Understanding POWER multiprocessors

Susmit Sarkar

University of Cambridge

Multicore Programming, 2011

POWER: A Different Hardware Model

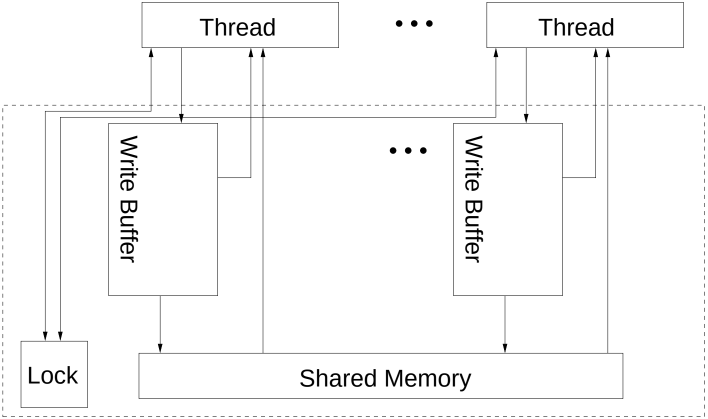
IBM Power – used in big servers (with lots of processors)

Different Relaxed Memory Model to X86
 . . . and much more subtle

ARM memory model is similar: your (next?) phone/tablet!

High-level languages (C++/C) models informed by POWER/ARM features

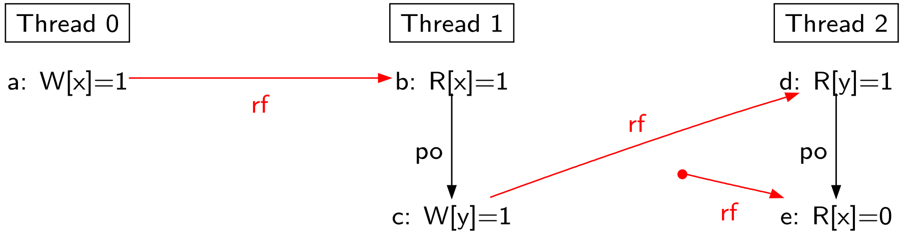
Recall: X86-TSO



On POWER, things are a little more subtle!

Subtlety 1: Writes propagating in different orders

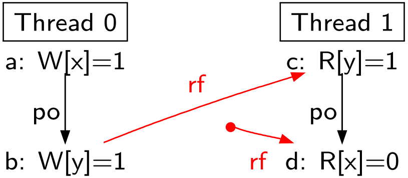
Write-to-Read Causality: WRC



Test WRC : Allowed

Writes propagate to different threads in different orders

Subtlety 2: Program order not maintained



Test MP : Allowed

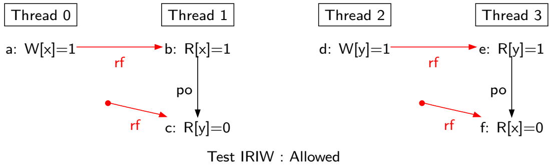
Writes and reads can execute out-of-program order

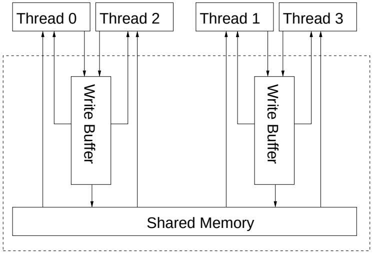
What is going on?

Visible Microarchitectural Effects:

- Out-of-order, and Speculative Execution
- Buffering of Stores and Loads
- Topology of Interconnection

A (Misleading) Microarchitectural view

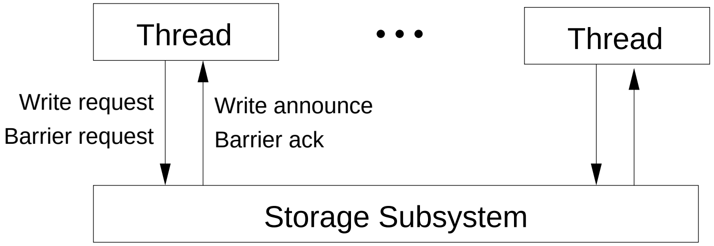




Perhaps with even more hierarchy

The model structure

Overall structure:



- Some aspects are thread-only, some storage-only, some both
- Threads and Storage Subsystem: Abstract state machines

  Speculative execution in Threads;
  Topology-independent Storage Subsystem
- Formally: transitions, guarded by preconditions, change state, and synchronize with each other

Storage Subsystem I

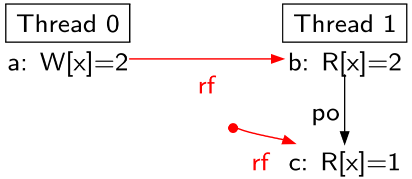
Storage subsystem has (among other things):

- Per-thread, a list of events propagated there
- The last write is the value to be read

Stores can propagate to another thread at any time,
subject to . . .

Coherence

Coherence: total order of writes to each location,
Such that no reader reads out-of-sequence



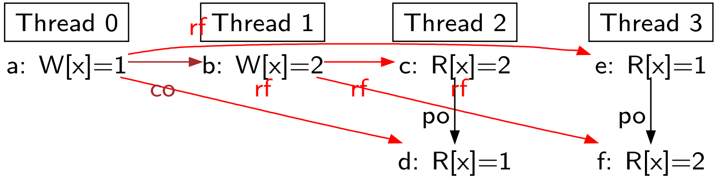
Test CoRR1 : Forbidden

Storage Subsystem II

Storage subsystem has (among other things):

- Per-thread, a list of events propagated there
- The last write is the value to be read
- A set of constraints on coherence order (partial order)
- (Partial) Coherence Commitments can be made at any time

Write Propagation and Coherence: Demo



Test CoRR2 : Forbidden

Thread Subsystem I

Instructions can be executed out-of-order, and speculated
In-flight instructions are committed when [. . . TBD]

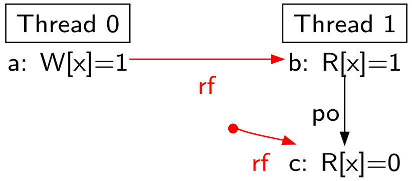
$$\begin{tikzcd}
 & & & & & i_6 \arrow[r] & i_7 & & & \\
\cdots\ i_1 \arrow[r] & i_2 \arrow[r] & i_3 \arrow[r] & i_4 \arrow[r] & i_5 \arrow[ur] \arrow[dr] & & & i_{10} \arrow[r] & i_{11} \arrow[r] & i_{12} \\
 & & & & & i_8 \arrow[r] & i_9 \arrow[ur] \arrow[dr] & & & \\
 & & & & & & & i_{13} & &
\end{tikzcd}$$

Until then, subject to rollback

Thread Subsystem II

Read requests can be issued at any time, satisfied by storage subsystem
Remember: Read request is different from Committing Read
Subject to rollback

Read Satisfy and Restart: Demo



Test CoRR1 : Forbidden

Enforcing order where needed

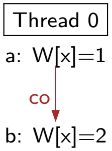
How to enforce order?

- Coherence
- Dependencies
- Barriers
- Synchronizing Instructions (LL/SC)

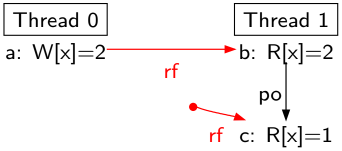
Restoring Order when needed: Proposed C++ mapping

C++11 Operation	POWER Implementation
Non-atomic Load	`ld`
Load Relaxed	`ld`
Load Consume	`ld` (and preserve dependency)
Load Acquire	`ld; cmp; bc; isync`
Load Seq Cst	`hwsync; ld; cmp; bc; isync`
Non-atomic Store	`st`
Store Relaxed	`st`
Store Release	`lwsync; st`
Store Seq Cst	`hwsync; st`
Cmpxchg Relaxed	`_loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; _exit:`
Cmpxchg Acquire	`_loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit:`
Cmpxchg Release	`lwsync; _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; _exit: stwcx.; bc _loop; isync; _exit:`
Cmpxchg SeqCst	`hwsync; _loop: lwarx; cmp; bc _exit; stwcx.; bc _loop; isync; _exit:`
Acquire Fence	`lwsync`
Release Fence	`lwsync`
SeqCst Fence	`hwsync`

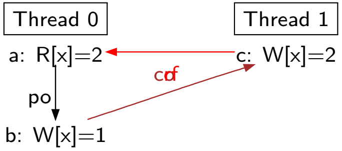
Coherence



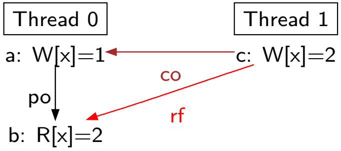
Test CoWW : Forbidder



Test CoRR1 : Forbidden



Test CoRW : Forbidden



Test CoWR : Forbidden

Dependencies

Address Dependency: A read feeds value of address of subsequent read/write

```
r1 = x; y = (*r1);
```

Data Dependency: A read feeds value of data of subsequent write

```
r2 = x; y = r2;
```

Control Dependency: A read feeds a condition value branched on, and a write is after the branch

```
if (x == 1) {y = 2;}
```

Dependencies, microarchitecturally

Address Dependency:

```
r1 = x; y = (*r1);
```

Cannot ask storage subsystem to do read/write before address available

Data Dependency:

```
r2 = x; y = r2;
```

Cannot ask storage subsystem to do write before data available

Control Dependency:

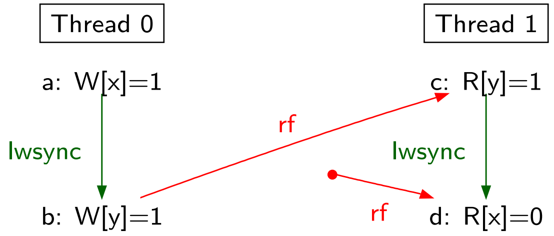
```
if (x == 1) {y = 2;}
```

Cannot ask storage subsystem to commit write before branch resolved

Barriers

- General-purpose (often expensive) instructions to restore order
- Here will look at two: `lwsync` and `sync`
- (Others used for systems programming: `ptesync`, `eieio`, `mbar`)

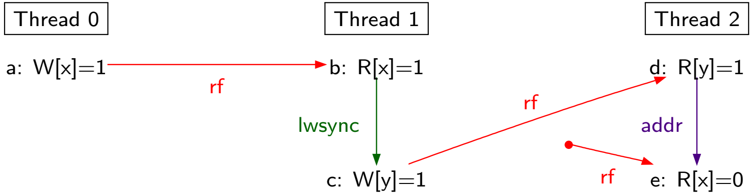
Barriers keep instructions in order



Test MP+lwsyncs : Forbidden

Instructions committed in order, and
Writes propagated in order

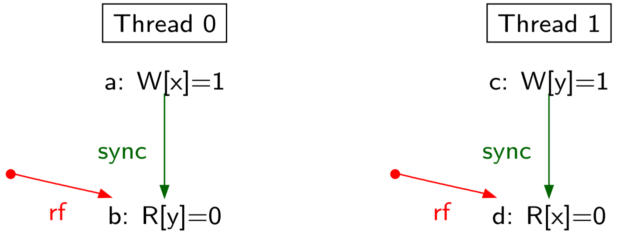
Programming on many-threads: Cumulativity



Test WRC+lwsync+addr : Forbidden

Keep writes from other threads in order

(Heavyweight) Sync



Test SB+syncs : Forbidden

Have to wait till sync (and preceding writes) propagated everywhere

Restore SC if every instruction is separated by a sync (Can now be proved!)

How do we know?

- Read the Manuals

  “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

- Run lots of tests

WRC	23M/93G
WRC+sync+addr	0/110G

- Discuss with Designers/Architects

- Make tentative model, and Repeat

Smart synchronizations: Compare-and-Swap

Clever algorithms often require atomic load/store

Most commonly: Compare-and-Swap (CAS)

. . . CAS x,v1,v2: *if* `x` holds `v1`, then atomically write `v2`, else fail

e.g. Concurrent List (simplified):

```
push(x) {
    do {
        r = head;
        x.next = r;
    } while (!CAS (head,r,x))
}
```

Problems with CAS

```
push(x) {
    do {
        r = head;
        x.next = r;
    } while (!CAS (head,r,x))
}
```

- Two reads
- ABA problem: can concurrently change!

More general solution: LL/SC

Load-linked (`lwarx`): read data
Store-conditional (`stwcx`): if last value read is still most-recent (?) write

More general: CAS(x,v1,v2) is just

```
lwarx x,r1; cmp r1 v1; bc _exit; stwcx x,v2; _exit
```

But what *is* most recent?

- Microarchitecturally: if we have not lost the cache-line since last lwarx then stwcx can succeed
- More abstractly: need to relate to other events

Modeling `lwarx`/`stwcx`

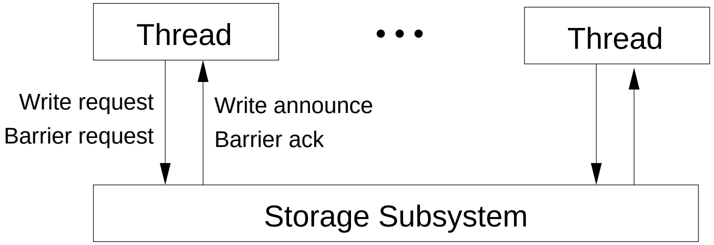
Intuition: Have to be atomic *for that location*

Idea: Look at coherence order

For a `stwcx` to succeed, it should be coherence later the write read from by the `lwarx`, *and* no other write should intervene

Coherence Point: everything below in coherence is linear (all decisions made), and no other write is later allowed to come below

Recap



How to enforce order?

- Coherence
- Dependencies
- Barriers
- Synchronizing Instructions (LL/SC)

More details

Paper and additional materials:

`http://www.cl.cam.ac.uk/~pes20/ppc-supplemental`

Look most particularly at the `ppcmem` tool:

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