



FDD2034

Research Preparation Course

Module 2, lecture 3, 2026

Weak Memory Models

Mads Dam

Includes material due to Pablo Buiras

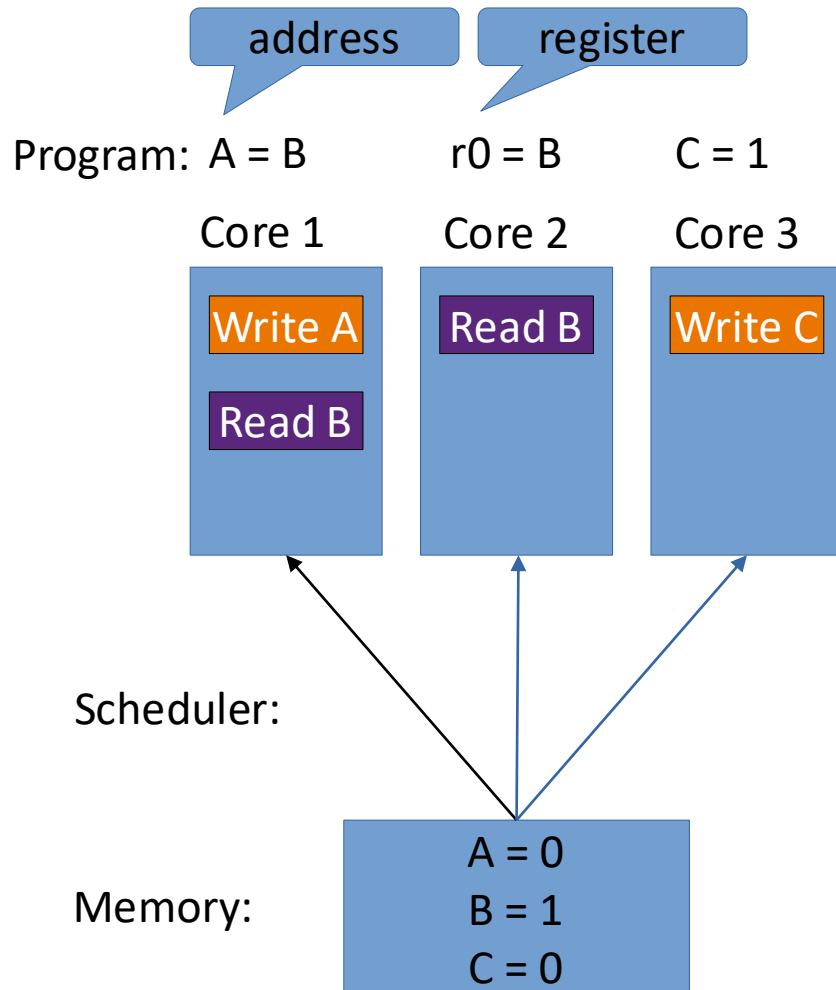
Today's Topics

- Sequential Consistency (Lamport)
 - All operations execute in some sequential order
 - Each thread's operations happen in program order
- Memory Models
 - Memory in multicore is complex
 - Different threads may “see” different contents of memory at the “same time”
 - How does this work?
 - How is it controlled?

Hardware Memory Models

- Multiprocessors reorder memory operations in unintuitive, scary ways
 - All in the interest of performance
 - Application programmers rarely see this behaviour
 - Kernel developers see it all the time
 - Basic principle:
 1. Respect dependencies
 2. Possibly reorder or speculate, subject to 1.
- Examples: x86, ARM, RISC-V, SPARC

Sequential Consistency Machine Model



- “Switch” model
- Instructions are executed atomically
- There is a scheduler that selects between the available operations
- Processors issue ops in program order
- Any interleaving of threads is possible

Memory Consistency Model

- Defines what reorderings of operations are allowed
- Examples:
 - HW: pipelining, out-of-order
 - Compiler: register allocation, loop transformations
- SC is the ‘strongest’ model
 - Allows fewest reorderings
 - Pros: Intuitive, easy to reason about
 - Cons: Performance cost, “switch” model too conservative

SC Reorderings

- SC: Partial order on memory requests $A \rightarrow B$
 - If P issues two requests A and B and A is issued before B in program order, then $A \rightarrow B$
 - A and B are issued to the same variable, and A is issued first, then $A \rightarrow B$ (on all processors)
- These partial orders can be interleaved, define a total order
- Many total orders are sequentially consistent!

Example

Recall: Same thread, or same variable

Program:

- P1: W(a), R(b), W(c)
- P2: R(a), W(a), R(b)

Q: Are the following schedules (total orders) sequentially consistent?

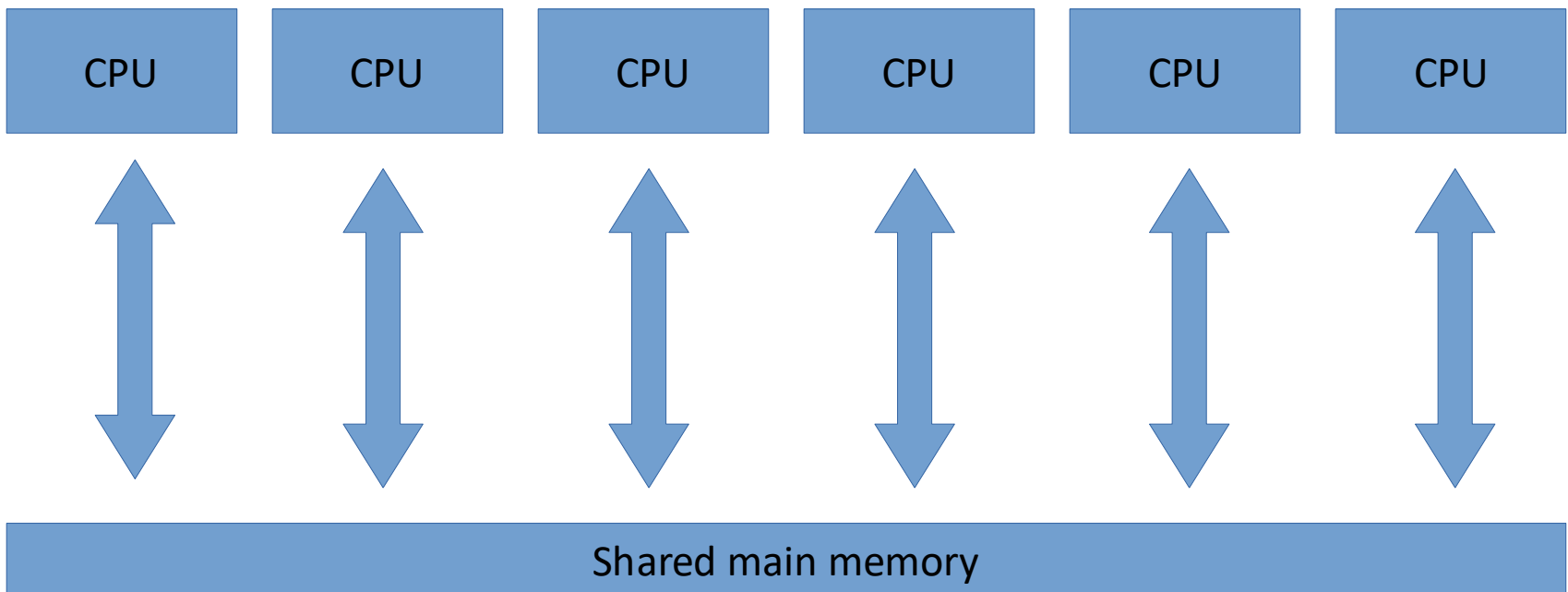
- 1. P1:W(a), P2:R(a), P2:W(a), P1:R(b), P2:R(b), P1:W(c)
- 2. P1:W(a), P2:R(a), P1:R(b), P2:R(b), P1:W(c), P2:W(a)
- 3. P2:R(a), P2:W(a), P1:R(b), P1:W(a), P1:W(c), P2:R(b)

Issues with SC

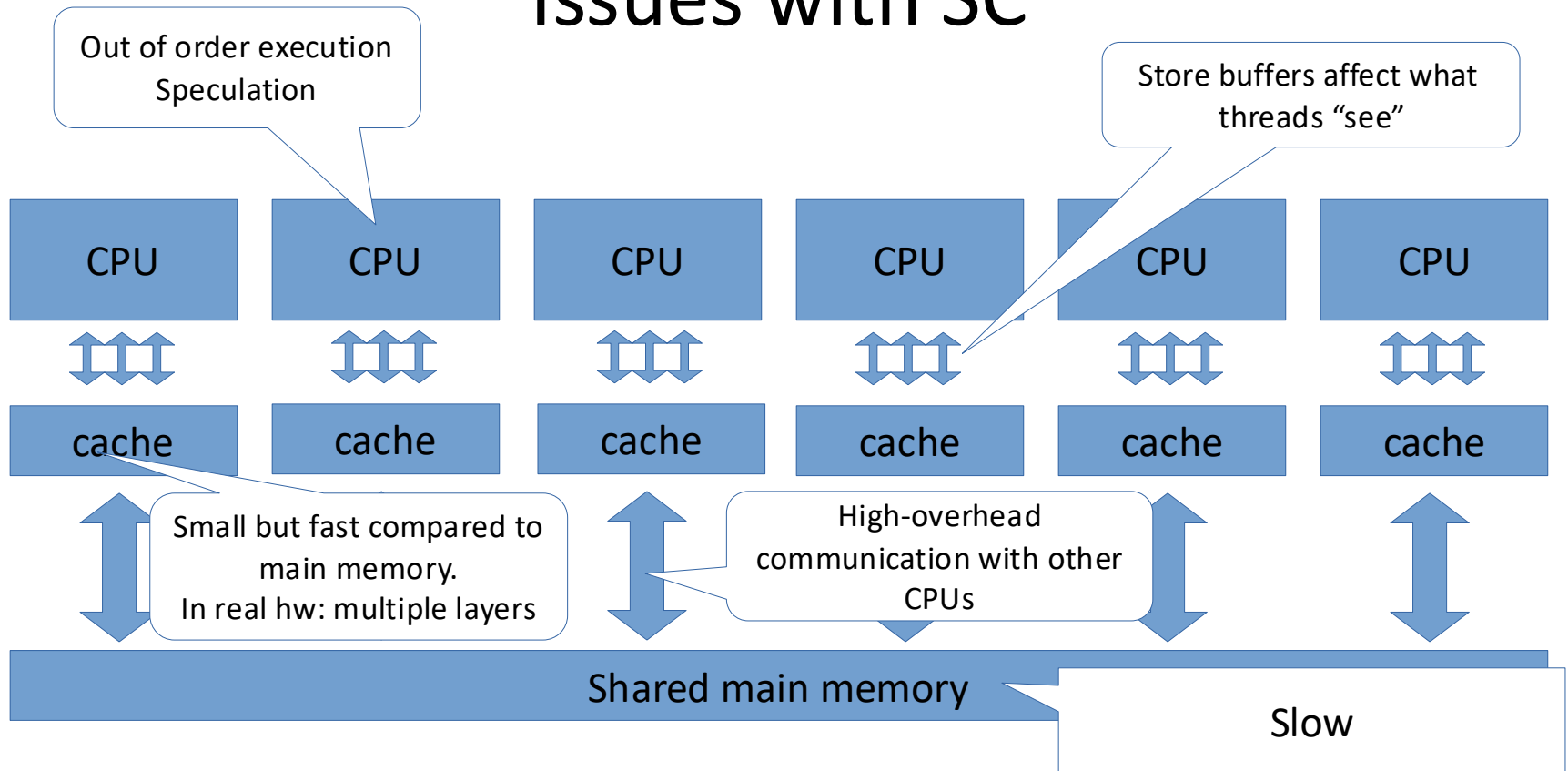
- Gets in the way of compiler optimisations
 - Transformation on the right should preserve semantics
 - Optimisation breaks “program order”
 - Note that this is problematic in a shared memory setting if X and Y are visible to other processors
 - We will return to this point later
- $X = 1$
 - $Y = 2$
 - $Z = X + Y$
- 
- $Y = 2$
 - $X = 1$
 - $Z = X + Y$

Issues with SC

Simplified multiprocessor machine model:



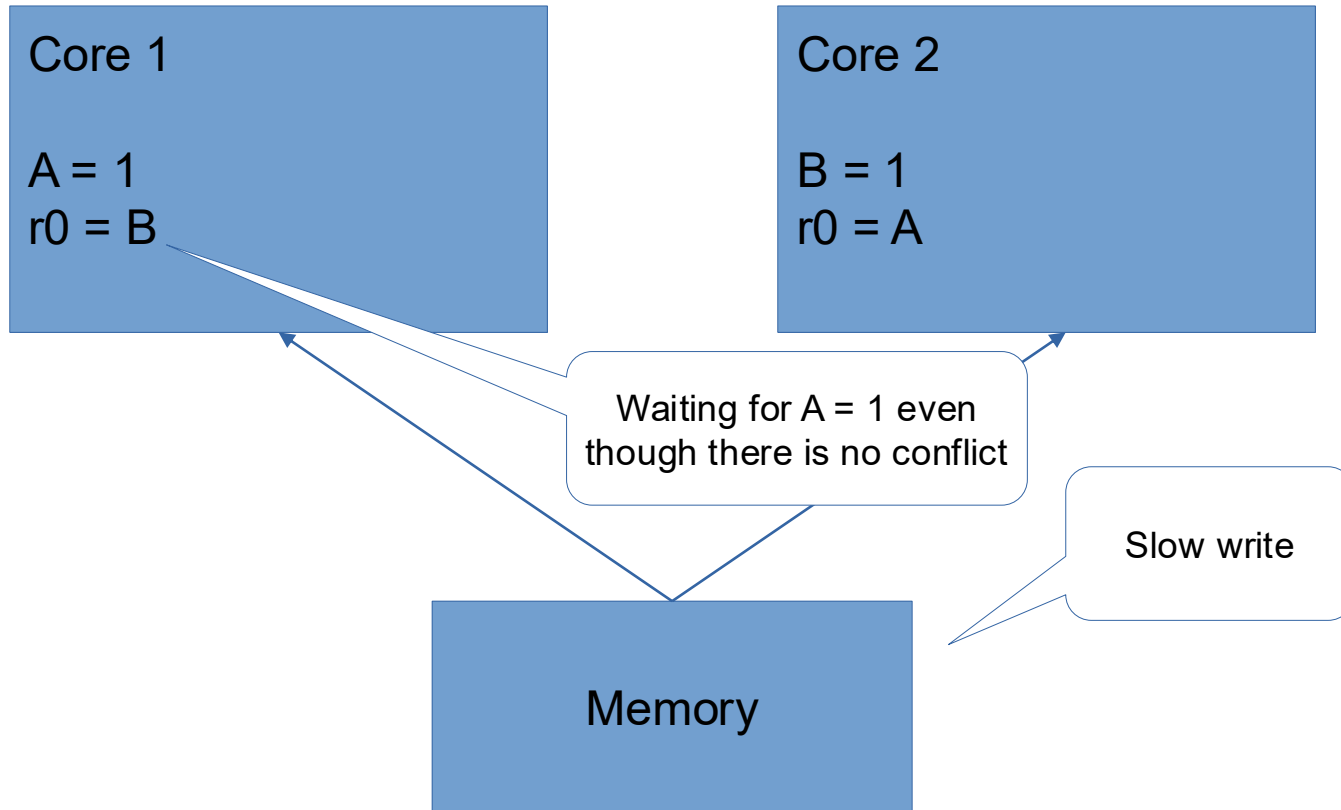
Issues with SC



Problem: SC causes too much cache synchronisation!

If all CPUs are to see the latest value, must push all writes through slow channels

Problem: Memory Latency



- May have a wait 100's of cycles if A is not in the cache

Optimisation: Store Buffers

- Writes are stored in a local buffer and the processor then proceeds to next instruction immediately (first-in–first-out)
- The cache will pull writes out of the write buffer when it's ready

A = 1
r0 = B

Core

A = 1

Write buffer

Cache

A = 0, B = 0

Memory

A = 0

B = 0

Optimisation: Store Buffers

- Writes are stored in a local buffer and the processor then proceeds to next instruction immediately (first-in–first-out)
- The cache will pull writes out of the write buffer when it's ready

A = 1
r0 = B

Core

r0 = B

Write buffer

A = 1

Cache

A = 0, B = 0

Memory

A = 0

B = 0

Optimisation: Store Buffers

- Writes are stored in a local buffer and the processor then proceeds to next instruction immediately (first-in–first-out)
- The cache will pull writes out of the write buffer when it's ready

A = 1
r0 = B

Core

r0 = B

Write buffer

Cache

A = 1, B = 0

Memory

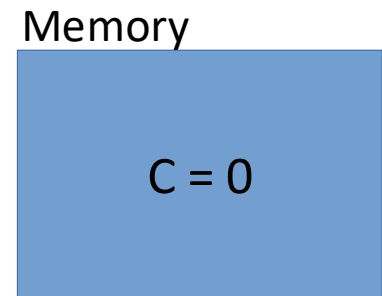
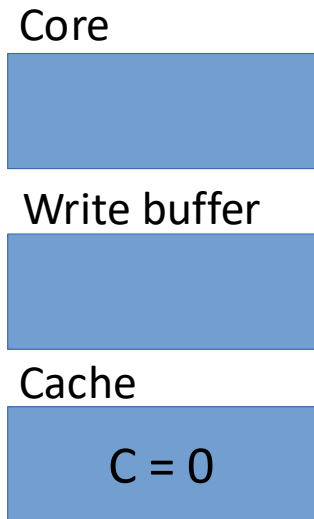
A = 0

B = 0

Optimisation: Store Forward

- Writes are stored in a local buffer and the processor then proceeds to next instruction immediately (first-in–first-out)
- The cache will pull writes out of the write buffer when it's ready

$C = 1$
 $r0 = C$



Optimisation: Store Forward

- Store writes in a local buffer and then proceeds to next instruction immediately (first-in–first-out)
- The cache will pull writes out of the write buffer when it's ready

$C = 1$
 $r0 = C$

Core

$C = 1$

Write buffer

Cache

$C = 0$

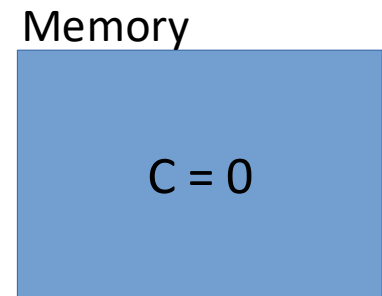
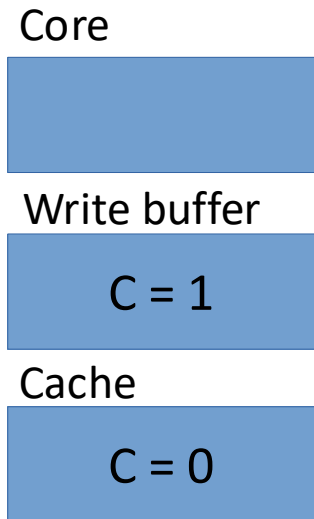
Memory

$C = 0$

Optimisation: Store Forward

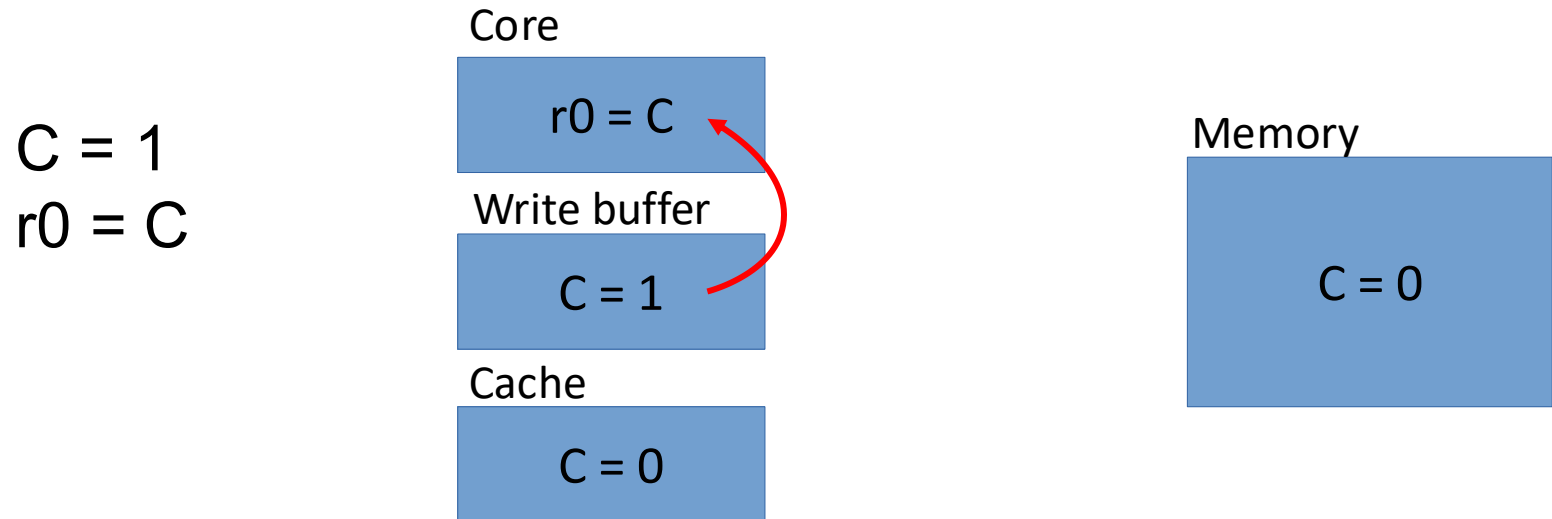
- Store writes in a local buffer and then proceeds to next instruction immediately (first-in–first-out)
- The cache will pull writes out of the write buffer when it's ready

$C = 1$
 $r0 = C$



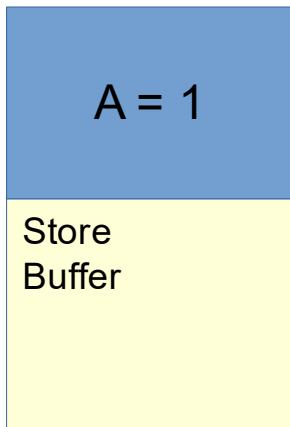
Optimisation: Store Forward

- Store writes in a local buffer and then proceeds to next instruction immediately (first-in–first-out)
- The cache will pull writes out of the write buffer when it's ready

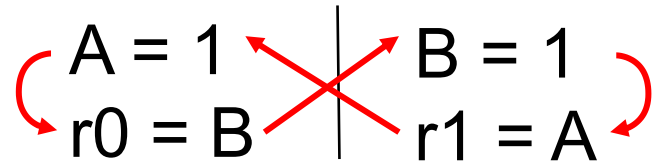
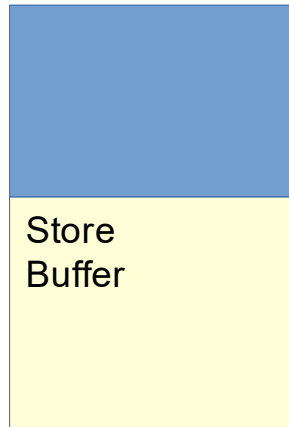


Store Buffers and SC

Core 1



Core 2

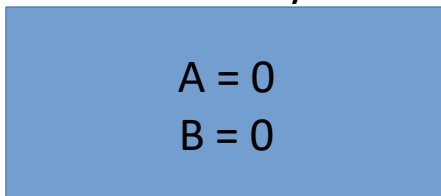


Is it possible for $r0 = 0$ and $r1 = 0$?

Not under SC!

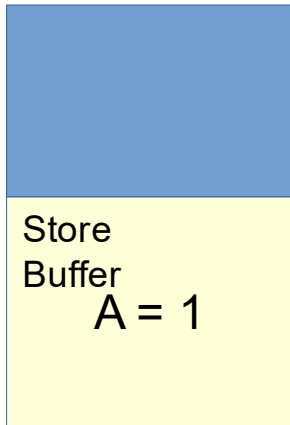
However, it can happen if we use store buffers

Memory

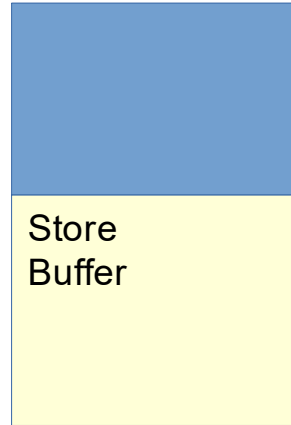


Store Buffers and SC

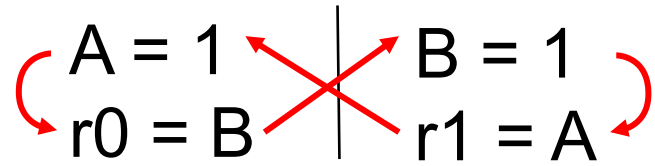
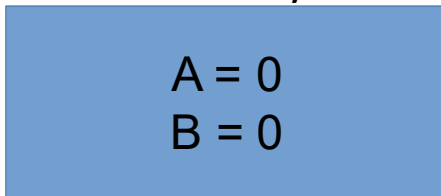
Core 1



Core 2



Memory



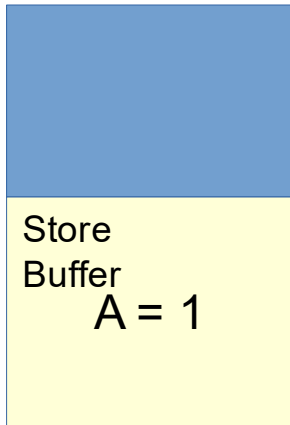
Is it possible for $r0 = 0$ and $r1 = 0$?

Not under SC!

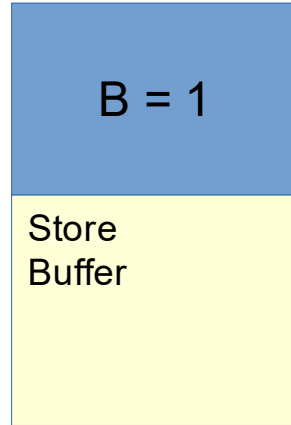
However, it can happen if we use store buffers

Store Buffers and SC

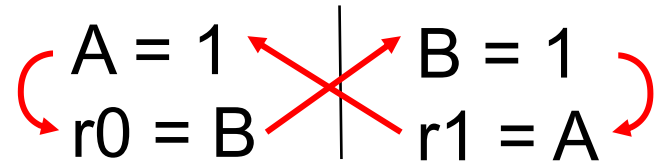
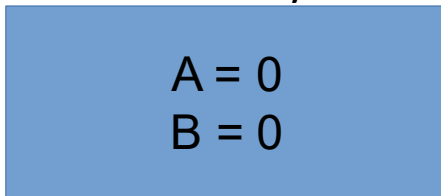
Core 1



Core 2



Memory



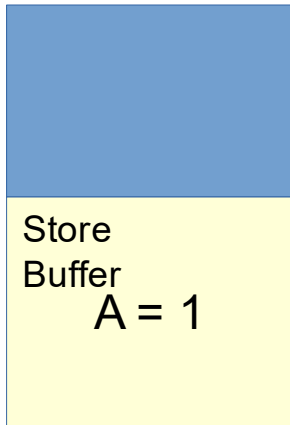
Is it possible for $r0 = 0$ and $r1 = 0$?

Not under SC!

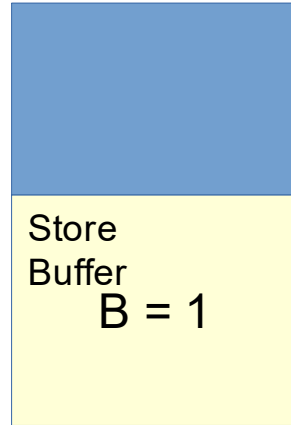
However, it can happen if we use store buffers

Store Buffers and SC

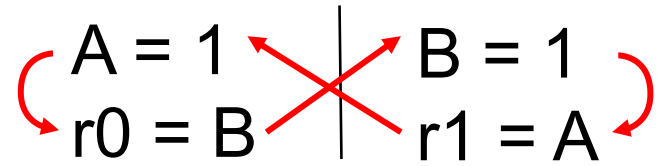
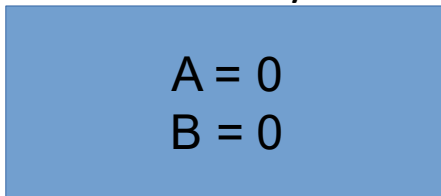
Core 1



Core 2



Memory



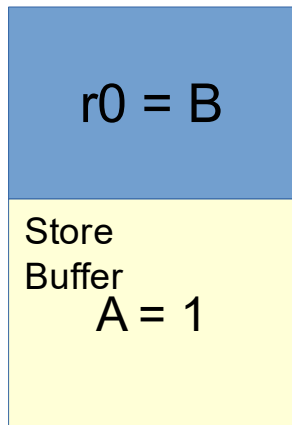
Is it possible for $r0 = 0$ and $r1 = 0$?

Not under SC!

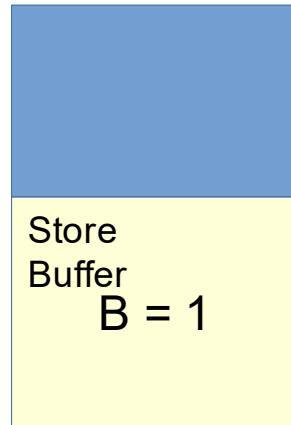
However, it can happen if we use store buffers

Store Buffers and SC

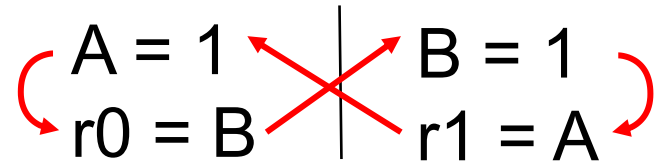
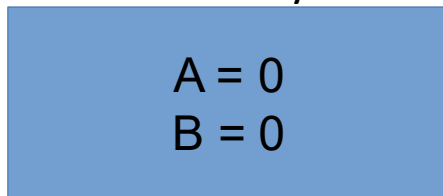
Core 1



Core 2



Memory



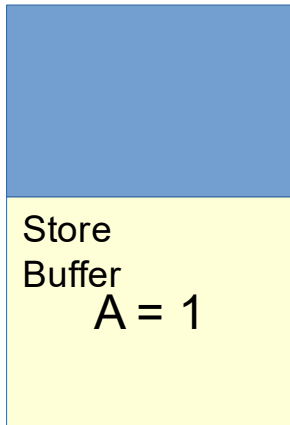
Is it possible for $r0 = 0$ and $r1 = 0$?

Not under SC!

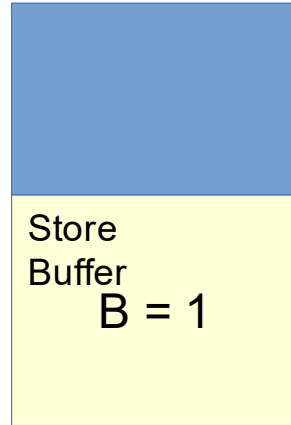
However, it can happen if we use store buffers

Store Buffers and SC

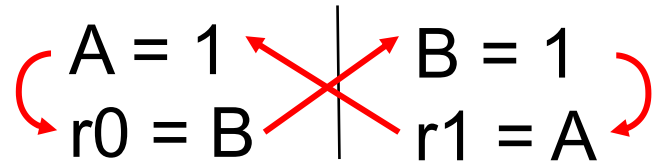
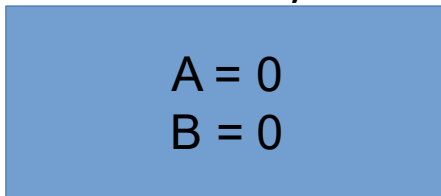
Core 1



Core 2



Memory



Is it possible for $r0 = 0$ and $r1 = 0$?

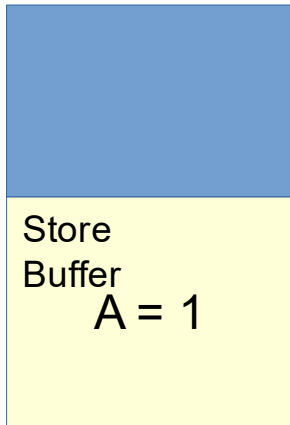
Not under SC!

However, it can happen if we use store buffers

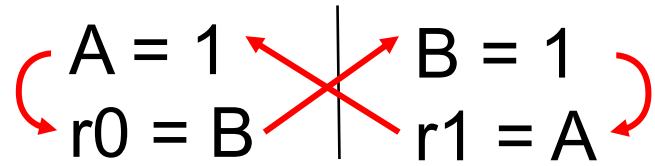
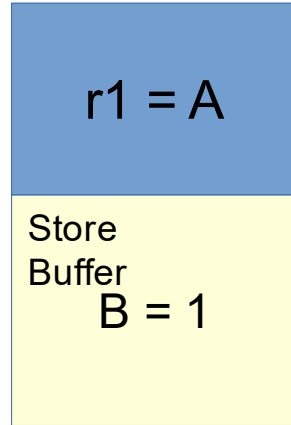
$r0 = 0$

Store Buffers and SC

Core 1



Core 2

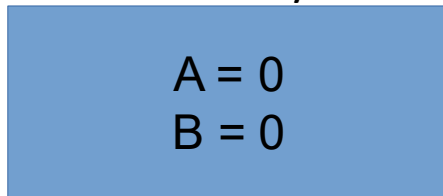


Is it possible for $r0 = 0$ and $r1 = 0$?

Not under SC!

However, it can happen if we use store buffers

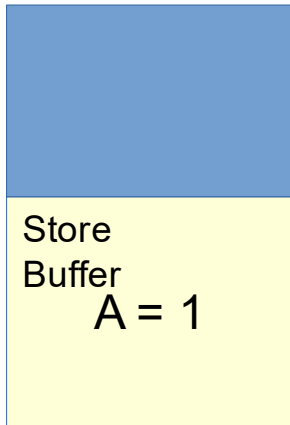
Memory



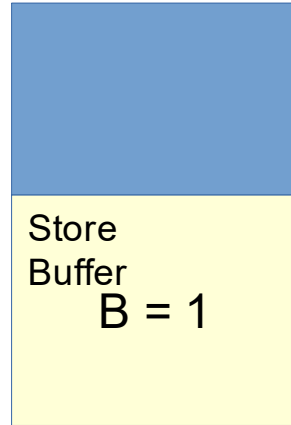
$r0 = 0$

Store Buffers and SC

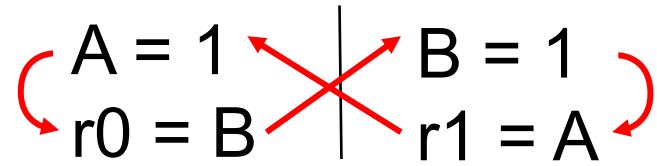
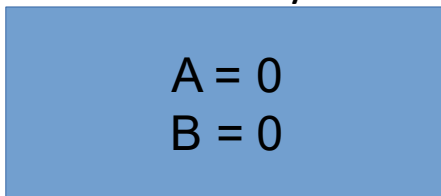
Core 1



Core 2



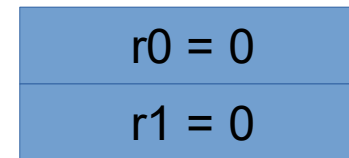
Memory



Is it possible for $r0 = 0$ and $r1 = 0$?

Not under SC!

However, it can happen if we use store buffers

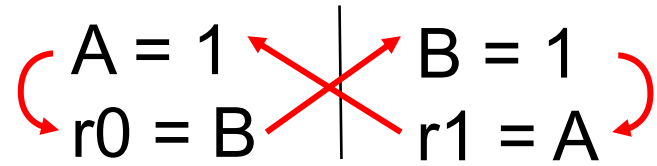
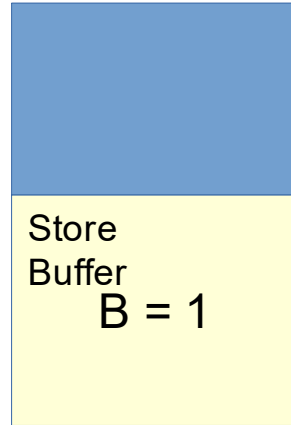


Store Buffers and SC

Core 1



Core 2

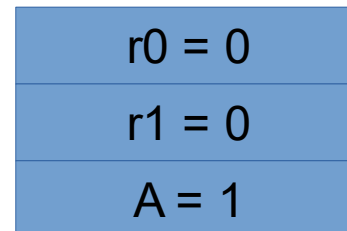
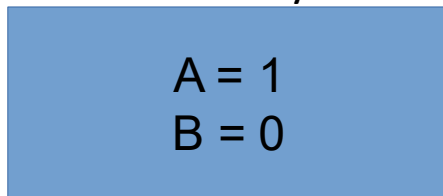


Is it possible for $r0 = 0$ and $r1 = 0$?

Not under SC!

However, it can happen if we use store buffers

Memory

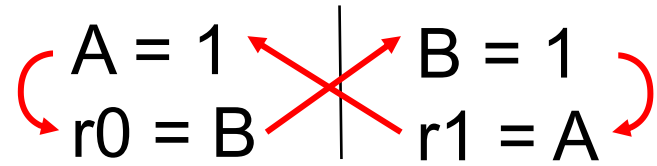
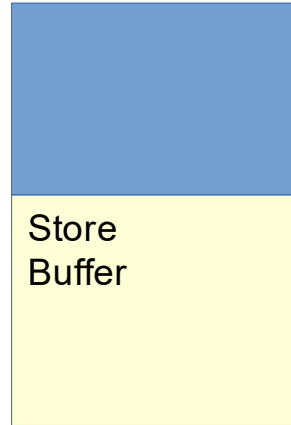


Store Buffers and SC

Core 1



Core 2

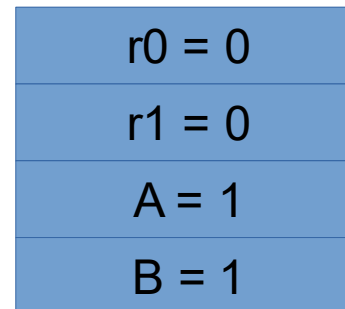
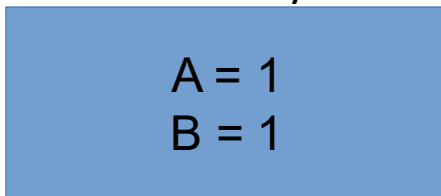


Is it possible for $r0 = 0$ and $r1 = 0$?

Not under SC!

However, it can happen if we use store buffers

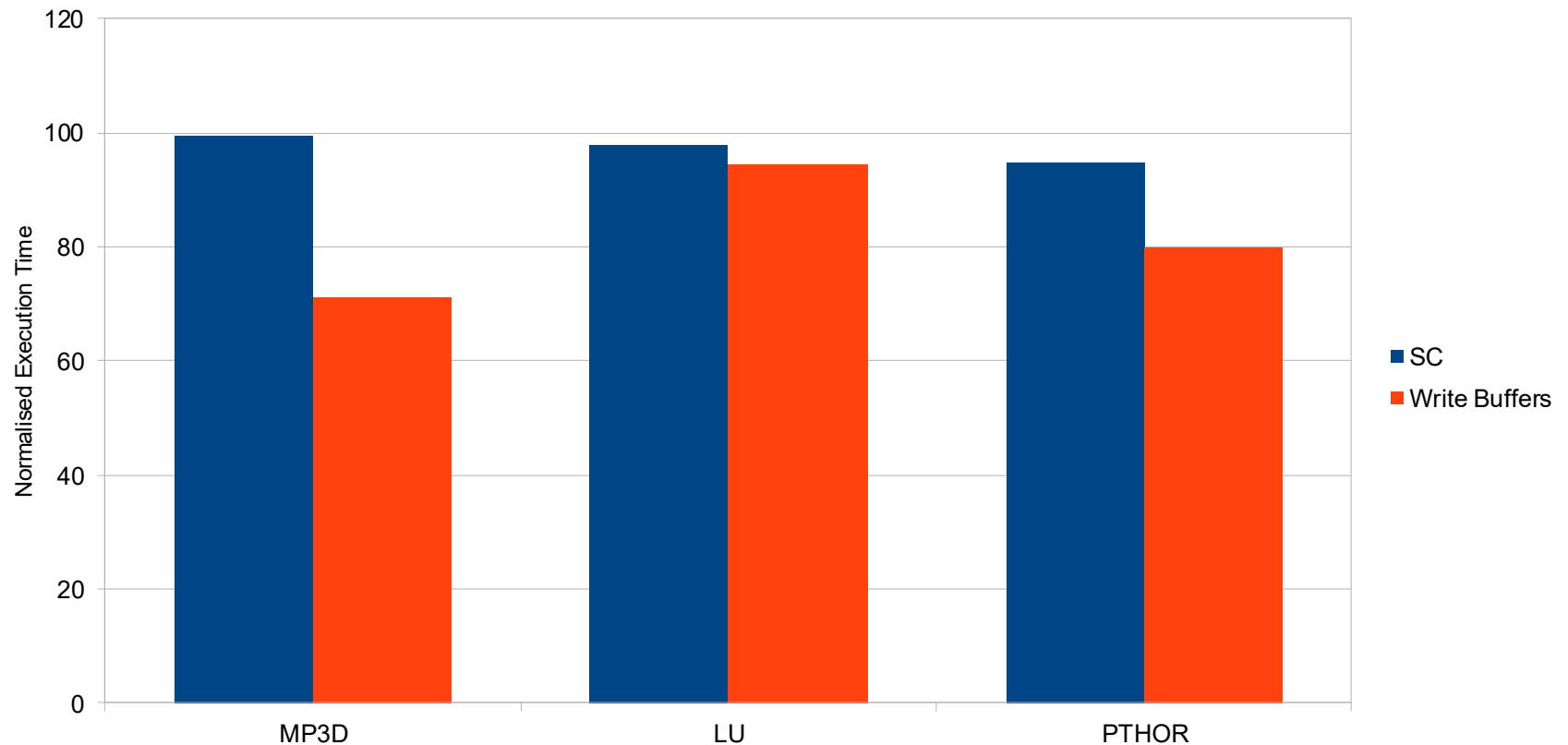
Memory



Total Store Ordering (TSO)

- Machine: SC + store buffers
- “Axiomatically”: allow reorderings that break $W \rightarrow R$
- Allows more behaviours than SC
 - Harder to program!
- Implemented in x86
- All modern processors use store buffers, so all hardware models are at least as weak as TSO (often weaker)

Is it worth it?



Source:

Gharachorloo, Gupta, Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. ASPLOS 1991.

Axiomatic Description

Program order $<_p$:

- Order in which stores and loads appear in the program code

Memory order $<_m$:

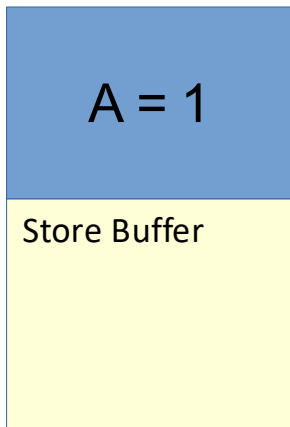
- Order in which stores and loads become effective/visible in “main memory”

TSO Ordering conditions:

- $(R \rightarrow R) \ l_1 <_p l_2 \Rightarrow l_1 <_m l_2$
- $(R \rightarrow W) \ l_1 <_p s_2 \Rightarrow l_1 <_m s_2$
- $(W \rightarrow W) \ s_1 <_p s_2 \Rightarrow s_1 <_m s_2$
- Latest write wins: $\text{val}(l_1) = \text{val}(\max_{<_m} \{s_1 \mid s_1 <_m l_1 \vee s_1 <_p l_1\})$
 - Needed for store forwarding

Another Example

Core 1



Core 2



$A = 1$

$r0 = A$

$r1 = B$

$B = 1$

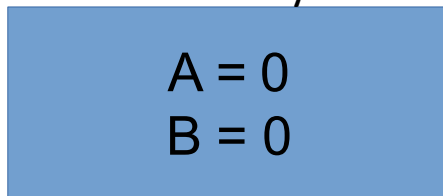
$r2 = B$

$r3 = A$

Is $(r0, r1, r2, r3) = (0, 1, 0, 1)$ possible?

Is $(r0, r1, r2, r3) = (1, 0, 1, 0)$ possible?

Memory



SC vs TSO

Sequential Consistency

- $R \rightarrow R$
- $R \rightarrow W$
- $W \rightarrow W$
- $W \rightarrow R$

TSO

- $R \rightarrow R$
- $R \rightarrow W$
- $W \rightarrow W$

Synchronisation instructions - Fence

- Usual RMW instructions: CAS, TAS, LL/SC
- A **barrier/fence** instruction forces all PO-prior memory operations to complete before any PO-succeeding memory operations can begin

Axiomatically:

- (Store $\rightarrow$ FENCE)
 - $S_1 <_p \text{FENCE} \Rightarrow S_1 <_m \text{FENCE}$
- (FENCE $\rightarrow$ Load)
 - $\text{FENCE} <_p L_1 \Rightarrow \text{FENCE} <_m L_1$

A = 1
fence
r0 = B

B = 1
fence
r1 = A

- Effectively reinstates sequential consistency at a particular point in program execution
- To be used sparingly

Memory models for x86

Problem:

- Regarding concurrency, most existing instruction set architecture (ISA) spec's are imprecise, ambiguous, and sometimes even unsound
- Accordingly, writing low level concurrent code is bit of a black art

The x86-TSO programmer's model [Sewell et al 2010]

- Mathematically precise, formal model matching x86 intentions
- Rigorously defined in theorem prover HOL4
- Provides intuitive abstract machine, accessible to programmers
- Memory model + semantics for machine instructions enables formal reasoning about program behaviour

How is this useful?

- Guides intuition of systems programmers that develop software for multithreaded systems

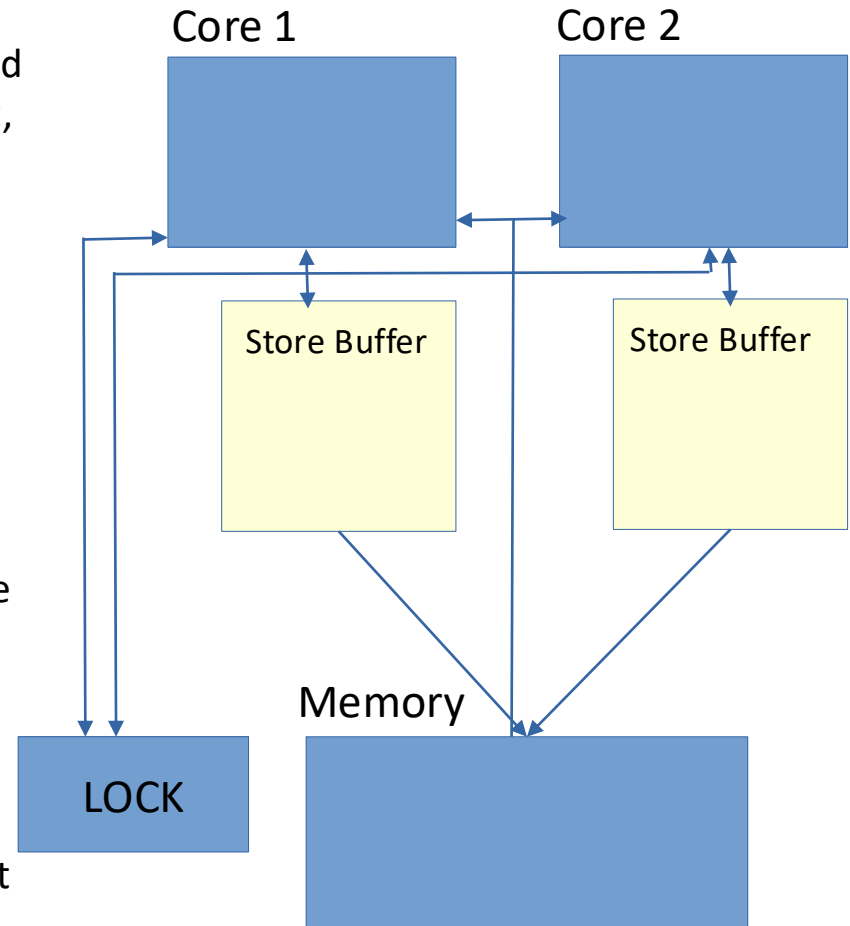
Scope

- Programs using cacheable, write-back memory
 - Q: What does this actually mean?
- Without
 - exceptions
 - misaligned accesses
 - non-temporal operations (which avoid updating L1 cache)
 - Q: And what does this mean?
 - self-modifying code
 - page table changes

x86-TSO

Store buffers are FIFO, a reading thread must read its most recent buffered store, or if none present, a value from memory

- MFENCE flushes a thread's store buffer
- LOCK'd instruction (LOCK is a modifier that applies to other instructions)
 - thread must obtain global lock
 - after instruction, thread flushes its store buffer
 - no other thread can read while global lock is held
- A buffered write can propagate to memory at any time, except when another thread holds the global lock



x86-TSO Behaviours

- $W \rightarrow W$, “Stores are not reordered with other stores”

$A = 1$		$r0 = B$
$B = 1$		$r1 = A$

- Forbidden final state: $r0 = 1, r1 = 0$

- Writes seen in order
- Reads also happen in order
- x86-TSO forbids $r0 = 1$ and $r1 = 0$ at the same time because the write buffer is FIFO, and the threads only communicate through shared memory

x86-TSO Behaviours

- $R \rightarrow W$, “Stores are not reordered with older loads”

$r0 = A$		$r1 = B$
$B = 1$		$A = 1$

- Loads are never delayed

- Forbidden final state: $r0 = 1, r1 = 1$

x86-TSO Behaviours

- $W \not\rightarrow R$, “Loads may be reordered with older stores”

$A = 1$		$B = 1$
$r0 = B$		$r1 = A$

- We saw this in the write buffer example earlier

- Allowed final state: $r0 = 0, r1 = 0$

x86-TSO Behaviours

- “Loads are not reordered with older stores to the same location”

$A = 1$

$r0 = A$

- Required final state: $r0 = 1$

- Note read and write to same location
- Reads must check the local write buffer

x86-TSO Behaviours

- “Stores are transitively visible”

$A = 1$		$r0 = A$		$r1 = B$
		$B = 1$		$r2 = A$

- Forbidden final state: $r0 = 1, r1 = 1, r2 = 0$

x86-TSO Behaviours

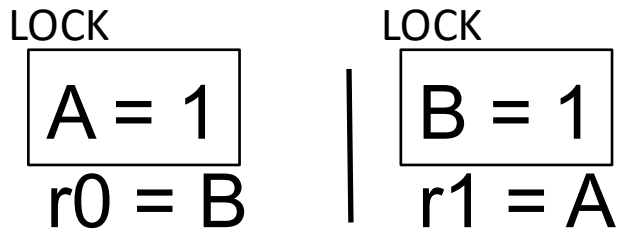
- “Stores are seen in a consistent order by other processors”

A = 1		B = 1		r0 = A		r2 = B
				r1 = B		r3 = A

- Forbidden final state: $r0 = 1, r1 = 0, r2 = 1, r3 = 0$
- $r0, r1$ constraints: A written before B
- $r2, r3$ constraints: B written before A

x86-TSO Behaviours

- “Locks flush the write buffer”



- Forbidden final state: $r0 = 0, r1 = 0$

x86-TSO Behaviours

- “Loads are not reordered with locks”

XCHG(A,r0)		XCHG(B,r2)
r1 = B		r3 = A

- XCHG(X,r0) swaps the contents of X and r0
 - implicitly LOCKs
- Similar to previous example

- Initial state: r0 = 1, r2 = 1, others 0
- Forbidden final state: r1 = 0, r3 = 0

x86-TSO Behaviours

- “Stores are not reordered with locks”

A = 1		B = 1
MFENCE		MFENCE
r0 = B		r1 = A

- Forbidden final state: r0 = 0, r1 = 0
- Similar to write buffer counterexample to SC but using fences to flush the write buffers instead of LOCK instructions
- Adding MFENCE between every instruction recovers SC (but ideally not so many needed)

Spinlock

lock(L) =

```
acquire: LOCK; DEC [L]      ;; decrement L in {0,1}
          JNS enter         ;; if L was >= 1, we have taken the lock
spin:     CMP [L], 0         ;; test L
          JLE spin          ;; if L <= 0, keep spinning (lock busy)
          JMP acquire        ;; try again
enter:    ;; critical section
```

unlock(L) = MOV [L], 1

- Is it OK to have just a normal store as the unlock operation?
 - Allows the thread to continue without flushing the buffer

???

Spinlock

lock(L) =

```
acquire: LOCK; DEC [L]      ;; decrement L in {0,1}
          JNS enter         ;; if L was >= 1, we have taken the lock
spin:     CMP [L], 0         ;; test L
          JLE spin          ;; if L <= 0, keep spinning (lock busy)
          JMP acquire        ;; try again
enter:                                ;; critical section
```

unlock(L) = MOV [L], 1

- Is it OK to have just a normal store as the unlock operation?
 - Allows the thread to continue without flushing the buffer
- Answer: Yes
 - Under TSO, the writes in the critical section cannot be reordered past the unlock write ($W \rightarrow W$)
 - i.e. they will all leave the write buffer before the unlock happens

What do we gain from all this?

- Looser HW memory models improve performance
 - operation latency can be overlapped with other operations
- HW memory models today are loose in many ways
 - operations within a thread may appear out of order
 - operations by different threads may only be partially ordered
- The x86-TSO model provides an understandable model for programming x86 systems
 - better than prior specifications, which were wrong in different ways

What makes a good memory model?

A good memory consistency model should possess Sarita Adve's 3Ps plus a fourth P:

Programmability: A good model should make it (relatively) easy to write multithreaded programs. The model should be intuitive to most users, even those who have not read the details. It should be precise, so that experts can push the envelope of what is allowed.

Performance: A good model should facilitate high-performance implementations at reasonable power, cost, etc. It should give implementors broad latitude in options.

Portability: A good model would be adopted widely or at least provide backward compatibility or the ability to translate among models.

Precision: A good model should be precisely defined, usually with mathematics. Natural languages are too ambiguous to enable experts to push the envelope of what is allowed.

How good are SC and TSO?

Using these 4Ps:

Programmability: SC is the most intuitive. TSO is close because it acts like SC for common programming idioms. Nevertheless, subtle non-SC executions can bite programmers and tool authors.

Performance: For simple cores, TSO can offer better performance than SC

Portability: SC is widely understood, while TSO is widely adopted.

Precision: SC, and TSO are formally defined.

Even More Relaxed Consistency

SPARC

- Partial Store Ordering
 - Write coalescing: wait until several writes to the same cache line accumulate in the write buffer before sending them to memory
 - Breaks $W \rightarrow W$

ARM and POWER

- ARM and POWER: similar to each other
- ARM: widely used inside smartphones and tablets
 - (battery-friendly)
- POWER architecture = Performance Optimization With Enhanced RISC
 - Main driver: IBM
- Memory model is much weaker than TSO

Message-passing example

- P_0 wants to pass a message over “channel” X to P_1 , with shared var Y used as a flag

P_0	P_1
$X = 1$ $Y = 1$	$\text{while } (Y = 0) \{$ $\quad r0 = X$

- Is the result $r0 = 0$ allowed?
- Not in (x86-)TSO!
- Read of Y needs to happen before read of X
- Write of X needs to happen before write of Y
- Would violate $R \rightarrow R$ and/or $W \rightarrow W$

Message-passing example

- P_0 wants to pass a message over “channel” X to P_1 , with shared var Y used as a flag

P_0	P_1
$X = 1$ $Y = 1$	$\text{while } (Y = 0) \{$ $\quad r0 = X$

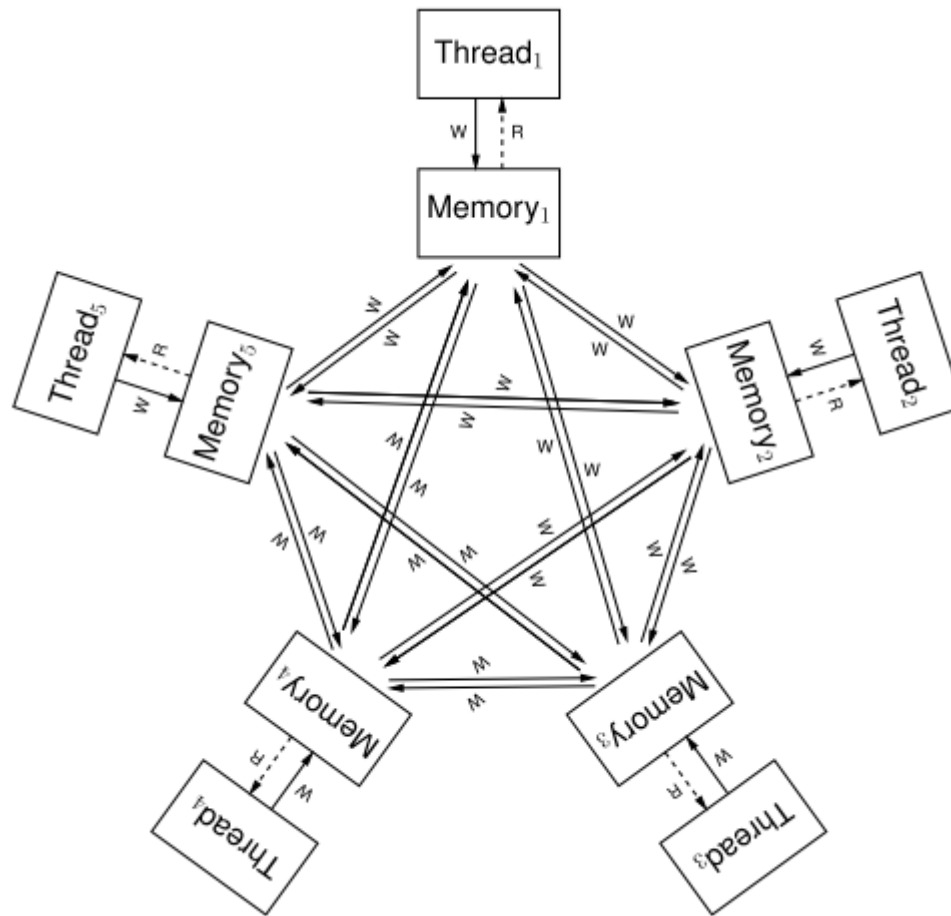
- In Arm and Power neither $W \rightarrow R$, $R \rightarrow R$, $W \rightarrow W$ are guaranteed

- Is the result $r0 = 0$ allowed?
- Not in (x86-)TSO!
- Read of Y needs to happen before read of X
- Write of X needs to happen before write of Y
- Would violate $R \rightarrow R$ and/or $W \rightarrow W$

ARM and POWER

- Program order is not preserved! Unless
 - Writes to the same location
 - Address dependency between two loads
 - Dependency between a load and a store,
 - address dependency
 - data dependency
 - control dependency
 - Use of synchronisation instructions

Conceptual model



PS Current Arm model is in fact “multicopy atomic”: Writes seen by *some* other thread is seen by *all* other threads. Q: How does this affect the picture?

Fixing the example

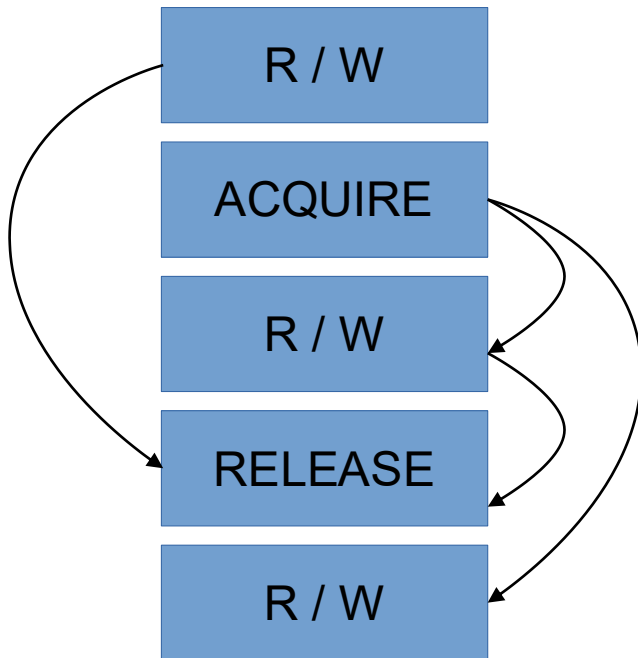
$X = 1$		<code>while (Y = 0) {}</code>
<code>fence</code>		<code>fence</code>
$Y = 1$		<code>r0 = X</code>

- With fences, message passing works as intended and `r0 = 0` never happens
- In ARM: `sync`

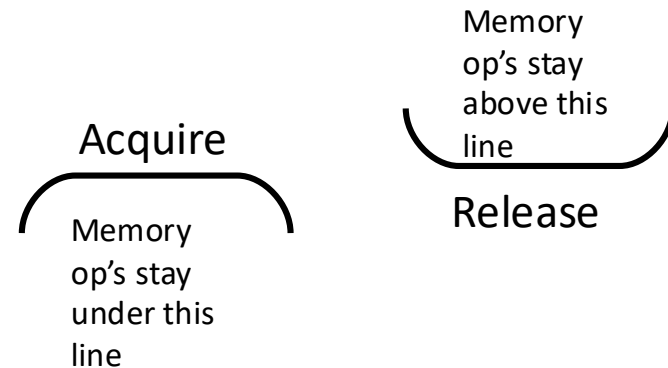
Acquire and Release

- Further relaxation of weak consistency
- Synchronisation accesses divided into acquire and release
- Semantics of acquire:
 - Acquire to complete prior to all following memory accesses
- Semantics of release:
 - All memory operations to complete prior to release
- Thus: A memory operation following acquire is guaranteed to “see” at least all memory operations preceding an earlier release
- However,
 - acquire does not wait for accesses preceding it
 - accesses after release in program order do not have to wait for release
 - operations which follow release and which need to wait must be protected by an acquire

Release Consistency



Which operations can be reordered?



ACQUIRE → Load, Store (but not Load, Store → ACQUIRE)

Load, Store → RELEASE (but not RELEASE → Load, Store)

SC ordering of ACQUIRES and RELEASES:

ACQUIRE → ACQUIRE

ACQUIRE → RELEASE

RELEASE → ACQUIRE, and

RELEASE → RELEASE

Spinlock in ARM

lock()

```
=      MOV   R1, #0x1                ; load the 'lock taken' value
try:LDAXR   R0, [L]                  ; load reserved the lock value
      CMPEQ  R0, #0                   ; is the lock free?
      BNE    try                      ; no - try again
      STREXEQ R1, R0, [L]             ; try and claim the lock
      CMPEQ  R0, #0                   ; did this succeed?
      BNE    try                      ; no - try again. . .
; yes - we have the lock
```

unlock() =

```
      STLR   ZR, [L]
```

- Release consistency prevents reorderings out of the critical section, though it does allow reorderings within it
- LDAXR R0, [L] atomically loads the value in L to register R0 with acquire semantics, and initializing a reservation on L (see also ll/sc)
- STREX R1, R0, [L] performs a Store-Exclusive operation to the address in L, conditionally storing the value from R1 and indicating success or failure in R0.

Spinlock in ARM

Load acquire exclusive reserved
Store exclusive

```
lock()
=    MOV    R1, #0x1                ; load the 'lock taken' value
try: LDAXR   R0, [L]                ; load acquire the lock value
      CMPEQ  R0, #0                 ; is the lock free?
      BNE    try                    ; no - try again
      STREXEQ R1, R0, [L]           ; try and claim the lock
      CMPEQ  R0, #0                 ; did this succeed?
      BNE    try                    ; no - try again. . .
; yes - we have the lock

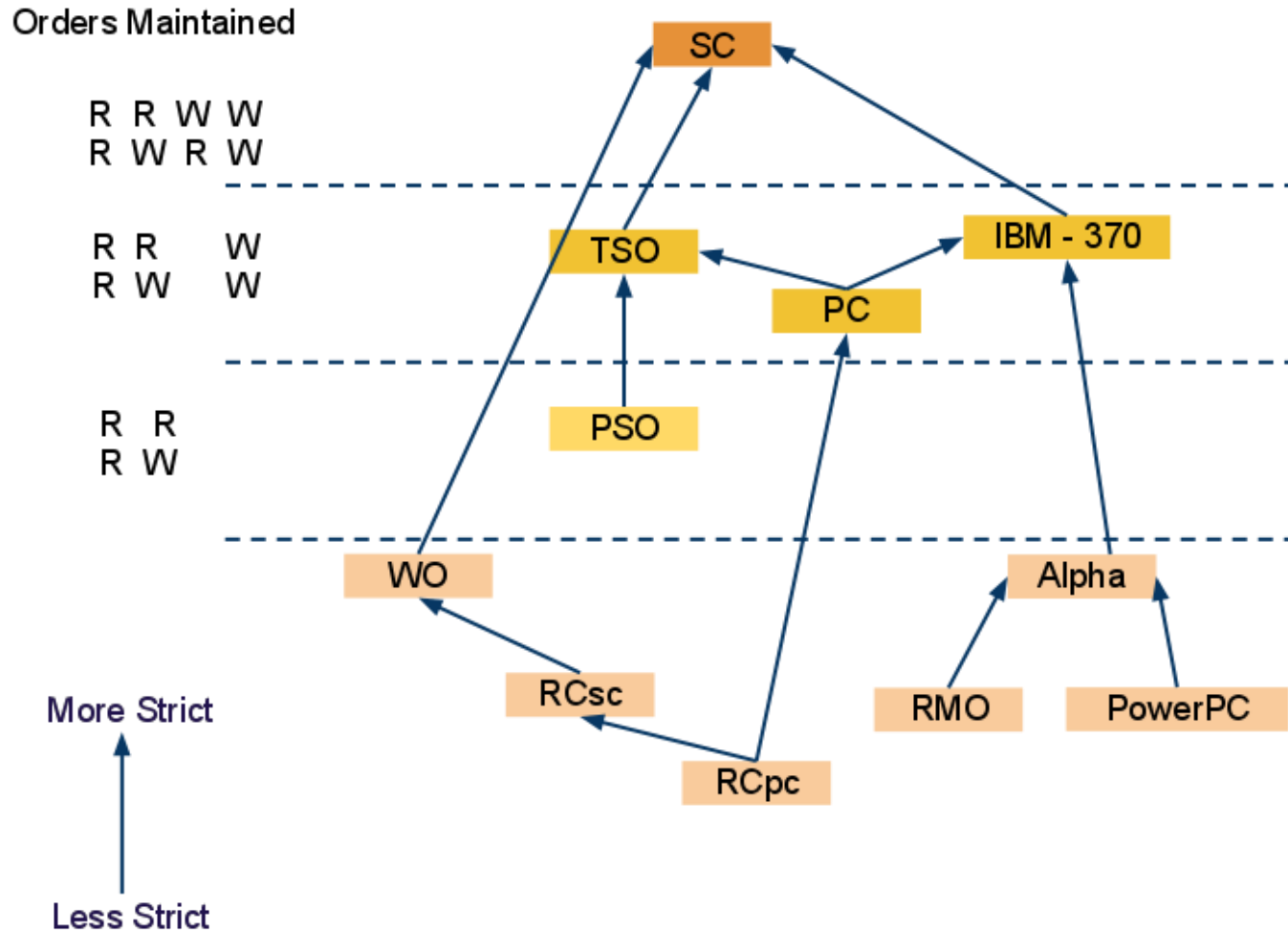
unlock() =
      STLR   ZR, [L]                ; Store release
```

- Release consistency prevents reorderings out of the critical section, though it does allow reorderings within it
- LDAXR R0, [L] atomically loads the value in L to register R0 with acquire semantics, and initializing a reservation on L (see also ll/sc)
- STREX R1, R0, [L] performs a Store-Exclusive operation to the address in L, conditionally storing the value from R1 and indicating success or failure in R0.

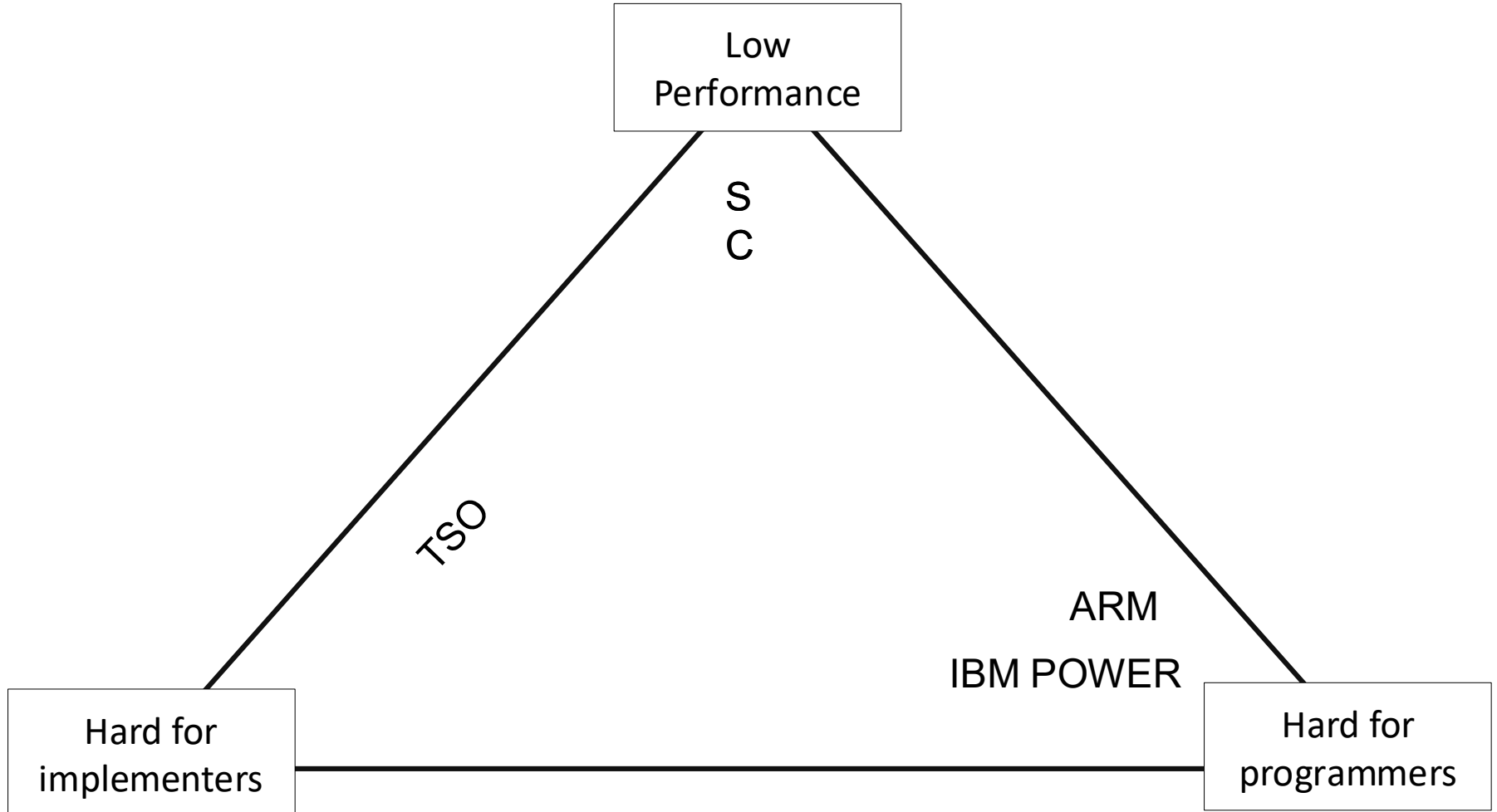
How good are these models?

- **Programmability:** Acceptable for those who use “SC for DRF.” Deeply understanding a relaxed model (e.g., to write compilers and runtime systems) is difficult.
- **Performance:** Can offer better performance than TSO, but the difference is smaller for manycore micro-architectures.
- **Portability:** Porting while staying within “SC for DRF” is manageable. Pushing the limits of these models, especially those that are incomparable, is hard.
- **Precision:** Many models are only informally defined. Moreover, formal definitions tend to be obtuse

The Big Picture



The Big Picture



Summary

- HW memory models are like **contracts** that specify how the processor may reorder memory operations
- SC is the simplest model, but expensive to implement
- Relaxed consistency: TSO (x86), ARM
 - Improved throughput
 - but harder to reason about
 - worth it?
- Programmers can recover SC using fences and other synchronisation primitives (RMW, Load-Acquire, Store-Release)
- What on earth happened to high level programming?
- More on this next week!