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

November 8, 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**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR X0,[X1] R y=1</td>
<td>STR X0,[X2] R x=0</td>
</tr>
<tr>
<td>LDR X0,[X1]</td>
<td>LDR X2,[X3]</td>
</tr>
</tbody>
</table>

**AArch64**

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR X0,[X1]</td>
<td>STR X0,[X2]</td>
</tr>
<tr>
<td>LDR X0,[X1]</td>
<td>LDR X2,[X3]</td>
</tr>
</tbody>
</table>

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

Allowed: 1:X0=1; 1:X2=0;
Message Passing (MP) Again

**Thread 0**
- a: \(W x=1\)
- b: \(W y=1\)

**Thread 1**
- c: \(R y=1\)
- d: \(R x=0\)

---

**MP**

<table>
<thead>
<tr>
<th>PowerARM</th>
<th>POWER</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kind</td>
<td>PowerG5</td>
<td>Power6</td>
</tr>
<tr>
<td>MP</td>
<td>Allow</td>
<td>10M/4.9G</td>
</tr>
</tbody>
</table>
Message Passing (MP) Again

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

Thread 0

a: W x=1
b: R y=0

Thread 1

c: W y=1
d: R x=0

SB

AArch64

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR X0, [X1] // a</td>
<td>STR X0, [X1] // c</td>
</tr>
<tr>
<td>LDR X2, [X3] // b</td>
<td>LDR X2, [X3] // d</td>
</tr>
</tbody>
</table>

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

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

- \( a: R x=1 \)
- \( b: W x=1 \)

CoWR0

- \( a: W x=1 \)
- \( b: R x=0 \)

CoWW

- \( a: W x=1 \)
- \( b: W x=2 \)

CoRW2

- **Thread 0**
  - \( a: W x=1 \)
  - \( b: R x=1 \)
  - \( c: W x=2 \)
- **Thread 1**
  - \( a: W x=1 \)
  - \( b: R x=1 \)

CoWR

- **Thread 0**
  - \( a: W x=1 \)
  - \( b: W x=2 \)
  - \( c: R x=1 \)
- **Thread 1**
  - \( a: W x=1 \)
  - \( b: R x=2 \)

CoRR

- **Thread 0**
  - \( a: W x=1 \)
  - \( b: R x=1 \)
  - \( c: R x=0 \)
- **Thread 1**
  - \( a: W x=1 \)
  - \( b: R x=1 \)

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 $\mathcal{C}$ 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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: W x=1</td>
<td>d: R y=1</td>
</tr>
<tr>
<td>c: W y=1</td>
<td>f: R x=0</td>
</tr>
</tbody>
</table>

MP+dmbs.sys AArch64

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR X0, [X1]</td>
<td>LDR X0, [X1]</td>
</tr>
<tr>
<td>DMB SY</td>
<td>DMB SY</td>
</tr>
<tr>
<td>STR X0, [X2]</td>
<td>LDR X2, [X3]</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: W x=1</td>
<td>d: R y=1</td>
</tr>
<tr>
<td>STR X0,[X1]</td>
<td>EOR X2,X0,X0</td>
</tr>
<tr>
<td>STR X0,[X2]</td>
<td>LDR X3,[X4,X2]</td>
</tr>
<tr>
<td>LDR X0,[X1]</td>
<td></td>
</tr>
<tr>
<td>EOR X2,X0,X0</td>
<td></td>
</tr>
<tr>
<td>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;</td>
<td></td>
</tr>
</tbody>
</table>
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)

**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.**

| 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; |
| 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)

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)

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:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: R x=1</td>
<td>c: R y=1</td>
</tr>
<tr>
<td>b: W y=1</td>
<td>d: W x=1</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>a: R x=1</td>
<td>c: R y=1</td>
</tr>
<tr>
<td>b: W y=1</td>
<td>d: W x=1</td>
</tr>
</tbody>
</table>

Thread 0:

1. LDR X0, [X1]
2. R x=1
3. STR X2, [X3]
4. W y=1

Thread 1:

1. LDR X0, [X1]
2. R y=1
3. STR X2, [X3]
4. W x=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:

<table>
<thead>
<tr>
<th>Kind</th>
<th>POWER</th>
<th>ARM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Power5</td>
<td>Power6</td>
</tr>
<tr>
<td>LB</td>
<td>Allow</td>
<td>0/7.4G</td>
</tr>
<tr>
<td>LB+addr</td>
<td>Forbid</td>
<td>0/6.9G</td>
</tr>
<tr>
<td>LB+datas</td>
<td>Forbid</td>
<td>0/6.9G</td>
</tr>
<tr>
<td>LB+ctrls</td>
<td>Forbid</td>
<td>0/4.5G</td>
</tr>
</tbody>
</table>
LB+datas – thin-air values?

Thread 0
a:Rx=1
b:Wy=1

Thread 1
c:Ry=1
d:Wx=1

r1=x
y=r1

r2=y
x=r2
LB+datas – thin-air values?

Forbidden!
Iterated Message Passing and Cumulative Barriers

<table>
<thead>
<tr>
<th>WRC-loop</th>
<th>Pseudocode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>x=1</td>
<td>while (x==0) {}</td>
</tr>
<tr>
<td></td>
<td>y=1</td>
</tr>
<tr>
<td>Initial state: x=0 ∧ y=0</td>
<td></td>
</tr>
<tr>
<td>Forbidden?: 2:r3=0</td>
<td></td>
</tr>
</tbody>
</table>
Iterated Message Passing and Cumulative Barriers

![Diagram showing message passing between threads]

- **Thread 0**
  - a: $W \times = 1$

- **Thread 1**
  - b: $R \times = 1$
  - c: $W y = 1$

- **Thread 2**
  - d: $R y = 1$
  - e: $R \times = 0$

### 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...

<table>
<thead>
<tr>
<th>WRC</th>
<th>AArch64</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td>Thread 1</td>
</tr>
<tr>
<td>STR X0, [X1] // a</td>
<td>LDR X0, [X1] // b</td>
</tr>
<tr>
<td>STR X2, [X3] // c</td>
<td>LDR X2, [X3] // e</td>
</tr>
</tbody>
</table>

---

Thread 0 uses **rf** and **fr** dependences to coordinate with Thread 1 and Thread 2.
Iterated Message Passing and Cumulative Barriers

Thread 0
a: W x=1

Thread 1
b: R x=1

c: W y=1

d: R y=1

e: R x=0

Thread 2

WRC+addrs

<table>
<thead>
<tr>
<th>Thread 0</th>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>STR X0, [X1] //a</td>
<td>LDR X0, [X1] //b</td>
<td>LDR X0, [X1] //d</td>
</tr>
<tr>
<td>EOR X2, X0, X0</td>
<td>EOR X2, X0, X0</td>
<td></td>
</tr>
<tr>
<td>STR X3, [X4, X2] //c</td>
<td>LDR X3, [X4, X2] //e</td>
<td></td>
</tr>
</tbody>
</table>

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
Thread 0: \( a: W \ x = 1 \)
Thread 1: \( b: R \ x = 1 \)
Thread 2: \( d: W \ y = 1 \)
Thread 3: \( e: R \ y = 1 \)

\[ \text{Initial state: } 0: X_1 = x; 0: X_0 = 1; 1: X_4 = y; 1: X_1 = x; 1: X_0 = 0; 1: X_3 = 0; 2: X_1 = y; 2: X_0 = 1; 3: X_4 = x; 3: X_1 = y; 3: X_0 = 0; 3: X_3 = 0; y = 0; x = 0; \]

Forbidden: \( 1: X_0 = 1; 1: X_3 = 0; 3: X_0 = 1; 3: X_3 = 0; \)

Likewise.

- x86, current ARMv8-A, RISC-V: \textit{(other) multicopy atomic}
- IBM POWER, old ARMv8-A, ARMv7-A: \textit{non-multicopy-atomic}