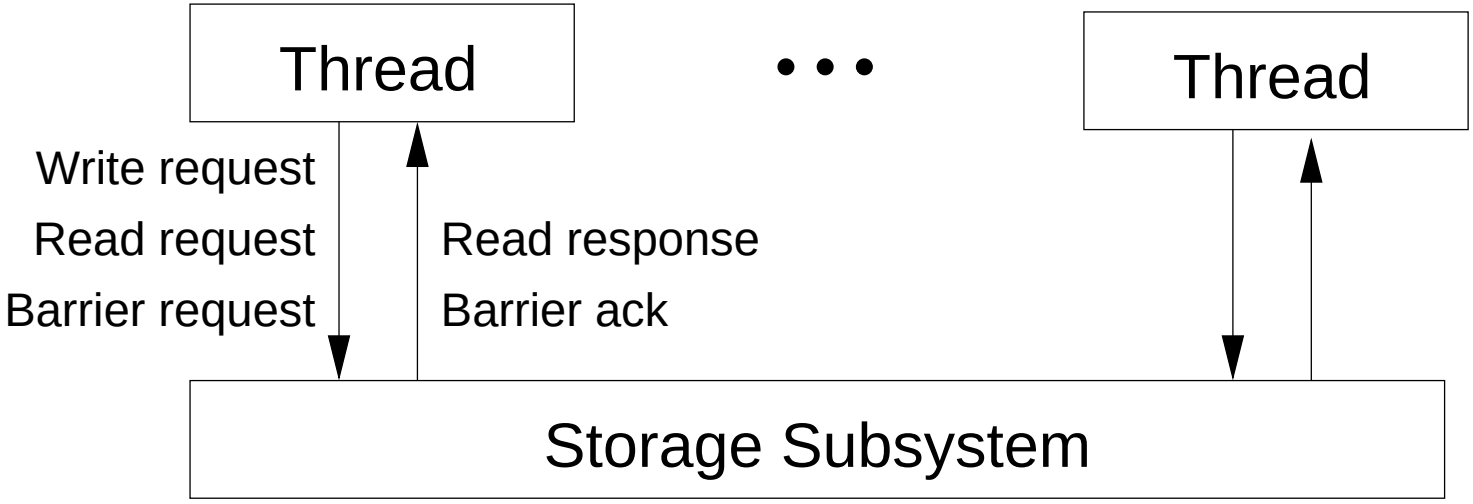
Basic Idea

With a microarchitectural flavour (so can discuss with architects and they can relate to their implementations)

But as abstract as possible: abstracting from store buffers, cache hierarchies, cache protocols, etc.

Aiming to be architecturally sound and complete: allowing exactly all the behaviour they intend to be allowed

Aiming to be sound w.r.t. current hardware implementations (modulo hardware bugs)


Thread
• • •
Thread
Write request
Read request
Barrier request
Read response
Barrier ack
Storage Subsystem

Storage Subsystem: Coherence by Fiat

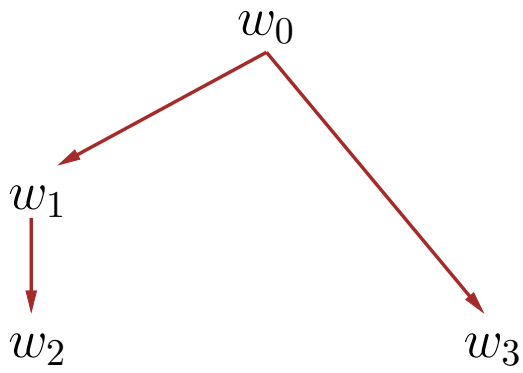
Suppose the storage subsystem has seen 4 writes to x:

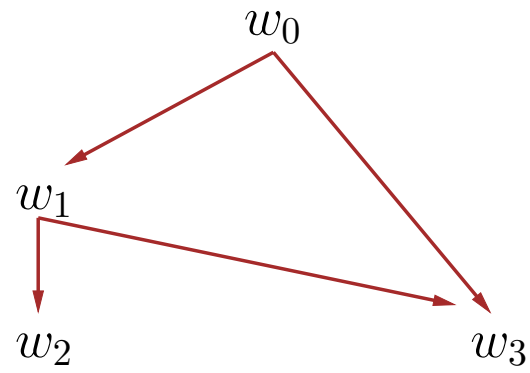
Suppose just $[w_1]$ has propagated to tid and then tid reads x.

- 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

Storage Subsystem: 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/

Systematic Test Families

Periodic table

`www.cl.cam.ac.uk/users/pes20/ppc-supplemental/poster1.pdf`

Systematic arrangement of small test shapes:

critical cycles of po, rf, co, and fr edges
(recall rf from initial state $=$ fr from co-first write)

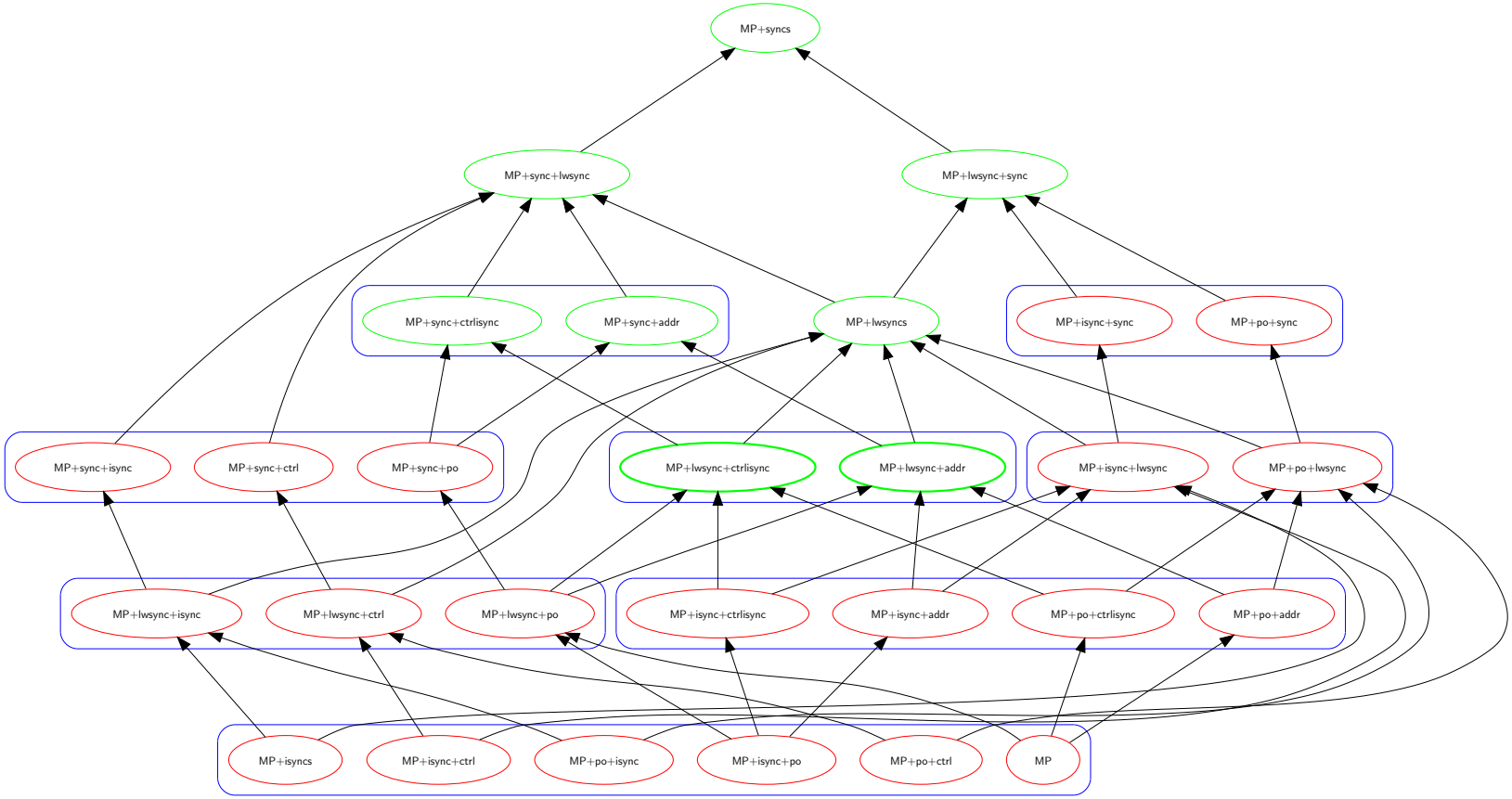
- the six 4-edge 2-thread 2-location tests (MP, S; SB, R, 2+2W; LB)
- 5- and 6-edge extensions pulling writes out along new rf edges (including WRC, IRIW, WRC)
- the ten 6-edge 3-thread tests (including ISA2, Z6.3)
- the five minimal coherence tests
- a few ad hoc tests

Minimal Strengthenings for a Test Shape

For each shape, consider the weakest replacements of po edges by dependencies or barriers that forbid the non-SC behaviour, e.g. for MP:

$$\begin{aligned} \text{RRdep} &::= \text{addr} \mid \text{ctrlisb/ctrlisync} \\ \text{RWdep} &::= \text{addr} \mid \text{data} \mid \text{ctrl} \mid \text{ctrlisb/ctrlisync} \end{aligned}$$

po $<$ {RRdep,RWdep} $<$ lwsync $<$ dmb/sync (ignoring “might”)



Atomic operations:
lwarx/stwcx and LDREX/STREX

Load-reserve/Store-conditional

aka Load-linked/Store-conditional

Analogue of x86 LOCK'd INC etc. and CMPXCHG (CAS), but RISC-friendly

lwarx/LDREX atomically (a) loads, and (b) creates a reservation for this "storage granule" (POWER terminology: architectural abstraction of implementation "cache line")

stwcx/STREX atomically (a) stores and (b) sets a flag, *if* the storage granule hasn't been written to by any thread in the meantime

Can be used to implement CAS, atomic add, spinlocks, . . .

Universal (like CAS) [Herlihy'93] (and no ABA problem)

Atomic addition using lwarx/stwcx

Atomic Addition

```
loop:
  lwarx r, d
  add r,v,r
  stwcx r, d
  bne loop
```

- Informally, stwcx succeeds only if no other write to the same address since last lwarx, setting a flag iff it succeeds
- (though it may spontaneously fail)

What *is* no write since . . . ?

- In machine time?
  - Neither necessary, nor sufficient

- Microarchitecturally (simplified): if cache-line ownership not lost since last `lwarx`

  (but we don't want to model the microarchitecture...)

Modeling “not lost since”

- Abstractly: ownership chain modeled by building up coherence order
- Coherence: order relating stores to the same location (eventually linear)

- A `stwcx` succeeds only if it is (or at least, if it can become) coherence-next-to the write read from by `lwarx`
- . . . and no other write can later come in between

Modeling “not lost since”

- Abstractly: ownership chain modeled by building up coherence order
- Coherence: order relating stores to the same location (eventually linear)

- A `stwcx` succeeds only if it is (or at least, if it can become) coherence-next-to the write read from by `lwarx`
- . . . and no other write can later come in between

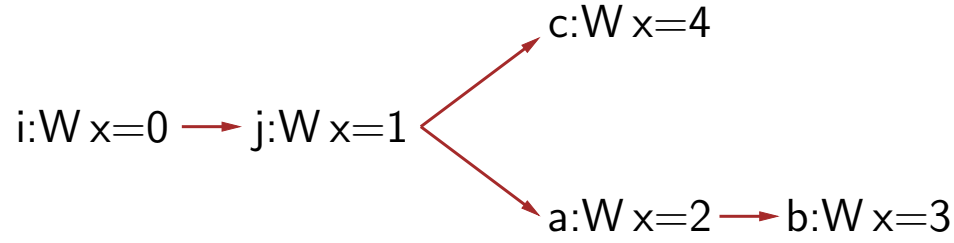
- Isolate key concept: write reaching coherence point —
  - coherence is linear below this write, and no new edges will be added below

Coherence points and a successful stwcx

Atomic Addition

```
loop:
  lwarx r, x
  add r,3,r
  stwcx r, x
  bne loop
```

Coherence order for x:



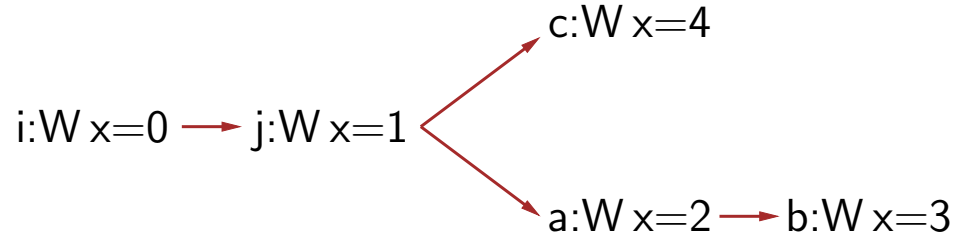
Suppose `lwarx` reads from the "a:W x:2"

Coherence points and a successful stwcx

Atomic Addition

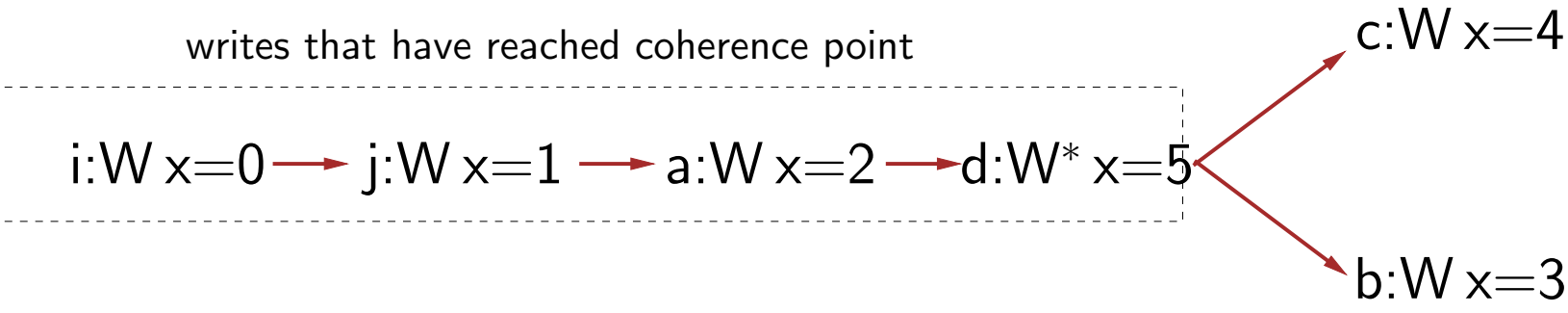
```
loop:
  lwarx r, x
  add r,3,r
  stwcx r, x
  bne loop
```

Coherence order for x:



Suppose `lwarx` reads from the "a:W x:2"

`stwcx` can succeed if this becomes possible:



Warning: `stwcx` can fail spuriously

Load-reserve/store-conditional and ordering

- Same-thread load-reserve/store-conditionals ordered by program order
  - If all memory accesses are l-r/s-c sequences
  - Then: only SC behaviour

- But . . . normal loads/stores (to different addresses) not ordered; the l-r/s-c do not act as a barrier
  - Confusion here led to Linux bug
  - . . . bad barrier placement in atomic-add-return

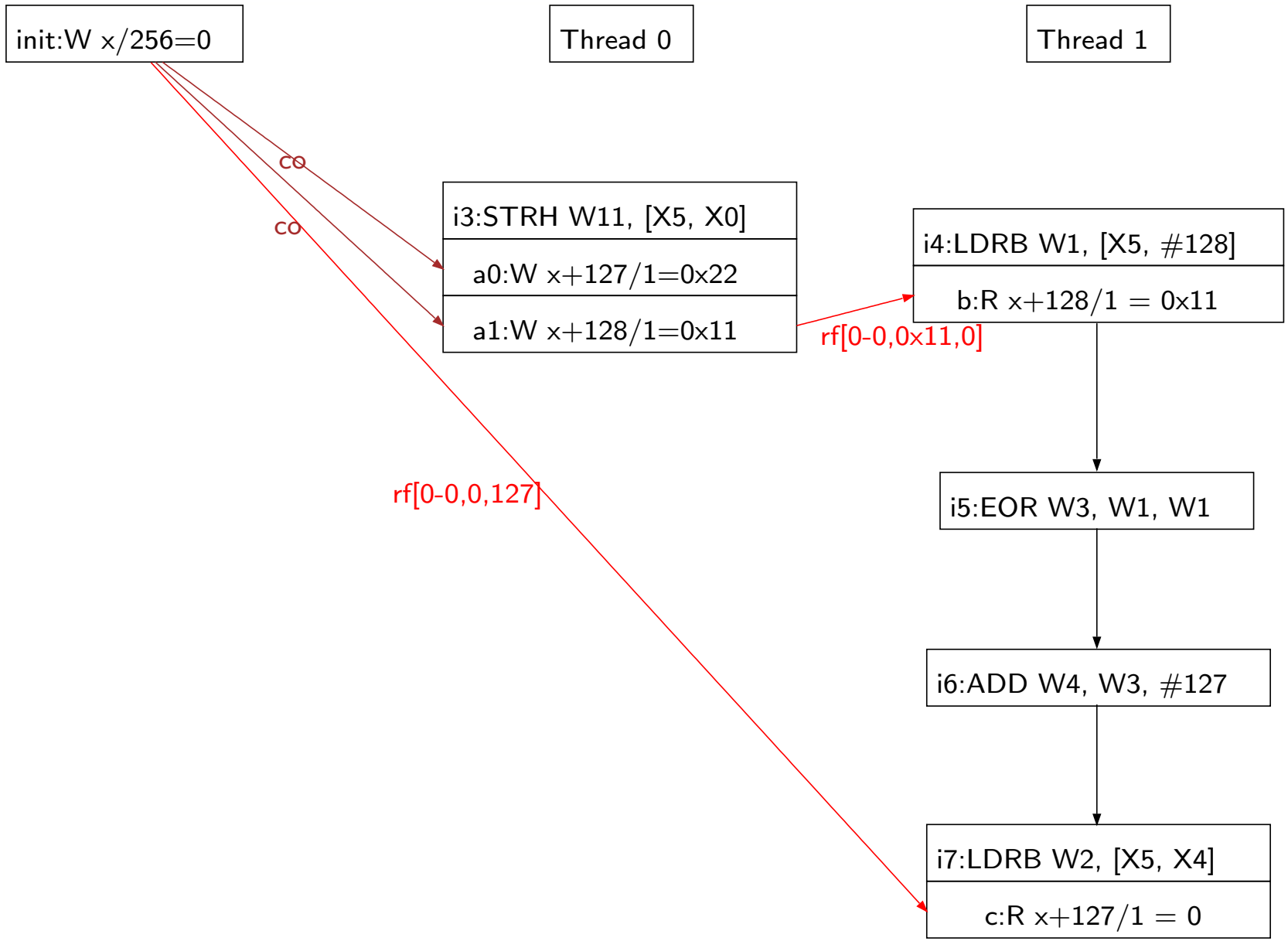
Misaligned and mixed-size accesses

Each architecture guarantees that certain combinations of access size and alignment will be indivisible (typically 2^n-size 2^n-aligned for some particular n's). [*"single-copy atomicity"*]

Others may, architecturally, be split into multiple byte-size accesses, though implementations typically split less.

Can the bytes of the 2-byte write of a STRH, if misaligned 1 byte off a cache-line boundary, be separately propagated to another thread?

```
AArch64 MP+misaligned2+127+addr
{
  uint8_t x[256];  (* two cache lines *)
  0:X5=x;
  0:X0=127;
  0:X11=0x1122;
  1:X5=x;
}

P0                                                | P1                                                ;
STRH W11,[X5,X0] (* *(&x+127)=(0x22,0x11) *)      | LDRB W1,[X5,#128] (* W1 = *(&x+128)    *) ;
                                                  | EOR  W3,W1,W1     (* W3 = W1 xor W1    *) ;
                                                  | ADD  W4,W3,#127                            ;
                                                  | LDRB W2,[X5,X4]   (* W2 = *(&x+127+W3) *) ;
exists (1:X1=0x11 /\ 1:X2=0)
```



Test MP+misaligned2+127+addr

Testing alignments w.r.t. a cache line

Test	flowing	pop	LG-H955
`MP+misaligned2+0+addr.litmus`	forbidden	forbidden	0/224M
`MP+misaligned2+1+addr.litmus`	allowed	allowed	0/20M
`MP+misaligned2+3+addr.litmus`	allowed	allowed	0/20M
`MP+misaligned2+7+addr.litmus`	allowed	allowed	0/220M
`MP+misaligned2+15+addr.litmus`	allowed	allowed	0/220M
`MP+misaligned2+127+addr.litmus`	allowed	allowed	20/222M
`MP+misaligned8+124+addr.litmus`	interactive	allowed	21/80M

LG-H955 phone: Snapdragon 810, Cortex-A57/A53

More mixed-size questions

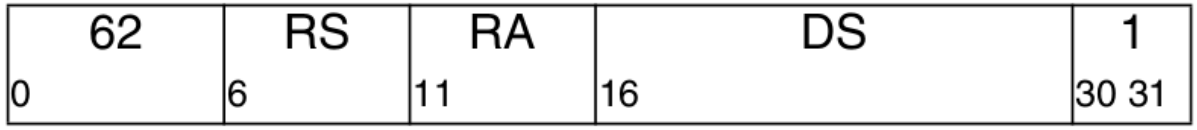
- splitting misaligned reads
- overlapping atomic writes
- footprint topology and coherence per-write or per-byte
- coherence: local reordering of disjoint reads
- coherence: propagation of non-coherence-superseded write slices
- forwarding from uncommitted writes
- dependency granularity via parts of system registers
- dependencies via load/store writeback register
- speculation of LR register valeus
- load/store multiple
- computed register footprints
- ARM conditional instructions

ISA semantics and ISA/concurrency integration

What does an ISA look like?

Store Doubleword with Update DS-form

stdu RS,DS(RA)




```
EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA
```

Let the effective address (EA) be the sum (RA)+ (DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None

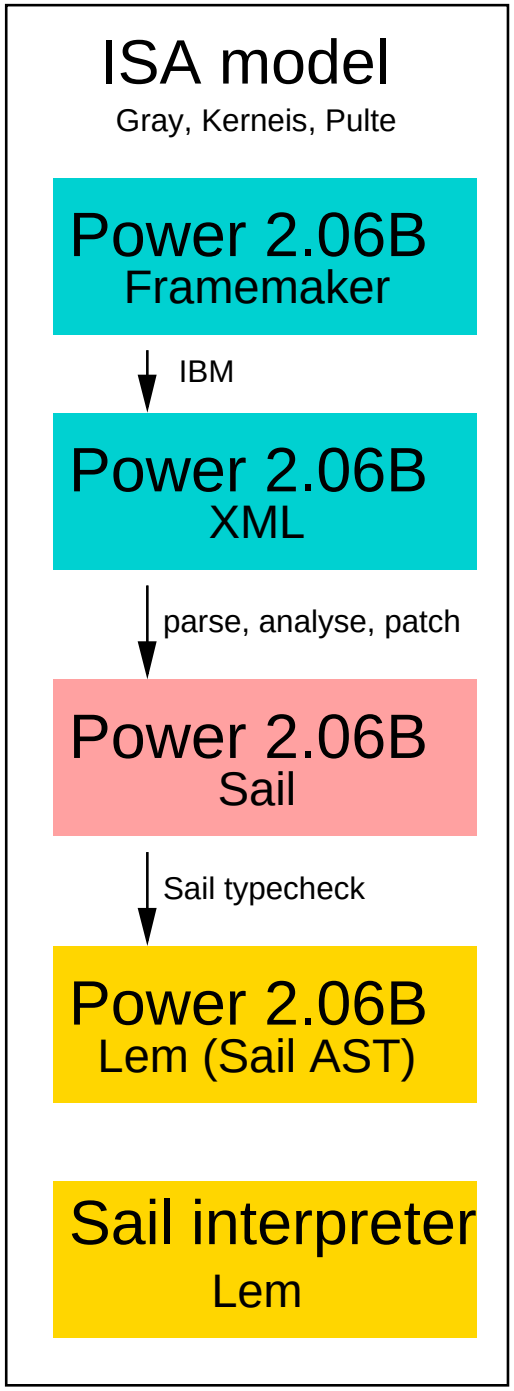
Problem 1: Scale

100s of instructions, some fiddly

changing (slowly) over time

want to maintain clear connection to vendor docs

want engineer-accessibility



Store Doubleword with Update DS-form

stdu RS,DS(RA)

62	RS	RA	DS	1
0	6	11	16	30 31

```
EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA
```

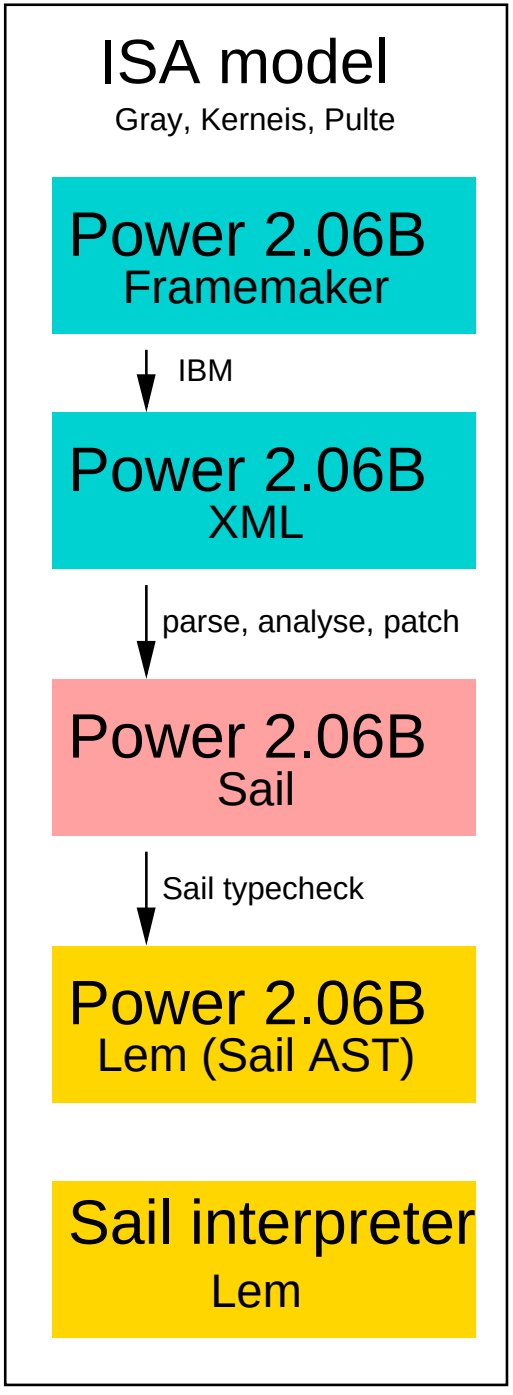
Let the effective address (EA) be the sum (RA)+ (DS||0b00). (RS) is stored into the doubleword in storage addressed by EA.

EA is placed into register RA.

If RA=0, the instruction form is invalid.

Special Registers Altered:

None



Store Doubleword with Update DS-form

stdu RS,DS(RA)

62	RS	RA	DS	1
0	6	11	16	30 31

```
EA ← (RA) + EXTS(DS || 0b00)
MEM(EA, 8) ← (RS)
RA ← EA
```

```
union ast member (bit[5],bit[5],bit[14]) Stdu

function clause decode
 (0b111110 : (bit[5]) RS : (bit[5]) RA
  : (bit[14]) DS : 0b01 as instr) =
 Stdu (RS,RA,DS)

function clause execute (Stdu (RS, RA, DS)) =
{ EA := GPR[RA] + EXTS (DS : 0b00);
  MEMw(EA,8) := GPR[RS];
  GPR[RA] := EA }
```

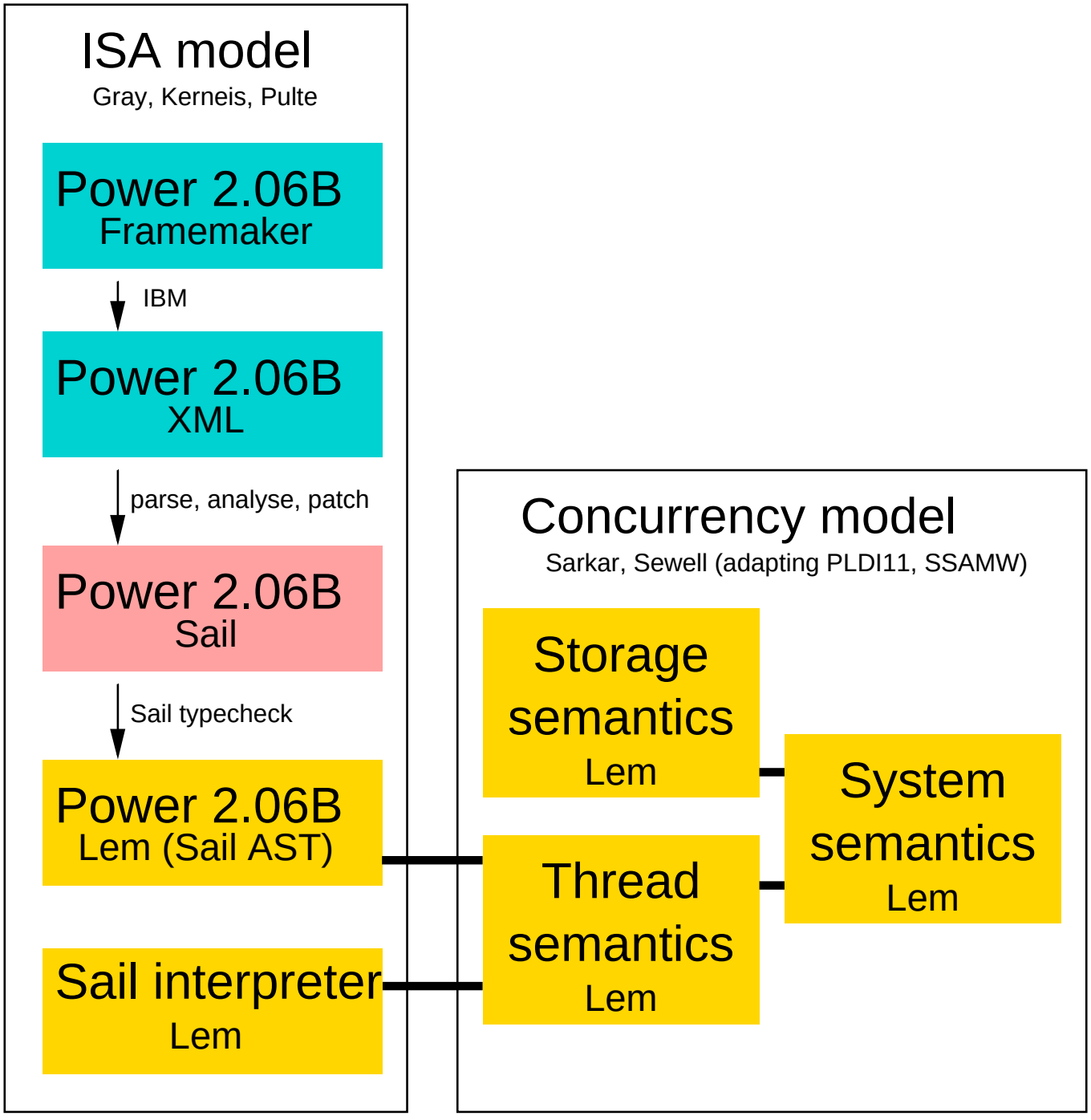
Problem 2: What Does It Mean?

```
function clause execute (Stdu (RS, RA, DS)) =
{ EA := GPR[RA] + EXTS (DS : 0b00);
  MEMw(EA,8) := GPR[RS];
  GPR[RA] := EA }
```

For sequential machine: run the micro-ops of each instruction in turn, sequentially, updating a shared memory state and thread-local register state

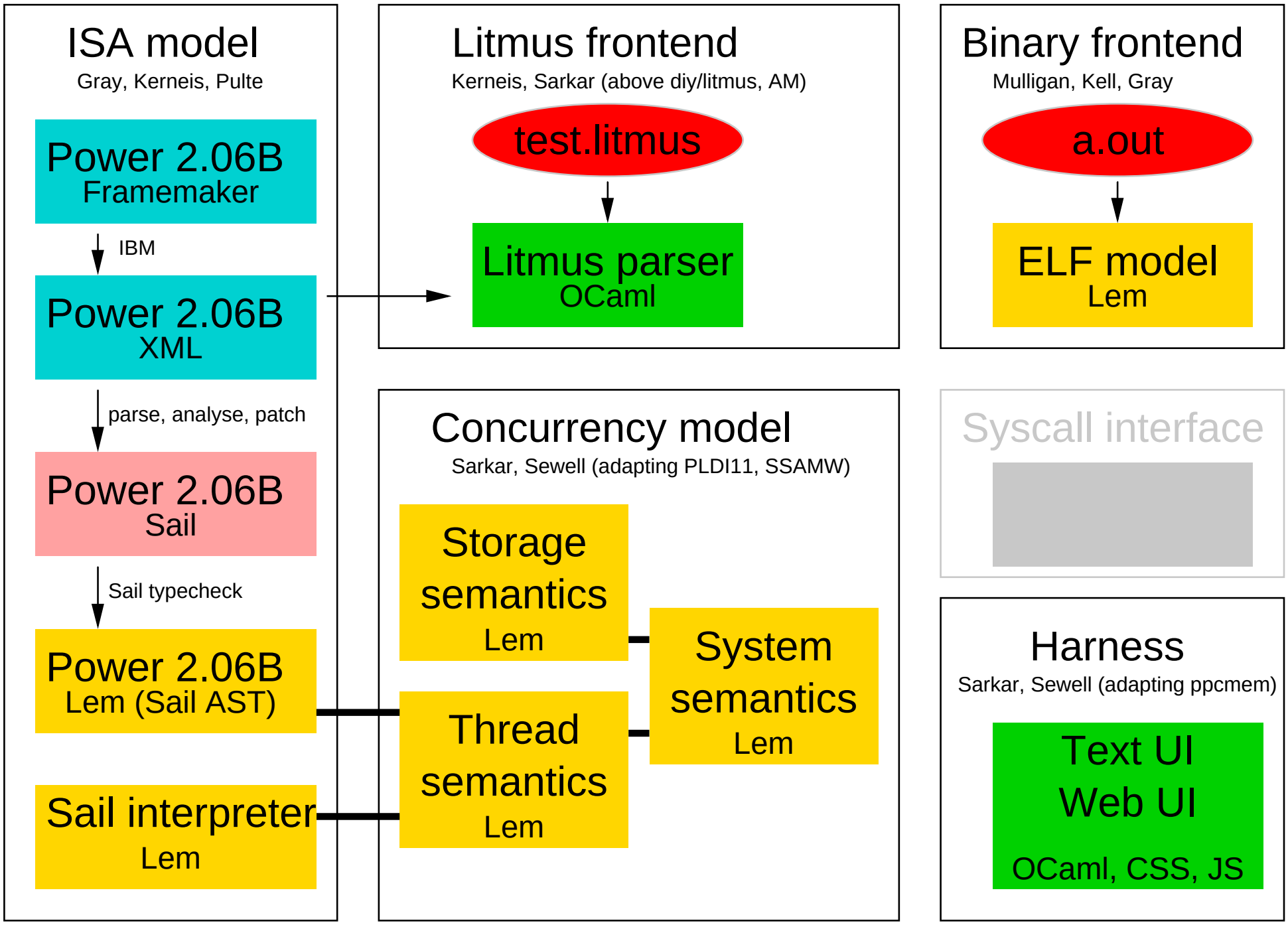
For SC or TSO multiprocessor: similar, interleaving

But ARM and Power? Observably out-of-order, speculative, non-multi-copy atomic, non-atomic intra-instruction semantics, dependency-sensitive


ISA model
Gray, Kerneis, Pulte
Power 2.06B
Framemaker
IBM
Power 2.06B
XML
parse, analyse, patch
Power 2.06B
Sail
Sail typecheck
Power 2.06B
Lem (Sail AST)
Sail interpreter
Lem
Concurrency model
Sarkar, Sewell (adapting PLDI11, SSAMW)
Storage
semantics
Lem
System
semantics
Lem
Thread
semantics
Lem

ISA / Concurrency Interface

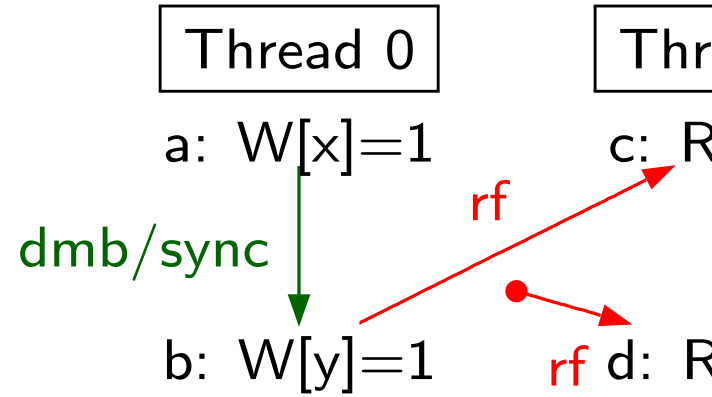
```
type instruction_state

val interp : instruction_state -> outcome

type outcome =
| Barrier of barrier_kind * instruction_state
| Read_mem of read_kind * address_lifted * nat
  * (memory_value -> instruction_state)
| Write_mem of write_kind * address_lifted * nat * memory_value
  * (bool -> instruction_state)
| Read_reg of reg_name * (register_value -> instruction_state)
| Write_reg of reg_name * register_value * instruction_state
| ...
```


ISA model
Gray, Kerneis, Pulte
Power 2.06B
Framemaker
IBM
Power 2.06B
XML
parse, analyse, patch
Power 2.06B
Sail
Sail typecheck
Power 2.06B
Lem (Sail AST)
Sail interpreter
Lem
Litmus frontend
Kerneis, Sarkar (above diy/litmus, AM)
test.litmus
Litmus parser
OCaml
Binary frontend
Mulligan, Kell, Gray
a.out
ELF model
Lem
Concurrency model
Sarkar, Sewell (adapting PLDI11, SSAMW)
Storage
semantics
Lem
System
semantics
Lem
Thread
semantics
Lem
Syscall interface
Harness
Sarkar, Sewell (adapting ppcmem)
Text UI
Web UI
OCaml, CSS, JS

Demo

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	



Test MP+dmb/sync+ctrl: Al

```
 P0             | P1                ;
stw r7,0(r1)    | lwz r5,0(r2)      ;
sync            | cmpw r5,r7        ;
stw r8,0(r2)    | beq  L            ;
                | L:                ;
                | lwz r4,0(r1)      ;
```

ARM
Testing
Performance
...

“Architectural Emulator”?

System that takes a machine program and gives you *all* architecturally allowed behaviours

“Architectural Emulator”?

System that takes a machine program and gives you *all* architecturally allowed behaviours

Either:

- interactively
- exhaustively (for small programs!)
- pseudorandomly (but complete in the limit)

For use as a test oracle for testing h/w, and for testing s/w.

“Architectural Emulator”?

System that takes a machine program and gives you *all* architecturally allowed behaviours

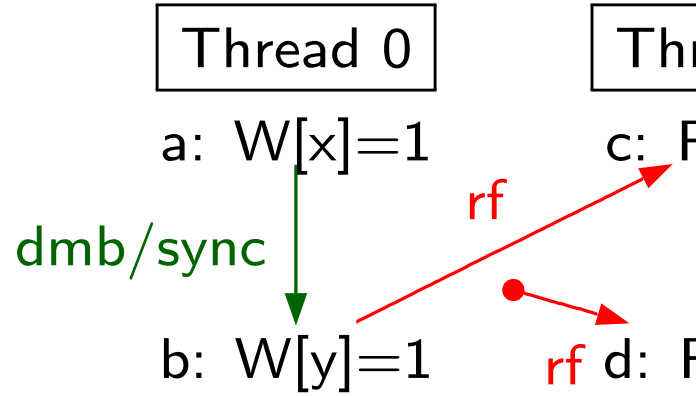
Preferably embodying an architecture definition that also serves:

- for informal communication — engineer-accessible
- for proof — mathematically precise

No Single Program Point

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	

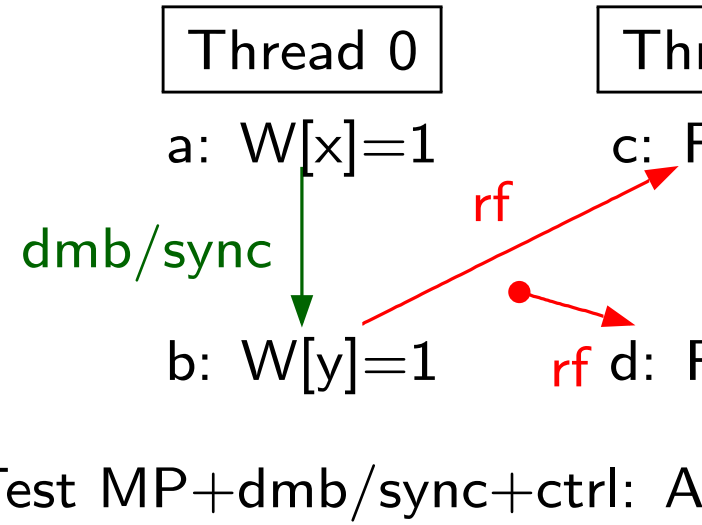


Test MP+dmb/sync+ctrl: A

No Single Program Point

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	



Test MP+dmb/sync+ctrl: A

Hence: we must maintain a list or tree of in-flight instructions

No Collected Register State

MP+dmb/sync+rs

Thread 0	Thread 1
x=1 dmb/sync y=1	r3=y r1=r3 r3 = x
Allowed: 1:r1=1 $\wedge$ 1:r3=0	

Hence: for a register read, we must walk back through its program-order predecessors to find the most recent that might write to that register (and block if it hasn't yet)

We assume each instruction has a determined register read+write footprint (calculate with exhaustive interpreter) and that it writes exactly once to each in the write footprint (eyeball check).

Reading from uncommitted instructions

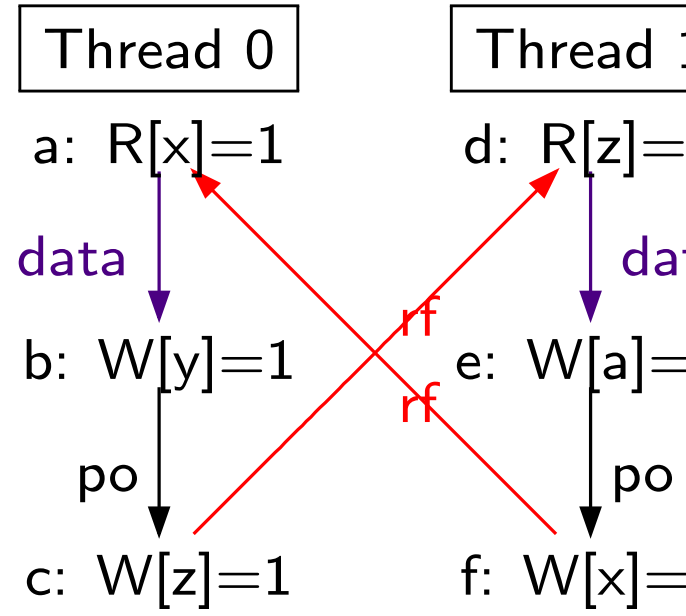
...instructions have to be able to read from register writes of uncommitted program-order-previous instructions

...and they also have to be able to read from *memory* writes of uncommitted program-order-previous instructions
(cf PPOCA, observable on Power and ARM)

n-atomic intra-instruction semantics for register rea

LB+datas+WW

Thread 0	Thread 1
a: r1=x	d: r2=z
b: y=r1	e: a=r2
c: z=1	f: x=1
Initial state: x=0 ∧ z=0	
Allowed: r1=1 ∧ r2=1	



Test LB+datas+WW: Allow

```
function clause execute (Stdu (RS, RA, DS)) =
{ EA := GPR[RA] + EXTS (DS : 0b00);
  MEMw(EA,8) := GPR[RS];
  GPR[RA] := EA }
```

...calculate might-access-same-address using exhaustive

Register Granularity Matters

- entire registers (including the flags register as a single entity)?
- the 4-bit subfields of the Power CR flags register?
- individual bits?

Commit Atomicity?

We used to assume that an in-flight instruction commits when it's finished, and at that point all writes and barriers become visible to the storage subsystem.

But:

```
function clause execute (Stdu (RS, RA, DS)) =
{ EA := GPR[RA] + EXTS (DS : 0b00);
  MEMw(EA,8) := GPR[RS];
  GPR[RA] := EA }
```

Now assume an instruction has at most one memory read, write, or barrier. Its micro-ops are executed in-order, and might be committed when it reaches a write or barrier. Then *finished* later.

Load/store Multple?

Rewrite to have a single (wide) read or write in Sail.

Plan to have surrounding plumbing split that up into multiple memory writes for storage subsystem.

(sound w.r.t. out-of-order execution after a partially executed load-multiple?)

Execution Past a Conditional Branch

In `beq target` pseudocode, `NIA` is calculated after the register reads that determine whether the branch is taken, but the h/w can speculate in either direction before those values are available.

```
function clause execute (Bc (BO, BI, BD, AA, LK)) =
{ if mode64bit then M := 0 else M := 32;
  if ~ (BO[2]) then CTR := CTR - 1 else ();
  ctr_ok := (BO[2] | (CTR[M .. 63] != 0) ^ BO[3]);
  cond_ok := (BO[0] | CR[BI + 32] ^ ~ (BO[1]));
  if ctr_ok & cond_ok then
    if AA then
      NIA:=EXTS(BD:0b00)
    else
      NIA:=CIA+EXTS(BD:0b00)
  else
```

Computed Branch Speculation?

```
function clause execute (Bclr (BO, BI, BH, LK)) =
{
  if mode64bit then M := 0 else M := 32;
  if ~ (BO[2]) then CTR := CTR - 1 else ();
  ctr_ok := (BO[2] | (CTR[M .. 63] != 0) ^ BO[3]);
  cond_ok := (BO[0] | CR[BI + 32] ^ ~ (BO[1]));
  if ctr_ok & cond_ok then NIA := LR[0..61]:0b00 else ();
  if LK then LR := CIA + 4 else ()
}
```