More aggressively relaxed architectures: ARM, IBM POWER, and RISC-V

November 21, 2019

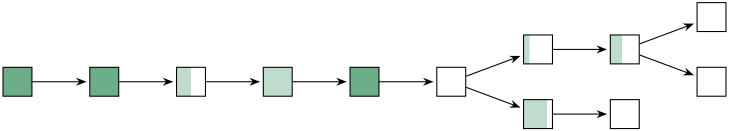
x86

- programmers can usually assume instructions execute in program order (but with FIFO store buffer)
- (actual hardware may be more aggressive, but not visibly so)

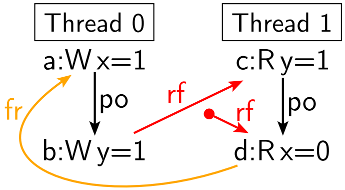
ARM, IBM POWER, RISC-V

- by default, instructions can observably execute out-of-order and speculatively
- ...except as forbidden by coherence, dependencies, barriers
- much weaker than x86-TSO
- similar but not identical to each other

Most observable relaxed phenomena can be viewed as arising from pipeline effects – out-of-order and speculative execution:



Message Passing (MP) Again



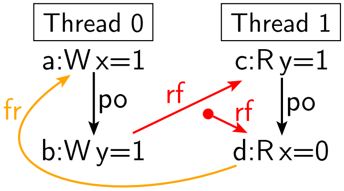
MP AArch64

Thread 0	Thread 1
`STR X0,[X1]//a`	`LDR X0,[X1]//c`
`STR X0,[X2]//b`	`LDR X2,[X3]//d`

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;`

Allowed: `1:X0=1; 1:X2=0;`

Message Passing (MP) Again



MP AArch64

Thread 0	Thread 1
`STR X0,[X1]//a`	`LDR X0,[X1]//c`
`STR X0,[X2]//b`	`LDR X2,[X3]//d`

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;`

Allowed: `1:X0=1; 1:X2=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

Message Passing (MP) Again

$$\begin{tikzcd}
\boxed{\text{Thread 0}} & & \boxed{\text{Thread 1}} \\
a{:}W\,x{=}1 \arrow[d, "po"] & & c{:}R\,y{=}1 \arrow[d, "po"] \\
b{:}W\,y{=}1 \arrow[urr, "rf"] & \bullet \arrow[r, "rf"] & d{:}R\,x{=}0 \arrow[ull, bend left=60, "fr"]
\end{tikzcd}$$

MP AArch64

Thread 0	Thread 1
`STR X0,[X1]//a`	`LDR X0,[X1]//c`
`STR X0,[X2]//b`	`LDR X2,[X3]//d`

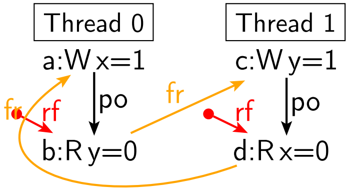
Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;`

Allowed: `1:X0=1; 1:X2=0;`

Microarchitecturally:

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

SB Again



SB AArch64

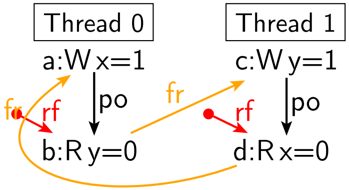
Thread 0	Thread 1

`STR X0,[X1]//a`
`LDR X2,[X3]//b`
	`STR X0,[X1]//c`
	`LDR X2,[X3]//d`

Initial state: `0:X3=y; 0:X1=x; 0:X0=1; 0:X2=0; 1:X3=x; 1:X1=y; 1:X0=1; 1:X2=0; y=0; x=0;`

Allowed: `0:X2=0; 1:X2=0;`

SB Again



SB AArch64

Thread 0	Thread 1
`STR X0,[X1]//a`	`STR X0,[X1]//c`
`LDR X2,[X3]//b`	`LDR X2,[X3]//d`

Initial state: `0:X3=y; 0:X1=x; 0:X0=1; 0:X2=0; 1:X3=x; 1:X1=y; 1:X0=1; 1:X2=0; y=0; x=0;`

Allowed: `0:X2=0; 1:X2=0;`

Microarchitecturally:

- pipeline: out-of-order execution of the store and load
- write buffering

So what guarantees do you get?

Coherence

Reads and writes *to each location in isolation* behave SC

CoRW1

$$\begin{tikzcd}
\text{a:R x=1} \arrow[d, "\text{po}"] \\
\text{b:W x=1} \arrow[u, bend left, "\text{rf}"]
\end{tikzcd}$$

CoWR0

$$\begin{tikzcd}
\text{a:W x=1} \arrow[d, "\text{po}"] & \\
\text{b:R x=0} \arrow[u, bend left, "\text{fr}"] & {} \arrow[l, "\text{rf}"']
\end{tikzcd}$$

CoWW

$$\begin{tikzcd}
\text{a:W x=1} \arrow[d, "\text{po}"] \\
\text{b:W x=2} \arrow[u, bend left, "\text{co}"]
\end{tikzcd}$$

CoRW2

$$\begin{tikzcd}
\boxed{\text{Thread 0}} & \boxed{\text{Thread 1}} \\
\text{a:W x=1} \arrow[r, "\text{rf}"] & \text{b:R x=1} \arrow[d, "\text{po}"] \\
 & \text{c:W x=2} \arrow[ul, "\text{co}"]
\end{tikzcd}$$

CoWR

$$\begin{tikzcd}
\boxed{\text{Thread 0}} & \boxed{\text{Thread 1}} \\
\text{a:W x=1} \arrow[r, "\text{co}"] \arrow[dr, "\text{rf}"] & \text{b:W x=2} \arrow[d, "\text{po}"] \\
 & \text{c:R x=1} \arrow[u, bend left, "\text{fr}"]
\end{tikzcd}$$

CoRR

$$\begin{tikzcd}
\boxed{\text{Thread 0}} & \boxed{\text{Thread 1}} \\
\text{a:W x=1} \arrow[r, "\text{rf}"] & \text{b:R x=1} \arrow[d, "\text{po}"] \\
{} \arrow[r, "\text{rf}"] & \text{c:R x=0} \arrow[ul, "\text{fr}"]
\end{tikzcd}$$

All these are forbidden

Coherence

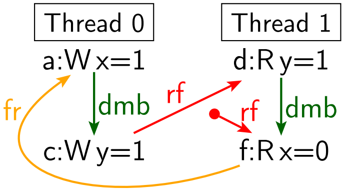
Reads and writes *to each location in isolation* behave SC

In any execution, for each location, there exists some total order co over the writes to that location, that's consistent with program order (on each hardware thread) and with reads-from.

Microarchitecturally:

- cache protocol (MSI, MESI, MOESI,...)
- interconnect design as a whole
- hazard checks in the pipeline

Enforcing Order with Barriers



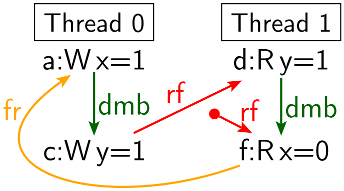
MP+dmb.sys AArch64

Thread 0	Thread 1
`STR X0,[X1]//a`	`LDR X0,[X1]//d`
`DMB SY``//b`	**`DMB SY`**`//e`
`STR X0,[X2]//c`	`LDR X2,[X3]//f`

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;`

Forbidden: `1:X0=1; 1:X2=0;`

Enforcing Order with Barriers



MP+dmb.sys AArch64

Thread 0	Thread 1
`STR X0,[X1]//a`	`LDR X0,[X1]//d`
`DMB SY``//b`	**`DMB SY`**`//e`
`STR X0,[X2]//c`	`LDR X2,[X3]//f`

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;`

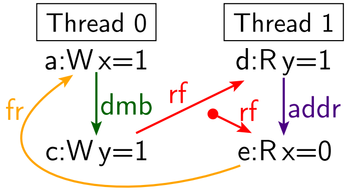
Forbidden: `1:X0=1; 1:X2=0;`

		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	—	—	—	—

The ARMv8-A dmb sy, IBM POWER sync, or RISC-V fence rw,rw memory barrier prevents reordering of loads and stores.

Likewise, inserting those barriers is enough to make SB forbidden.

Enforcing Order with Dependencies (read-to-read address)



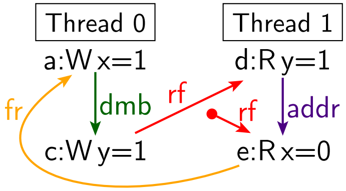
MP+dmb.sy+addr AArch64

Thread 0	Thread 1
`STR X0,[X1]//a`	`LDR X0,[X1]//d`
`DMB SY``//b`	`EOR X2,X0,X0`
`STR X0,[X2]//c`	`LDR X3,[X4,X2]//e`

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X4=x; 1:X1=y; 1:X0=0; 1:X3=0; y=0; x=0;`

Forbidden: `1:X0=1; 1:X3=0;`

Enforcing Order with Dependencies (read-to-read address)



MP+dmb.sy+addr AArch64

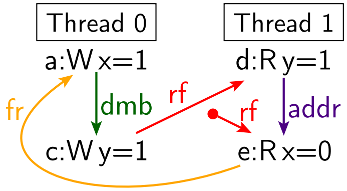
Thread 0	Thread 1
`STR X0,[X1]//a`	`LDR X0,[X1]//d`
`**DMB SY**//b`	`EOR X2,X0,X0`
`STR X0,[X2]//c`	`LDR X3,[X4,X2]//e`

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X4=x; 1:X1=y; 1:X0=0; 1:X3=0; y=0; x=0;`

Forbidden: `1:X0=1; 1:X3=0;`

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

Enforcing Order with Dependencies (read-to-read address)



MP+dmb.sy+addr AArch64

Thread 0	Thread 1
`STR X0,[X1]//a`	`LDR X0,[X1]//d`
`DMB SY``//b`	`EOR X2,X0,X0`
`STR X0,[X2]//c`	`LDR X3,[X4,X2]//e`

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X4=x; 1:X1=y; 1:X0=0; 1:X3=0; y=0; x=0;`

Forbidden: `1:X0=1; 1:X3=0;`

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

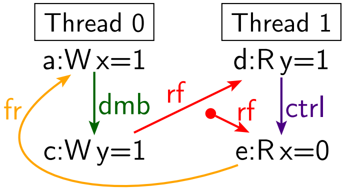
Architectural guarantee to respect read-to-read address dependencies even if they are "false" or "artificial", i.e. if they could "obviously" be optimised away.

x=1;	r1 = y;
y=2;	r2 = *(&x + (r1 ^ r1)) ;

x=1;	r1 = y;
y=&x;	r2 = *r1;

Beware: C/C++ do *not* guarantee to respect dependencies!

Enforcing Order with Dependencies (read-to-read control)



MP+dmb.sy+ctrl AArch64

Thread 0	Thread 1
`STR X0,[X1]//a`	`LDR X0,[X1]//d`
`DMB SY` `//b`	`CBNZ X0,LC00`
`STR X0,[X2]//c`	`LC00:`
	`LDR X2,[X3]//e`

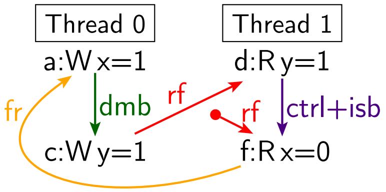
Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;`

Allowed: `1:X0=1; 1:X2=0;`

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

Architecturally: read-to-read control dependencies are not respected.

Enforcing Order with Dependencies (read-to-read ctrl-isb)



MP+dmb.sy+ctrlisb AArch64

Thread 0	Thread 1
`STR X0,[X1] //a`	`LDR X0,[X1] //d`
`DMB SY //b`	`CBNZ X0,LC00`
`STR X0,[X2] //c`	`LC00:`
	`ISB //e`
	`LDR X2,[X3] //f`

Initial state: `0:X2=y; 0:X1=x; 0:X0=1; 1:X3=x; 1:X1=y; 1:X0=0; 1:X2=0; y=0; x=0;`

Forbidden: `1:X0=1; 1:X2=0;`

Can strengthen with an ISB (Arm) or isync (POWER) 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: Summary

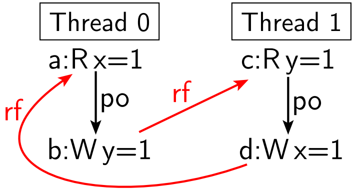
Read-to-Read: address and control-isb/control-isync dependencies respected; control dependencies *not* respected

Read-to-Write: address, data, *and control* dependencies all respected (writes are not observably speculated, at least as far as other threads are concerned)

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

“Load Buffering”?

Dual of first SB test:



LB AArch64

Thread 0	Thread 1
`LDR X0,[X1]//a`	`LDR X0,[X1]//c`
`STR X2,[X3]//b`	`STR X2,[X3]//d`

Initial state: `0:X3=y; 0:X2=1; 0:X1=x; 0:X0=0; 1:X3=x; 1:X2=1; 1:X1=y; 1:X0=0; y=0; x=0;`

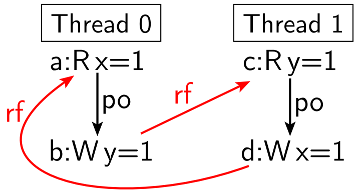
Allowed: `0:X0=1; 1:X0=1;`

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

Architecturally allowed on ARM, POWER, and RISC-V

"Load Buffering"?

Dual of first SB test:


$$\begin{tikzcd} \boxed{\text{Thread 0}} & \boxed{\text{Thread 1}} \\ a{:}R\,x{=}1 \arrow[d, "po"] & c{:}R\,y{=}1 \arrow[d, "po"] \\ b{:}W\,y{=}1 \arrow[ur, "rf"] & d{:}W\,x{=}1 \arrow[ul, bend left, "rf"] \end{tikzcd}$$

LB	AArch64
Thread 0	Thread 1
`LDR X0,[X1]//a`	`LDR X0,[X1]//c`
`STR X2,[X3]//b`	`STR X2,[X3]//d`

Initial state: `0:X3=y; 0:X2=1; 0:X1=x; 0:X0=0; 1:X3=x; 1:X2=1; 1:X1=y; 1:X0=0; y=0; x=0;`

Allowed: `0:X0=1; 1:X0=1;`

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

Architecturally allowed on ARM, POWER, and RISC-V
Forbid with address or data dependencies:

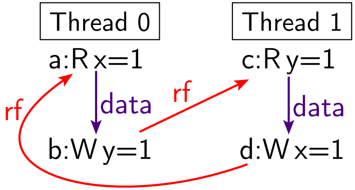
		POWER			ARM			
	Kind	PowerG5	Power6	Power7	Tegra2	Tegra3	APQ8060	A5X
LB	Allow	0/7.4G	0/43G	0/258G	1.5M/3.9G	124k/16M	58/1.6G	1.3M/185M
LB+addrs	Forbid	0/6.9G	0/40G	0/216G	0/24G	0/39G	0/26G	0/2.2G
LB+datas	Forbid	0/6.9G	0/40G	0/252G	0/16G	0/23G	0/18G	0/2.2G
LB+ctrls	Forbid	0/4.5G	0/16G	0/88G	0/8.1G	0/7.5G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

LB+datas – thin-air values?

$$\begin{tikzcd}
\boxed{\text{Thread 0}} & \boxed{\text{Thread 1}} \\
\text{a:R x=1} \arrow[d, "\text{data}"] & \text{c:R y=1} \arrow[d, "\text{data}"] \\
\text{b:W y=1} \arrow[ur, "\text{rf}"] & \text{d:W x=1} \arrow[ul, bend left=60, "\text{rf}"]
\end{tikzcd}$$

r1=x
y=r1
	r2=y
	x=r2

LB+datas – thin-air values?



r1=x	r2=y
y=r1	x=r2

Forbidden!

Iterated Message Passing and Cumulative Barriers

WRC-loop Pseudocode

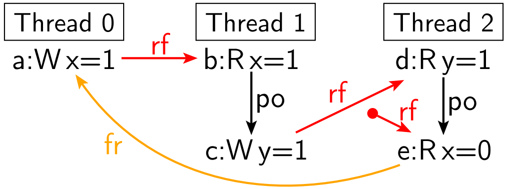
Thread 0	Thread 1	Thread 2

x=1	while (x==0) {}
	y=1
		while (y==0) {}
		r3=x

Initial state: x=0 ∧ y=0		
Forbidden?: 2:r3=0		

First, replace loops by a non-looping test with conditions on read values...

Iterated Message Passing and Cumulative Barriers



WRC AArch64

Thread 0	Thread 1	Thread 2

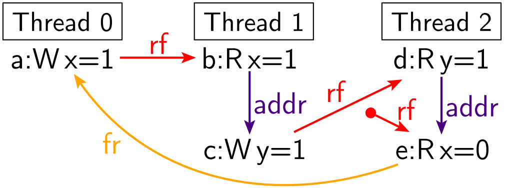
`STR X0,[X1]//a`	`LDR X0,[X1]//b`
	`STR X2,[X3]//c`
		`LDR X0,[X1]//d`
		`LDR X2,[X3]//e`

Initial state: `0:X1=x; 0:X0=1; 1:X3=y; 1:X2=1; 1:X1=x; 1:X0=0; 2:X3=x; 2:X1=y; 2:X0=0; 2:X2=0; y=0; x=0;`

Allowed: `1:X0=1; 2:X0=1; 2:X2=0;`

Trivially allowed, just by local reordering. Add address dependencies...

Iterated Message Passing and Cumulative Barriers



WRC+addrs AArch64

Thread 0	Thread 1	Thread 2

`STR X0,[X1] //a`	`LDR X0,[X1] //b`
	`EOR X2,X0,X0`
	`STR X3,[X4,X2] //c`
		`LDR X0,[X1] //d`
		`EOR X2,X0,X0`
		`LDR X3,[X4,X2] //e`

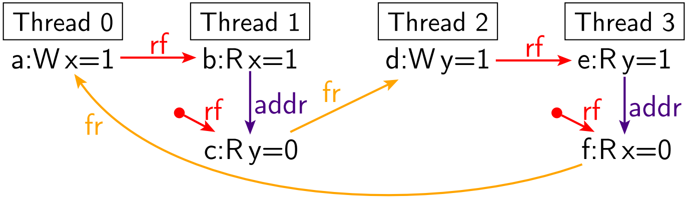
Initial state: `0:X1=x; 0:X0=1; 1:X4=y; 1:X3=1; 1:X1=x; 1:X0=0; 2:X4=x; 2:X1=y; 2:X0=0; 2:X3=0; y=0; x=0;`

Allowed: `1:X0=1; 2:X0=1; 2:X3=1;`

- IBM POWER: Allowed
- ARMv7-A and old ARMv8-A: Allowed
- current ARMv8-A: Forbidden
- RISC-V: Forbidden

Cumulative Barriers

A non-multicopy-atomic architecture needs *cumulative* barriers to be useful
WRC+fen+addr



IRIW+addrs AArch64

Thread 0	Thread 1	Thread 2	Thread 3

`STR X0,[X1]//a`	`LDR X0,[X1] //b`
	`EOR X2,X0,X0`
	`LDR X3,[X4,X2]//c`
		`STR X0,[X1]//d`	`LDR X0,[X1] //e`
			`EOR X2,X0,X0`
			`LDR X3,[X4,X2]//f`

Initial state: `0:X1=x; 0:X0=1; 1:X4=y; 1:X1=x; 1:X0=0; 1:X3=0; 2:X1=y; 2:X0=1; 3:X4=x; 3:X1=y; 3:X0=0; 3:X3=0; y=0; x=0;`

Forbidden: `1:X0=1; 1:X3=0; 3:X0=1; 3:X3=0;`

Likewise.

- x86, current ARMv8-A, RISC-V: *(other) multicopy atomic*
- IBM POWER, old ARMv8-A, ARMv7-A: *non-multicopy-atomic*

. . . continuing ARM/POWER/RISC-V concurrency

- introduce the formal model
- revisit some examples using the model

Most observable relaxed phenomena can be viewed as arising from pipeline effects – out-of-order and speculative execution.

So our model will have to explain this pipeline behaviour.

We could model the pipeline. But:

1. too complicated: micro-architectural detail
2. we don't have a pipeline model: confidential
3. it would be model of one CPU's pipeline,
   not architectural envelope

pipeline effects abstractly:

- instructions can be fetched before predecessors finished

pipeline effects abstractly:

- instructions can be fetched before predecessors finished
- instructions independently make progress

pipeline effects abstractly:

- instructions can be fetched before predecessors finished
- instructions independently make progress
- branch speculation allows fetching successors of branches

pipeline effects abstractly:

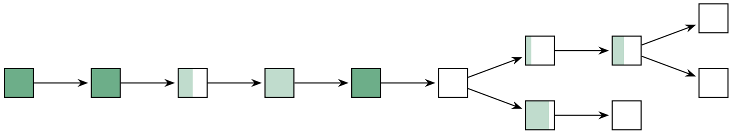
- instructions can be fetched before predecessors finished
- instructions independently make progress
- branch speculation allows fetching successors of branches
- multiple potential successors can be explored

pipeline effects abstractly:

- instructions can be fetched before predecessors finished
- instructions independently make progress
- branch speculation allows fetching successors of branches
- multiple potential successors can be explored

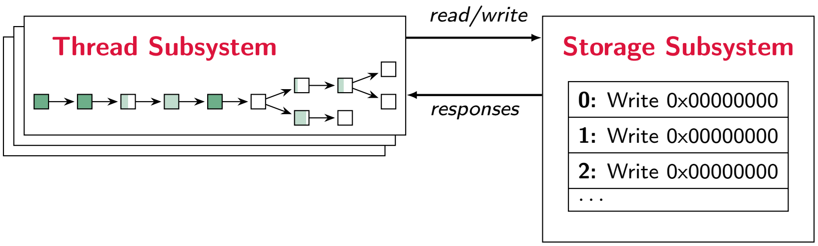
pipeline effects abstractly:

- instructions can be fetched before predecessors finished
- instructions independently make progress
- branch speculation allows fetching successors of branches
- multiple potential successors can be explored



Formal concurrency model

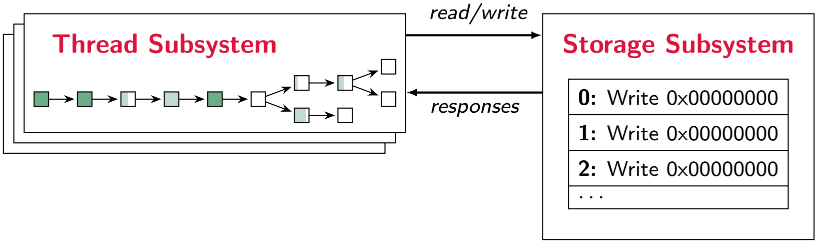
- each thread has a tree of instruction instances;
- threads execute in parallel above a simple memory state: mapping from addresses to write request



(For now: plain memory reads, writes, strong barriers.
All memory accesses of the same size.)

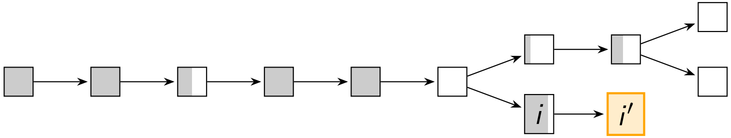
Formal concurrency model

- each thread has a tree of instruction instances;
- threads execute in parallel above a simple memory state: mapping from addresses to write request
- for Power: with fancier memory state



(For now: plain memory reads, writes, strong barriers.
All memory accesses of the same size.)

Fetch instruction instance

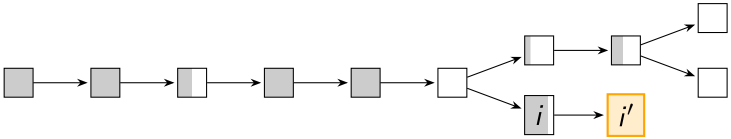


Condition:

A possible program-order successor i' of instruction instance i can be fetched from address *loc* and decoded if:

1. it has not already been fetched as successor of i
2. there is a decodable instruction in program memory at *loc*; and
3. *loc* is a possible next fetch address for i:
    - 3.1 for a non-branch/jump instruction, the successor instruction address (*i.program_loc+4*);
    - 3.2 for an instruction that has performed a write to the program counter register (`PC`), the value that was written;
    - 3.3 for a conditional branch, either the successor address or the branch target address; or
    - 3.4 . . . .

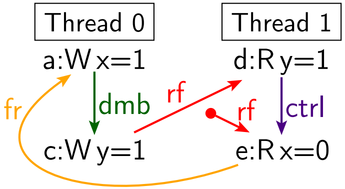
Fetch instruction instance



Action: construct a freshly initialised instruction instance i' for the instruction in program memory at *loc* and add i' to the thread's *instruction_tree* as a successor of i.

Example: speculative fetching

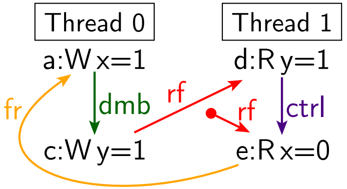
MP+dmb.sy+ctrl
(with “real” control dependency)



rmem web UI

Example: speculative fetching

MP+dmb.sy+ctrl
(with "real" control dependency)



rmem web UI

(Allowed. the barrier orders the writes, but the control dependency is weak: *e* can be speculatively fetched and satisfied early.)

Instruction semantics (ignore the details)

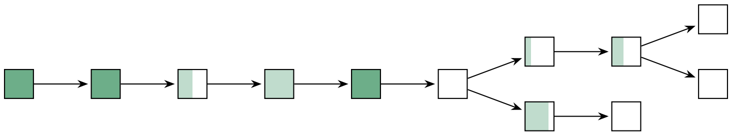
How do instructions work?

Instruction semantics (ignore the details)

How do instructions work? Each instruction is specified as a small imperative *Sail* program. For example:

```
function clause execute ( LoadRegister(n,t,m,acctype,memop, ...) ) = {
  (bit[64]) offset := ExtendReg(m, extend_type, shift);
  (bit[64]) address := 0;
  (bit['D]) data := 0;             (* some local definitions *)
  ...
  if n == 31 then { ...  } else
    address := rX(n);              (* read the address register *)

  if ~(postindex) then             (* some bitvector arithmetic *)
    address := address + offset;

  if memop == MemOp_STORE then     (* announce the address *)
    wMem_Addr(address, datasize quot 8, acctype, false);
  ...

  switch memop {
    case MemOp_STORE -> {
      if rt_unknown then
        data := (bit['D]) UNKNOWN
      else
        data := rX(t);             (* read the data register *)
```

Instruction instance states



each instruction instance has:

- `pseudocode_state`: the Sail state
- `reg_reads`, `reg_writes`: register accesses so far
- `mem_reads`, `mem_writes`: memory accesses so far
- status: finished, committed (for stores), ...
- the statically known register footprint: `regs_in`, `regs_out`
- `instruction_kind`: load, store, barrier, branch, ...
- ...

Sail pseudocode states (ignore the details)

```
type outcome =          (∗ request to concurrency model ∗)
  | Done                (∗ Sail execution ended ∗)
  | Internal of ..      (∗ Sail internal step ∗)
  | Read_mem of  ..     (∗ read memory ∗)
  | Write_ea of ..      (∗ announce write at address ∗)
  | Write_memv of ..    (∗ request to write memory ∗)
  | Read_reg of  ..     (∗ read register ∗)
  | Write_reg of ..     (∗ write register ∗)
  | Barrier of  ..      (∗ barrier effect ∗)
```

Sail pseudocode states (ignore the details)

```
type outcome =              (* request to concurrency model *)
  | Done                    (* Sail execution ended *)
  | Internal of ..          (* Sail internal step *)
  | Read_mem of  ..         (* read memory *)
  | Write_ea of ..          (* announce write at address *)
  | Write_memv of ..        (* request to write memory *)
  | Read_reg of  ..         (* read register *)
  | Write_reg of ..         (* write register *)
  | Barrier of  ..          (* barrier effect *)

type pseudocode_state =
  | Plain of outcome
  | Pending_memory_read of read_continuation
  | Pending_memory_write of write_continuation
```

Last lecture: in ARM, POWER, RISC-V, by default instructions execute out of order. Except, they provide certain guarantees:

- (BO) ordering from barriers
- (DO) ordering from dependencies
- (CO) coherence
- ...

The instruction tree machinery allows speculative and out-of-order execution. We will see how the model provides these guarantees.

Instruction life time: barrier instructions

- fetch and decode
- commit barrier
- finish

Commit Barrier

Condition:
A barrier instruction *i* in state Plain *(*Barrier*(barrier_kind, next_state′))* can be committed if:

1. all po-previous conditional branch instructions are finished;
2. (BO) if *i* is a `dmb sy` instruction, all po-previous memory access instructions and barriers are finished.

Commit Barrier

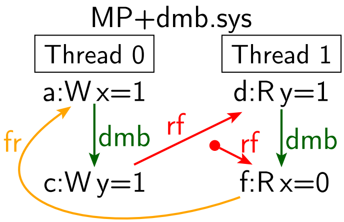
Action:

1. update the state of i to Plain $next\_state'$.

Barrier ordering

- so: a dmb barrier can only commit when all preceding memory accesses are finished
- a barrier commits before it finishes
- also (not seen yet): reads can only satisfy and writes can only propagate when preceding dmb barriers are finished

Barrier ordering



(Forbidden: *c* can only propagate when the dmb is finished, the dmb can only finish when committed, and only commit when *a* is propagated; similarly, the dmb on Thread 1 forces *f* to satisfy after *d*.)

Instruction life time: non-load/store/barrier instructions

for instance: `ADD`, branch, etc.

- fetch and decode
- register reads
- internal computation; just runs a Sail step (omitted)
- register writes
- finish

Register write

Condition:

An instruction instance i in state Plain $(\text{Write\_reg}(reg\_name, reg\_value, next\_state'))$ can do the register write.

Register write

Action:

1. record *reg_name* with *reg_value* and *write_deps* in *i.reg_writes*; and
2. update the state of *i* to Plain *next_state'*.

where *write_deps* is the set of all *read_sources* from *i.reg_reads*

...

Register read

Condition:

An instruction instance *i* in state Plain (Read_reg(*reg_name*, *read_cont*)) can do a register read if:

- (DO) the most recent preceding instruction instance that will write the register has done the expected register write.

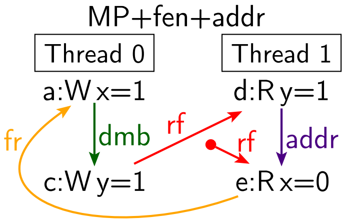
Register read

Let *read_source* be the write to *reg_name* by the most recent instruction instance that will write to the register, if any. If there is none, the source is the initial value. Let *reg_value* be its value.

Action:

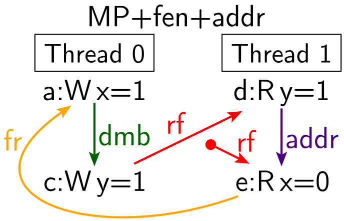
1. Record *reg_name*, *read_source*, and *reg_value* in *i.reg_reads*; and
2. update the state of *i* to Plain *(read_cont(reg_value))*.

Example: register dataflow dependencies



rmem web UI

Example: register dataflow dependencies



rmem web UI

(Forbidden. The barrier orders the writes, the address dependency prevents executing *e* before *d*.)

Instruction life time: loads

- fetch and decode
- register reads
- internal computation
- initiate read; when the address is available, constructs a read request (omitted)
- satisfy read
- complete load; hands the read value to the Sail execution (omitted)
- register writes
- finish

Satisfy read in memory

Condition:
A load instruction instance *i* in state Pending_mem_reads *read_cont* with unsatisfied read request *r* in *i.mem_reads* can satisfy *r* from memory if the read-request-condition predicate holds. This is if:

1. (BO) all po-previous `dmb sy` instructions are finished.

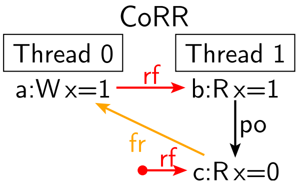
Satisfy read in memory

Let w be the write in memory to r's address. **Action:**

1. update r to indicate that it was satisfied by w; and
2. (CO) restart any speculative instructions which have violated coherence as a result of this.
   I.e. for every non-finished po-successor instruction i' of i with a same-address read request r', if r' was satisfied from a write $w' \neq w$ that is not from a po-successor of i, restart i' and its data-flow dependents.

Let w be the write in memory to r's address. **Action:**

1. update r to indicate that it was satisfied by w; and
2. (CO) restart any speculative instructions which have violated coherence as a result of this.
   I.e. for every non-finished po-successor instruction i' of i with a same-address read request r', if r' was satisfied from a write $w' \neq w$ that is not from a po-successor of i, restart i' and its data-flow dependents.



rmem web UI

(Forbidden. If c is satisfied from the initial write $x = 0$ before b is satisfied, once b reads from a it restarts c.)

Finish instruction

Condition:

A non-finished instruction *i* in state Plain *(*Done*)* can be finished if:

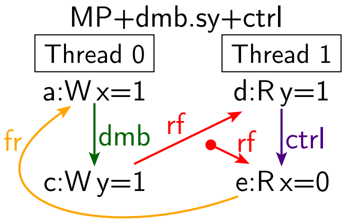
1. (CO) *i* has fully determined data;
2. all po-previous conditional branches are finished; and
3. if *i* is a load instruction:
   - 3.1 (BO) all po-previous `dmb sy` instructions are finished;
   - 3.2 (CO) it is guaranteed that the values read by the read requests of *i* will not cause coherence violations, i.e. . . .

Finish instruction

Action:

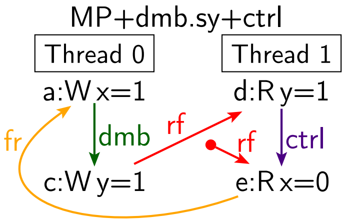
1. record the instruction as finished, i.e., set *finished* to *true*; and
2. if *i* is a branch instruction, discard any untaken path of execution. I.e., remove any (non-finished) instructions that are not reachable by the branch taken in *instruction_tree*.

Example: finishing loads and discarding branches



rmem web UI

Example: finishing loads and discarding branches



rmem web UI

(Speculatively executing the load past the conditional branch does not allow finishing the load until the branch is determined. Finishing the branch discards untaken branches.)

Instruction life time: stores

- fetch and decode
- register reads
- internal computation
- initiate write; when the address is available, constructs a write request without value (omitted)
- instantiate write; when the value is available, updates the write request's value (omitted)
- commit and propagate
- complete store; just resumes the Sail execution (omitted)
- finish

Commit store

Condition:
For an uncommitted store instruction *i* in state Pending_mem_writes *write_cont*, *i* can commit if:

1. (CO) *i* has fully determined data (i.e., the register reads cannot change);
2. all po-previous conditional branch instructions are finished;
3. (BO) all po-previous `dmb sy` instructions are finished;
4. (CO) all po-previous memory access instructions have initiated and have a fully determined footprint

Action: record *i* as committed.

Propagate write

Condition:

For an instruction *i* in state Pending_mem_writes *write_cont* with unpropagated write, *w* in *i.mem_writes*, the write can be propagated if:

1. (CO) all memory writes of po-previous store instructions that to the same address have already propagated
2. (CO) all read requests of po-previous load instructions to the same address have already been satisfied, and the load instruction is non-restartable.

Propagate write

Action:

1. record w as propagated; and
2. update the memory with w; and
3. (CO) restart any speculative instructions which have violated coherence as a result of this.
   I.e., for every non-finished instruction i' po-after i with read request r' that was satisfied from a write $w' \neq w$ to the same address, if w' is not from a po-successor of i,restart i' and its data-flow dependents.

Action:

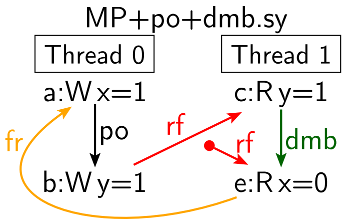
1. record w as propagated; and
2. update the memory with w; and
3. (CO) restart any speculative instructions which have violated coherence as a result of this.
   I.e., for every non-finished instruction i' po-after i with read request r' that was satisfied from a write $w' \neq w$ to the same address, if w' is not from a po-successor of i,restart i' and its data-flow dependents.

CoWR

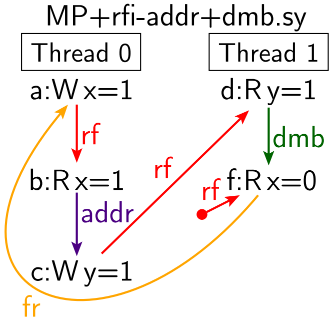
$$\begin{tikzcd}
\boxed{\text{Thread 0}} & \boxed{\text{Thread 1}} \\
\text{a:W x=1} \arrow[r, "\text{co}"] \arrow[dr, "\text{rf}"] & \text{b:W x=2} \arrow[d, "\text{po}"] \\
 & \text{c:R x=1} \arrow[u, bend left, "\text{fr}"]
\end{tikzcd}$$

(Forbidden. If c is satisfied from a before b is propagated, once b propagates it restarts c.)

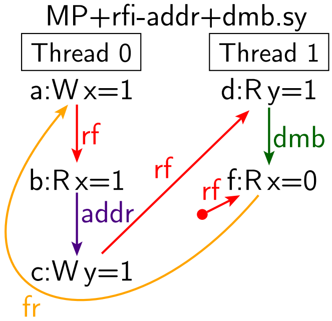
Example


MP+po+dmb.sy
Thread 0
Thread 1
a:W x=1
c:R y=1
po
rf
fr
rf
dmb
b:W y=1
e:R x=0

Example


MP+rfi-addr+dmb.sy
Thread 0
Thread 1
a:W x=1
d:R y=1
rf
dmb
b:R x=1
rf
rf
f:R x=0
addr
c:W y=1
fr

Example: write forwarding



(Allowed. *b* can see *a* before *a* is propagated to other threads, resolve the address dependency and allow *c* to propagate before *a*.)

Satisfy read by forwarding

Condition:

A load instruction instance *i* in state Pending_mem_reads *read_cont* with unsatisfied read request *r* in *i.mem_reads* can satisfy *r* by forwarding an unpropagated write by a program-order earlier store instruction instance, if the *read-request-condition* predicate holds. This is if:

1. (BO) all po-previous `dmb sy` instructions are finished.

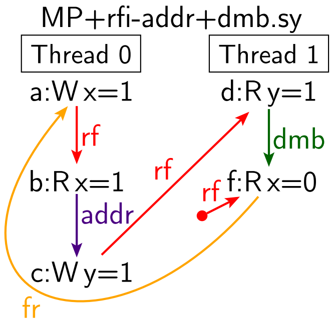
Satisfy read by forwarding

Let *w* be the most-recent write from a store instruction instance po-before *i*, to the address of *r*, and which is not superseded by an intervening store that has been propagated or read from by this thread. That last condition requires:

- (CO) that there is no store instruction po-between *i* and *i′* with a same-address write, and
- (CO) that there is no load instruction po-between *i* and *i′* that was satisfied by a same-address write from a different thread.

Action: Apply the action of Satisfy read in memory.

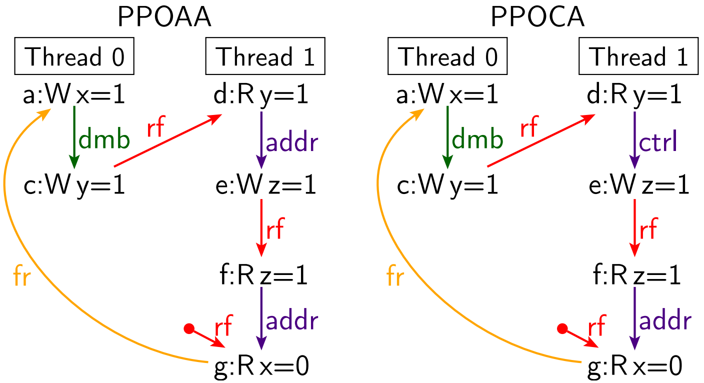
Example: write forwarding



rmem web UI

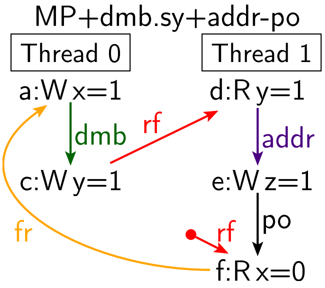
(Allowed. *b* can see *a* before *a* is propagated to other threads, resolve the address dependency and allow *c* to propagate before *a*.)

Write forwarding again



PPOCA rmem web UI

Non-dependent register re-use does not create ordering



rmem web UI

Axiomatic Models

- Operational: define abstract machine, with states and transitions
- Axiomatic: define allowed/forbidden predicate on candidate executions

Why two styles of definition?

Operational:

- more concrete hardware intuition (for abst.microarch.op.)
- builds valid executions incrementally
- SOTA includes mixed-size support, ISA integration, ELF support
- more complex

Axiomatic:

- more abstract
- global properties of full executions (but only those; not incremental)
- pure memory model
- more concise

Candidate Executions

Consider a single candidate execution, and focus just on its read and write events. Give them IDs $a, b, \ldots$ (unique within an execution): $a : t : \mathrm{R}\, x{=}n$ and $a : t : \mathrm{W}\, x{=}n$.

Say a *candidate pre-execution* E consists of

- a finite set E of such events
- *program order* (*po*), an irreflexive transitive relation over E
  [intuitively, from a control-flow unfolding and choice of arbitrary memory read values of the source program]
- subrelations of *po* identifying events related by dependencies or separated by barriers, *addr*, *data*, *ctrl*, *dmb*, etc.

Say a *candidate execution* consists of that together with

- *reads-from* (*rf*), a relation over E relating writes to the reads that read from them (with same address and value)
  [note this is intensional: it identifies *which write*, not just the value]
- *coherence* (*co*), an irreflexve transitive relation over E relating only writes that are to the same address; total when restricted to the writes of each address separately
  [intuitively, the hardware coherence order for each address]

Axiomatic models in Herd syntax

Define auxiliary relations, mostly with standard relational algebra:

- from-reads (fr):

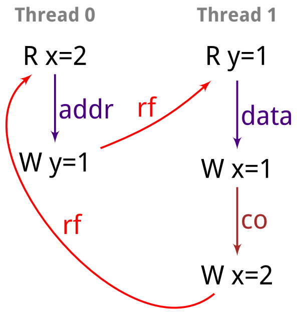
$$r \xrightarrow{\text{fr}} w \quad \text{iff} \quad (\exists w_0.\ w_0 \xrightarrow{\text{co}} w \quad \wedge \quad w_0 \xrightarrow{\text{rf}} r) \quad \vee$$
$$(\neg\exists w_0.\ w_0 \xrightarrow{\text{rf}} r)$$

- internal (same-thread) and external (different-thread) subrelations of rf, co, fr: rfi/rfe etc.
- relation union: r1 | r2
- relation composition: r1 ; r2
- identity relation on particular kinds of events: [W]

Require that particular relations are acyclic, irreflexive, or empty (these are the "axioms" of an axiomatic model. Not to be confused with "axiomatic PL semantics).

Official axiomatic model

```
(* Observed-by *)
let obs = rfe | rfe | fre | coe

(* Dependency-ordered-before *)
let dob = addr | data
        | ctrl; [W]
        | (ctrl | (addr; po)); [ISB]; po; [R]
        | addr; po; [W]
        | (ctrl | data); coi
        | (addr | data); rfi

(* Atomic-ordered-before *)
let aob = rmw
        | [range(rmw)]; rfi; [A | Q]

(* Barrier-ordered-before *)
let bob = po; [dmb.full]; po
        | [L]; po; [A]
        | [R]; po; [dmb.ld]; po
        | [A | Q]; po
        | [W]; po; [dmb.st]; po; [W]
        | po; [L]
        | po; [L]; coi

(* Ordered-before *)
let ob = (obs | dob | aob | bob)+

acyclic po-loc|fr|co|rf       as internal
irreflexive ob                as external
empty rmw & (fre; coe)        as atomic
```



Herd

Alglave + Maranget

http://diy.inria.fr/doc/herd.html

Operational-Axiomatic Correspondence (Pulte thesis)

loads

- fetch
- initiate-memory-read (footprint known)
- satisfy-read by-forwarding (from po-predecessor write)
- satisfy-read-from-memory
- complete-load (all reads satisfied)
- finish

stores

- fetch
- announce-write-footprint
- initiate-memory-write (data known)
- commit-store
- propagate-memory-write
- complete-store
- finish

barriers

- fetch
- commit-barrier
- finish

Operational-Axiomatic Correspondence (Pulte thesis)

loads

- fetch
- initiate-memory-read (footprint known)
- satisfy-read by-forwarding (from po-predecessor write)
- satisfy-read-from-memory
- complete-load (all reads satisfied)
- finish

stores

- fetch
- announce-write-footprint
- initiate-memory-write (data known)
- commit-store
- propagate-memory-write
- complete-store
- finish

barriers

- fetch
- commit-barrier
- finish

Under this correspondence the relations of ARMv8-ax can be viewed as describing the order of transitions in an ARMv8-op trace for a given execution:

Theorem (Pulte)

Let $x = (po, co, rf, rmw)$ be a finite candidate execution of ARMv8-axiomatic for a given program P. The execution x is valid under ARMv8-axiomatic if and only if there exists a valid finite trace t of ARMv8-operational for the program P such that $(po_t, co_t, rf_t, rmw_t) = (po, co, rf, rmw)$.

(here $\mathtt{rf}_t$ etc. are relations extracted from the operational trace t)

Back to IBM POWER

There the operational model has a more complex storage subsystem state: for each hardware thread, a list of the writes and barriers propagated *to that thread*.

Omitted

- some other “exotic” phenomena: might-access-same-address etc.
- mixed-size effects
- system semantics – e.g. instruction fetch and i/d cache maintenance