

EECS 570

Lecture 15

# Weaker Consistency Models

Winter 2025

Prof. Satish Narayanasamy

<http://www.eecs.umich.edu/courses/eecs570/>

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzky, Reinhardt, Singh, Smith, Torrellas and Wenisch.

# Announcements

Midterm exam – 26<sup>th</sup> Wednesday

**Old exams posted on the website**

**Time: 3-4:20p**

**Location:**

DOW 1017                      (Last name: A – M)

IOE 1680                      (Last name: N- T)

DOW 1206                      (Last name: U-Z)

**Syllabus: all lectures and reading up until this week**

# Readings

For this week:

Sorin, Hill, Wood. A Primer on Memory Consistency and Cache Coherence. Synthesis Lectures, 2011. Chapter 3

A Safety-First Approach to Memory Models. Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, Madanlal Musuvathi. IEEE Micro, Top Picks from the 2012 Computer Architecture Conferences, May/June 2013.

Skim this paper: [The Silently Shifting Semicolon.](#)

Daniel Marino, Todd D. Millstein, Madanlal Musuvathi, Satish Narayanasamy, Abhayendra Singh, [Madan Musuvathi](#)  
***1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*** | May 2015  
[Publication](#)

# What states are infeasible under SC?

Core 0	Core 1
(i1) x = 1; (i2) y = 1;	(i3) r1 = y; (i4) r2 = x;
<hr/>	

A=0    flag=0

Processor 0

A=1;  
flag=1;

Processor 1

while (!flag); // spin  
print A;

---

A=0    B=0

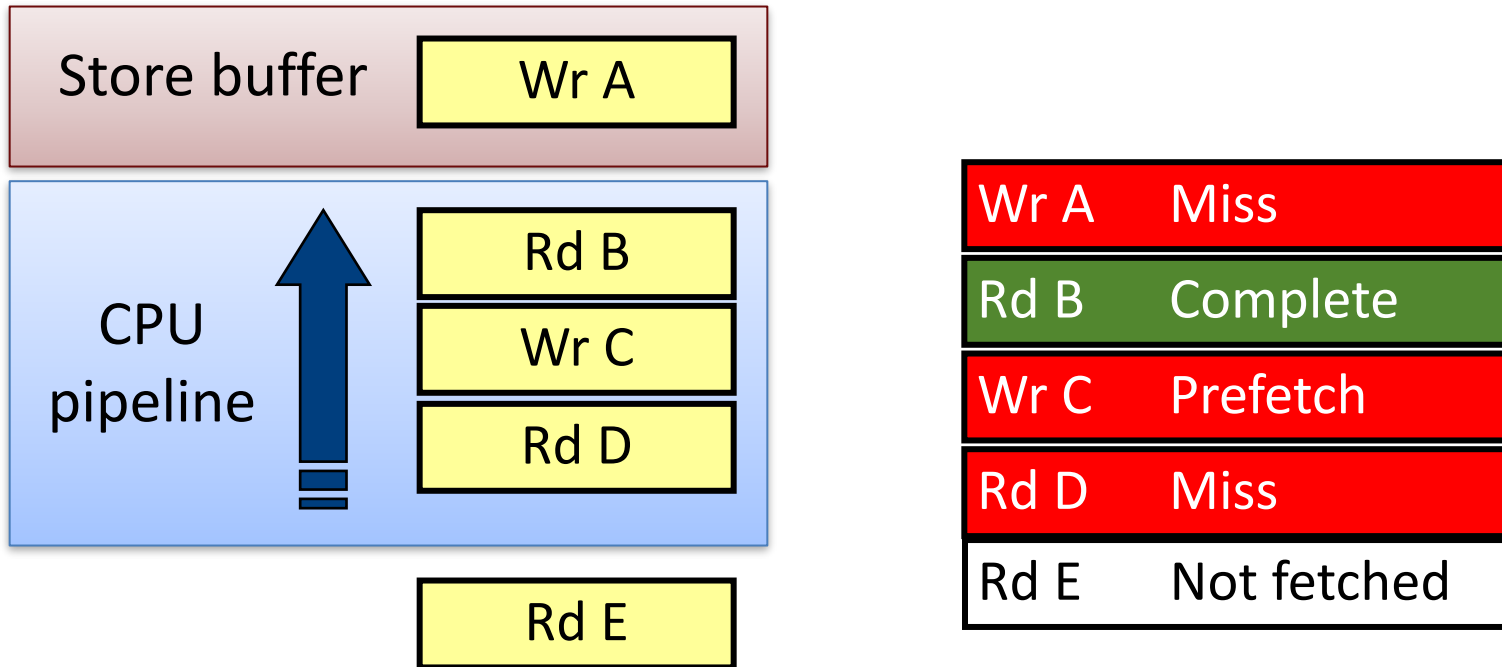
P1

A=1;

P2

while (A==0);  
B = 1;

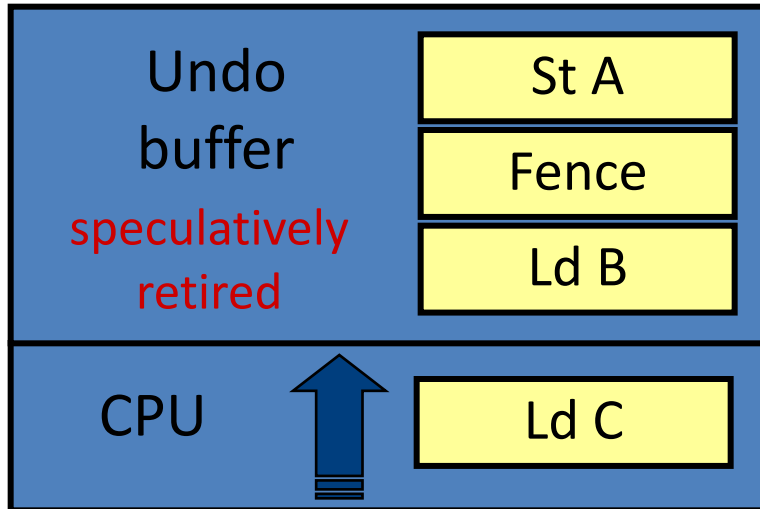
# In-Window Speculation



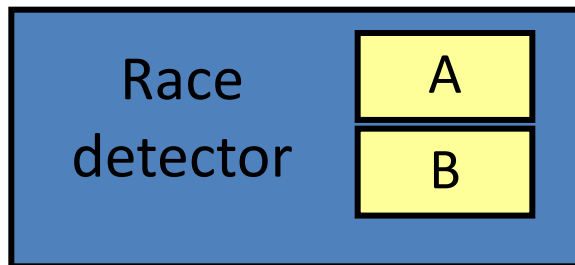
- Key Idea: Perform load speculatively, use branch rewind to roll back if the value of the load changes
  - Invalidation messages “snoop” load-store queue
    - If invalidation “hits” a speculative load, rewind & re-execute
    - Alternative implementation – redo all loads in program order at retirement (“Value”-based ordering) [Cain & Lipasti 04]

# Out-of-Window Speculation

Out-of-Window Speculation  
Early HW solutions



- Log all instructions  
**Large storage requirement**
- Read old value before store  
**Extra L1 traffic**



- Assoc. search on external req.  
**Limited capacity**

**Early solutions require impractical mechanisms**



- Key departure: apply to weakly-ordered system
  - ▣ Straightforward hardware; fewest stalls to address
- Augment with familiar deep speculation mechanisms
  - ▣ Violation detection: read/write bits in cache
  - ▣ Version management: clean to L2 before 1<sup>st</sup> write
- Result: eliminate fence stalls (up to 13% speedup)
  - ▣ No fine-grained (per-store) tracking
  - ▣ Fast & simple commit and rollback
  - ▣ Conventional memory system
- For strong ordering: speculate more (“implicit fences”)
  - ▣ Bonus: can even eliminate LSQ snooping! (a la [Ceze’07])

Language-Level  
DRF-0 Vs SC  
Memory Model

# Program Order

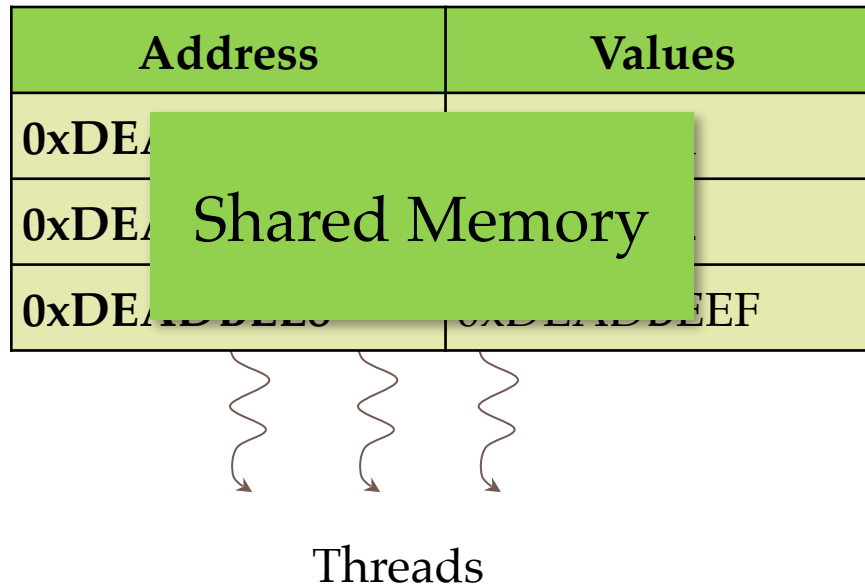


a thread

A ; B

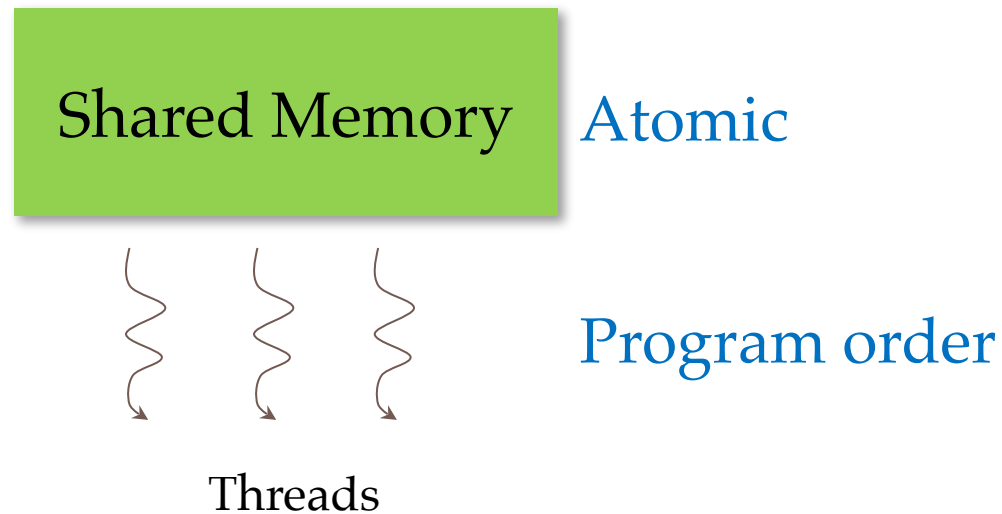
Execute A and then B

# Atomic Shared Memory



Memory is a map from address to values  
with reads / writes taking effect immediately

# Intuitive Concurrency Semantics



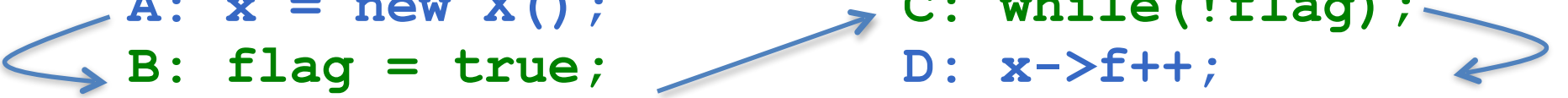
Memory model that guarantees this  
is called **sequential consistency**

# Sequential Consistency

```
X* x = null;  
bool flag = false;
```

// Producer Thread

A: x = new X();  
B: flag = true;



// Consumer Thread

C: while(!flag);  
D: x->f++;

sequential consistency  
(SC)

[Lamport 1979]

memory operations **appear** to occur  
in some global order consistent with  
the program order

# Intuitive reasoning fails in C++ / Java

```
X* x = null;  
bool flag = false;
```

// Producer Thread

A: `x = new X();`  
B: `flag = true;`

// Consumer Thread

C: `while(!flag);`  
D: `x->f++;`

**In C++ model this can crash!**



# Intuitive reasoning fails in C++/Java

```
X* x = null;  
bool flag = false;
```

```
// Producer  
A: x = new X();  
B: flag = true;
```

```
// Consumer  
C: while(!flag);  
D: x->f++;
```

Optimizing Compiler and Hardware



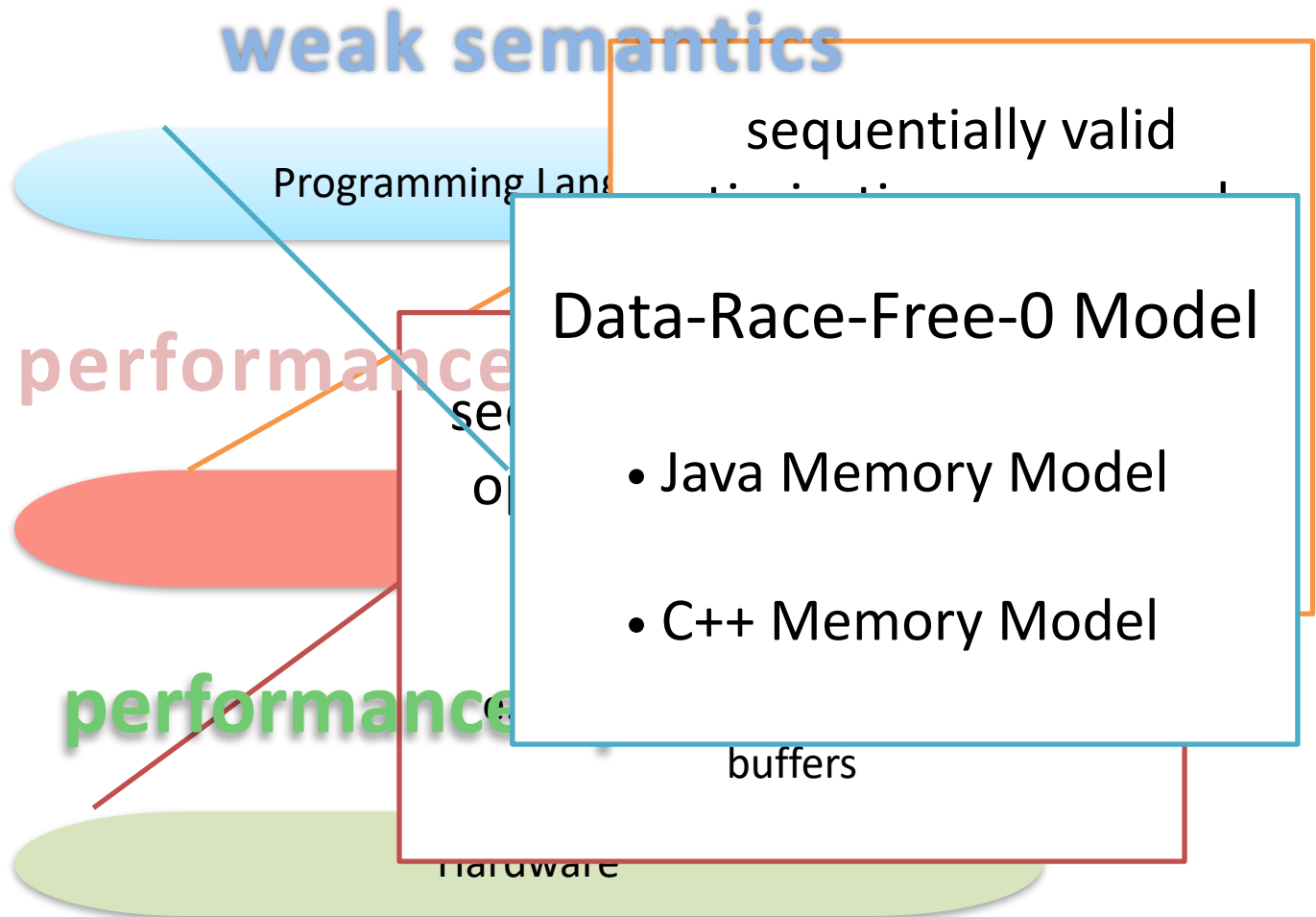
Null  
Dereference!

B doesn't depend on A.  
It might be faster to reorder them!





# Why are accesses reordered?



# A Short Detour: Data Races

A program has a **data race** if it has an execution in which two **conflicting accesses** to memory are simultaneously ready to execute.

// Thread 1: access the same memory location  
// Thread 2: at least one is a write  
A: x = 0; // Thread 1: read u  
B: flag = true; // Thread 2: write (!flag);  
D: x->f++;

Data Race

# Useful Data Races

- Data races are essential for implementing shared-memory synchronization

```
AcquireLock(){  
    while (lock == 1) {}  
    t = CAS (lock, 0, 1);  
    if (!t) retry;  
}
```

```
ReleaseLock() {  
    lock = 0;  
}
```

# Data Race Free Memory Model

A program is **data-race-free** if all data races are appropriately annotated (`volatile/atomic`)

## DRFO

[Adve & Hill 1990]

SC behavior for data-race-free programs,  
**weak** or **no** semantics otherwise

Java Memory Model  
(JMM)


[Manson et al. 2005]

C++0x Memory Model


[Boehm & Adve 2008]

# DRF0-compliant Program

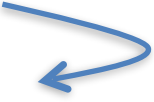
```
    X* x = null;  
    atomic bool flag = false;
```



```
A: x = new X();  
B: flag = true;
```



```
C: while(!flag);  
D: x->f++;
```



- DRF0 guarantees SC  
.... only if data-race-free (all *unsafe* accesses are annotated)
- What if there is one data-race?  
.... all bets are off (e.g., compiler can output an empty binary!)

# Data-Races are Common

- Unintentional data-races
  - Easy to accidentally introduce a data race
    - forget to grab a lock
    - grab the wrong lock
    - forget a `volatile` annotation
    - ...
- Intentional data-races
  - 100s of “benign” data-races in legacy code

[Narayanasamy et al. PLDI 2007]

# Data Races with no Race Condition (assuming SC)

- Single writer multiple readers

```
// Thread t  
A:  time++;
```

```
// Thread u  
B:  l = time;
```

# Data Races with no Race Condition (assuming SC)

- Lazy initialization

```
// Thread t
    if( p == 0 )
        p = init();
```

```
// Thread u
    if( p == 0 )
        p = init();
```



# Intentional Data Races

- ~97% of data races are not errors under SC
  - Experience from one Microsoft internal data-race detection study [Narayanasamy et al. PLDI'07]
- The main reason to annotate data races is to protect against compiler/hardware optimizations

# Data Race Detection is Not a Solution

- Current static data-race detectors are not sound *and* precise
  - typically only handle locks, conservative due to aliasing, ...
- Dynamic analysis is costly
  - DRFx: throw exception on a data-race [Marino'10]
  - Either slow (8x) or requires complex hardware
- Legacy issues

# Deficiencies of DRF0

weak or **no**  
semantics for data-  
racy programs



no easy way to  
identify & reject  
racy programs

---

problematic for

DEBUGGABILITY

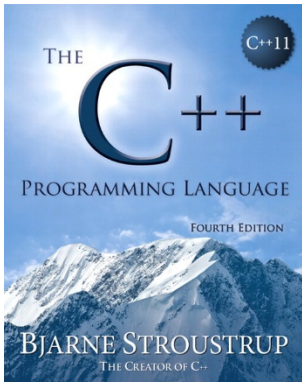
Analogous to unsafe languages:  
relying on programmer  
infallibility

optimi  
jump to arbitrary code!  
[Boehm et al., PLDI 2008]

RECTNESS  
n safety at the  
cost of complexity  
[Ševčík&Aspinall, ECOOP 2008]

# Languages, compilers, processors are adopting DRF0

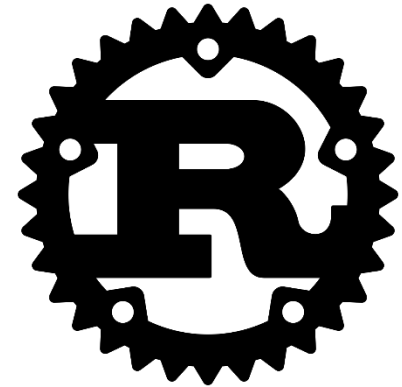
Not a strong foundation



[Foundations of the  
C++ concurrency  
memory model](#)



[The Java memory model](#)



[Rust:  
Atomics - The Rustonomicon](#)



[The Go Memory Model](#)

# Language-level SC: A Safety-First Approach

Program order and shared memory  
are important abstractions

Modern languages should **protect** them

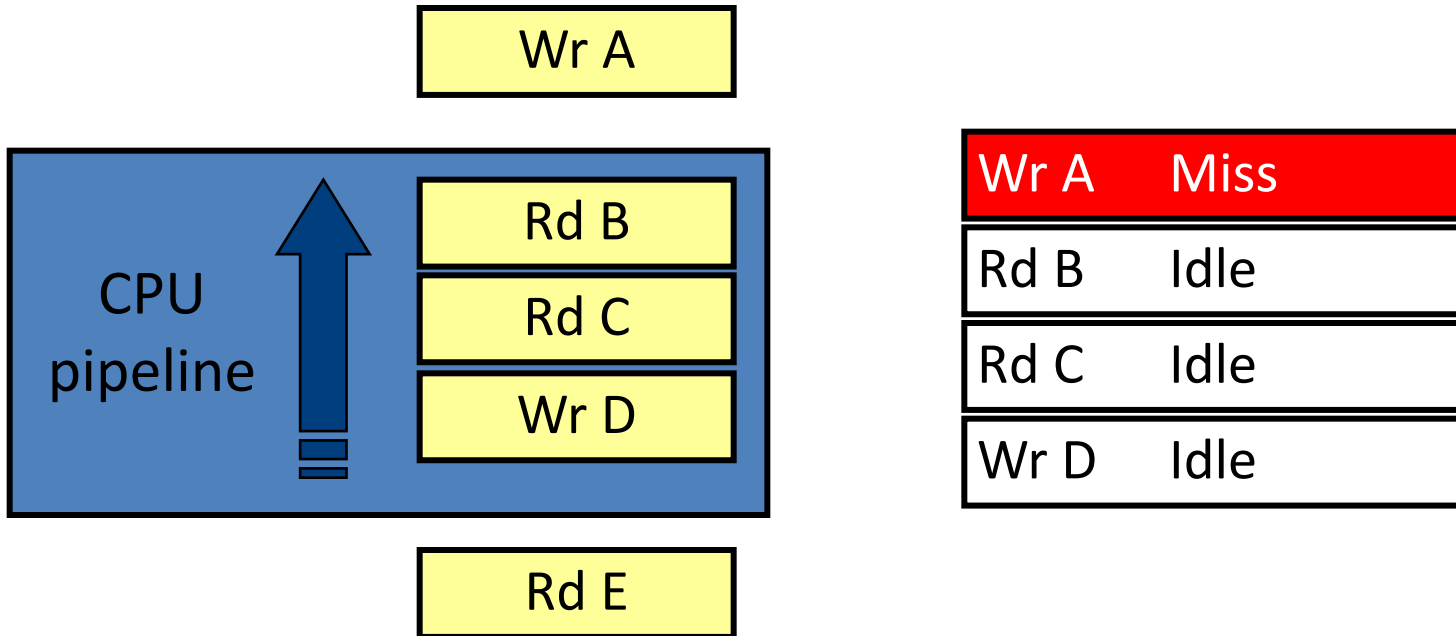
All programs, buggy or otherwise,  
should have SC semantics

# Relaxed Consistency

# Review: Problems with SC

- Difficult to implement efficiently in hardware
  - ▣ Straight-forward implementations:
    - No concurrency among memory access
    - Strict ordering of memory accesses at each node
    - Essentially precludes out-of-order CPUs
- Unnecessarily restrictive
  - ▣ Most parallel programs won't notice out-of-order accesses
- Conflicts with latency hiding techniques

# Execution in strict SC



- Miss on Wr A stalls all unrelated accesses

Memory accesses issue one-at-a-time



# Sun's "Total Store Order" (TSO)

- Formal requirements [v8 architecture manual]:
  - ❑ **Order** - There exists a partial memory order ( $<M$ ) and it is *total* for all operations with store semantics
  - ❑ **Atomicity** - Atomic read-modify-writes do not allow a store between the read and write parts
  - ❑ **Termination** - All stores are performed in finite time
  - ❑ **Value** – Loads return the most recent value w.r.t. memory order ( $<M$ ) and w.r.t. local program order ( $<p$ )
  - ❑ **LoadOP** – Loads are blocking
  - ❑ **StoreStore** – Stores are ordered

# Dekker's Algorithm

- Mutually exclusive access to a critical region
  - Works as advertised under sequential consistency

*/\* initial A = B = 0 \*/*

P1

A = 1;

if (B != 0) goto retry;

*/\* enter critical section\*/*

P2

B=1;

if (A != 0) goto retry;

*/\* enter critical section\*/*

# TSO: Programmer's Perspective

- Can occasionally lead to astonishing behavior changes
  - ❑ E.g., Dekker's algorithm doesn't work
  - ❑ ISAs provide an STBAR (store barrier) to manually force order
    - Semantics – store buffer must be empty before memory operations after STBAR may be executed
  - ❑ Can also enforce order by replacing stores with RMWs
- But, the key case, where sync is done with locks, simply works
  - ❑ Lock acquires are RMW operations  $\Rightarrow$   
they force order for preceding/succeeding loads/stores
    - Load semantics of RMW imply load-load orderings
    - Ditto for store semantics
  - ❑ Lock release is a store operation  $\Rightarrow$   
it must be performed after critical section is done

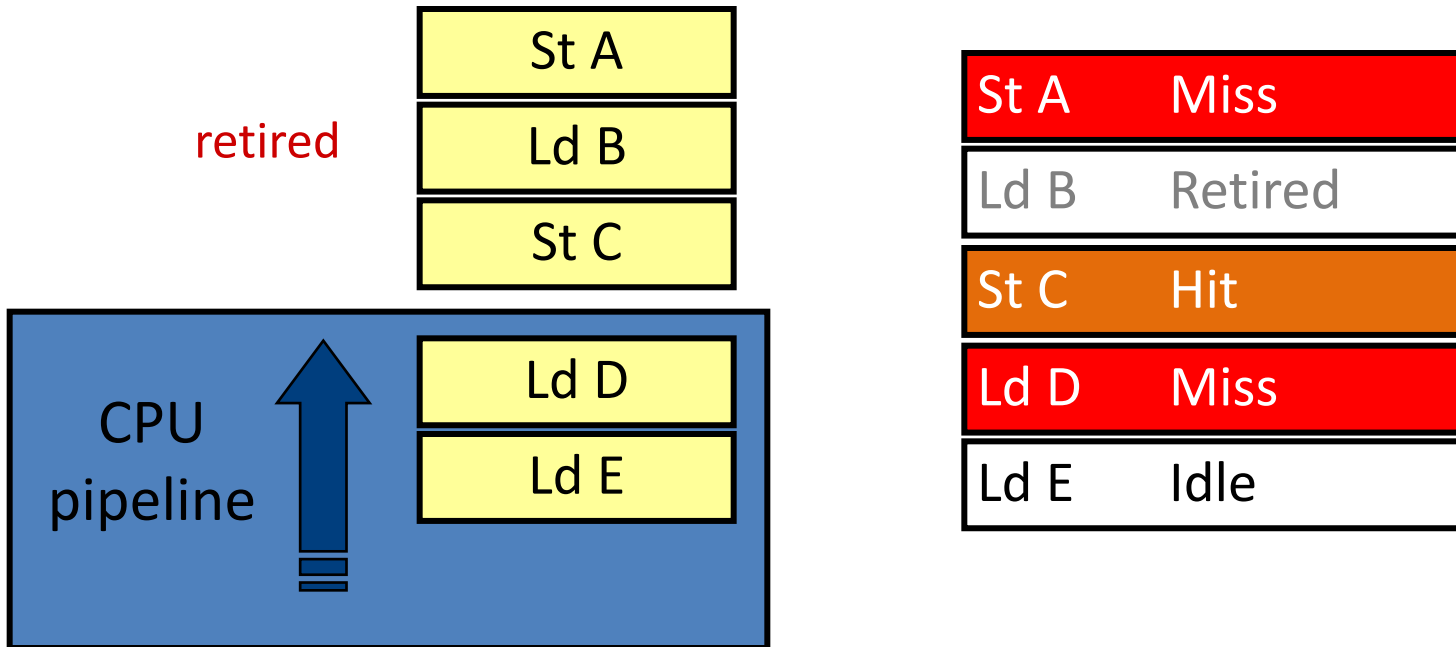
# TSO: Compiler's Perspective

- Compiler may now hoist loads across stores
  - ❑ Still can't reorder stores or move loads across loads
  - ❑ Not clear how helpful this is in practice...
  - ❑ ...Recent results from Prof. Satish's group:
    - 5-30% perf. gap vs. compiler that preserves SC [PLDI'11]
- No new crazy memory barriers to emit
  - ❑ Library/OS writers need to use a little bit of caution; use RMWs instead of loads/stores for synchronization
  - ❑ TSO-unsafe code is rare enough that it can be the programmer's problem
- No need to invoke “undefined behavior” to avoid onerous implementation requirements

# TSO: HW Perspective

- Allows a **FIFO-ordered, non-coalescing store buffer**
  - ▣ Typically maintains stores at word-granularity
  - ▣ Loads search buffer for matching store(s)
    - Some ISAs must deal with merging partial load matches
  - ▣ Coalescing only allowed among adjacent stores to same block
  - ▣ Must force buffer to drain on RMW and STBAR
  - ▣ Often, this is implemented in same HW structure as (speculative) store queue
- Can hide store latency!
  - ▣ But, store buffer may need to be quite big
    - Stores that will be cache hits remain buffered behind misses
  - ▣ Associative search limits scalability
    - E.g., certainly no more than 64 entries

# Execution in TSO



- Stores misses do not stall retirement
  - ❑ St A, St C, Ld D misses overlapped
  - ❑ Ld B retired without waiting for St A to fill
  - ❑ St C consumes space in store buffer even though it will hit

**TSO hides store miss latency**

# TSO Variants

- Differ in their notions of write atomicity
- IBM 370 was the same as TSO except that loads could not read from the store buffer early
- Consider:

Core 0	Core 1
(i1) x = 1; (i2) r1 = x; (i3) r2 = y;	(i4) y = 1; (i5) r3 = y; (i6) r4 = x;
IBM 370 Forbids: r1 = 1, r2 = 0, r3 = 1, r4 = 0	

# Processor Consistency (PC)

- [Goodman 1989]
- Basically TSO with the relaxation of store atomicity
- Consider the IRIW litmus test:

Core 0	Core 1	Core 2	Core 3
<code>x = 1;</code>	<code>y = 1;</code>	<code>r1 = x;</code> <code>r2 = y;</code>	<code>r3 = y;</code> <code>r4 = x;</code>
<b>Allowed under PC: <math>r1 = 1, r2 = 0, r3 = 1, r4 = 0</math></b>			

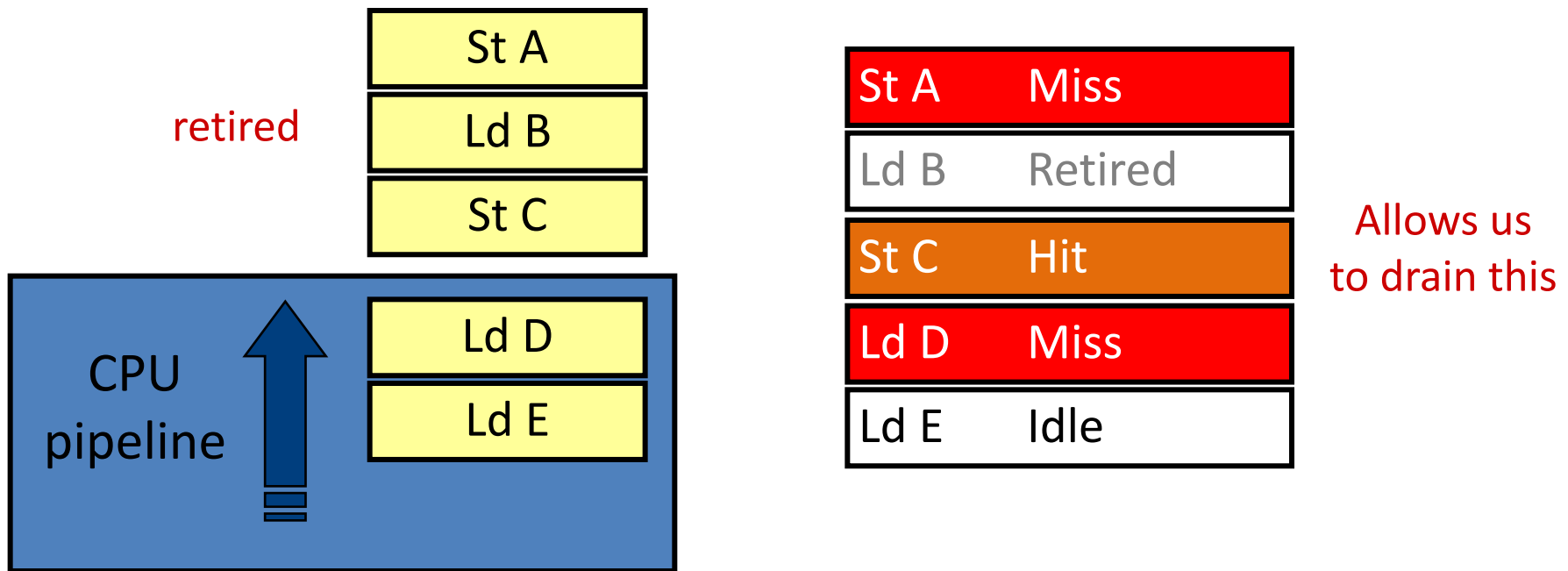


# Variants of Store Atomicity

- Notation from [Trippel et al. ASPLOS 2017]
- MCA: a store becomes visible to all cores (including the performing core) at the same time
  - ❑ IBM 370 style
- rMCA: a store may become visible to the performing core before other cores, but once it becomes visible to one other core, it must become visible to all other cores at the same time
  - ❑ TSO style
  - ❑ commonly referred to as “multicopy atomicity” in the literature today, but terminology can vary
- nMCA: a store may become visible to different cores at different times
  - ❑ PC style
  - ❑ Can **significantly** complicate reasoning

# Relaxing Write-to-Write Order

- Motivation: Coalescing store buffers & early drain



- Allows writes to coalesce in SB & drain early

# Sun's "Partial Store Order" (PSO)

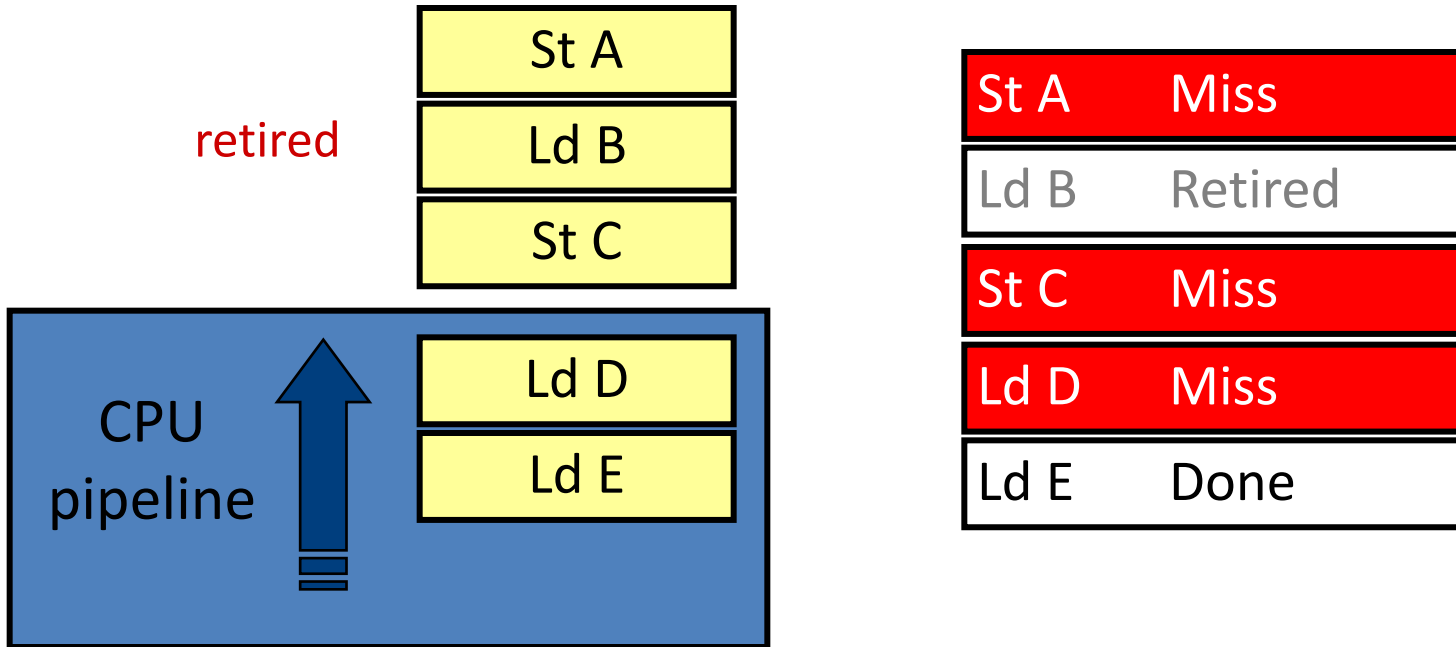
- Formal requirements [v8 architecture manual]:
  - ❑ **Order** - There exists a partial memory order ( $<M$ ) and it is total for all operations with store semantics
  - ❑ **Atomicity** - Atomic read-modify-writes do not allow a store between the read and write parts
  - ❑ **Termination** - All stores are performed in finite time
  - ❑ **Value** – Loads return the most recent value w.r.t. memory order ( $<M$ ) and w.r.t. local program order ( $<p$ )
  - ❑ **LoadOP** – Loads are blocking
  - ❑ **StoreStore** – Stores are ordered only if they are separated by a *membar* (memory barrier) instruction
  - ❑ **StoreStoreEq** – Stores to the same address are ordered

# PSO: Compiler/HW Perspective

- Allows an **unordered, coalescing post-retirement store buffer**
  - ❑ Can now use a cache-block-grain set-associative structure
  - ❑ Store misses leave store buffer upon cache fill
  - ❑ Loads search buffer for matching store(s)
  - ❑ Must force buffer to drain on RMW and STBAR
- Much more efficient store buffer
- But, still doesn't allow out-of-order loads
  - ❑ No OoO execution (without speculation)
  - ❑ Compiler's hands are still tied

# Relaxing all Order

- Motivation: Out-of-order execution & multiple load misses



- Now Ld E can complete even though earlier insn aren't done

# Two ISA Approaches to enforce order

- Approach 1: Using explicit “fence” (aka memory barrier)
  - ▣ Sun’s Relaxed Memory Order (RMO), Alpha, PowerPC, ARM

Ld, St, ...

		L L S S	
	Fence	↓ ↓ ↓ ↓	Enforces order if bit is set
		L S L S	

Ld, St, ...

- Approach 2: Annotate loads/stores that do synchronization
  - ▣ Weak Ordering, Release Consistency (RC)
  - ▣ Data-Race-Free-0 (DRF0) – prog. language-level model

Load.acquire	Lock1
...	
Store.release	Lock1

# More definitions... Dependence Order

- A refinement of program order ( $<p$ )
- Dependence order ( $<d$ ) captures the minimal subset of ( $<p$ ) that guarantees self-consistent execution traces.  $X <p Y$  implies  $X <d Y$  if at least one of the following is true:
  - ❑ The execution of  $Y$  is conditional on  $X$  and  $S(Y)$  ( $Y$  is a store)
  - ❑  $Y$  reads a register that is written by  $X$
  - ❑  $X$  and  $Y$  access the same memory location and  $S(X)$  and  $L(Y)$
- Dependence order captures what an out-of-order core is allowed to do (ignoring exceptions)

# Sun's "Relaxed Memory Order" (RMO)

- Formal requirements [v9 architecture manual]:

- $X <_d Y \wedge L(X) \Rightarrow X <_M Y$

- RMO will maintain dependence order if preceding insn. is a load

- $M(X, Y) \Rightarrow X <_M Y$

- MEMBAR instructions order memory operations

- $Xa <_p Ya \wedge S(Y) \Rightarrow X <_M Y$

- Stores to the same address are performed in program order

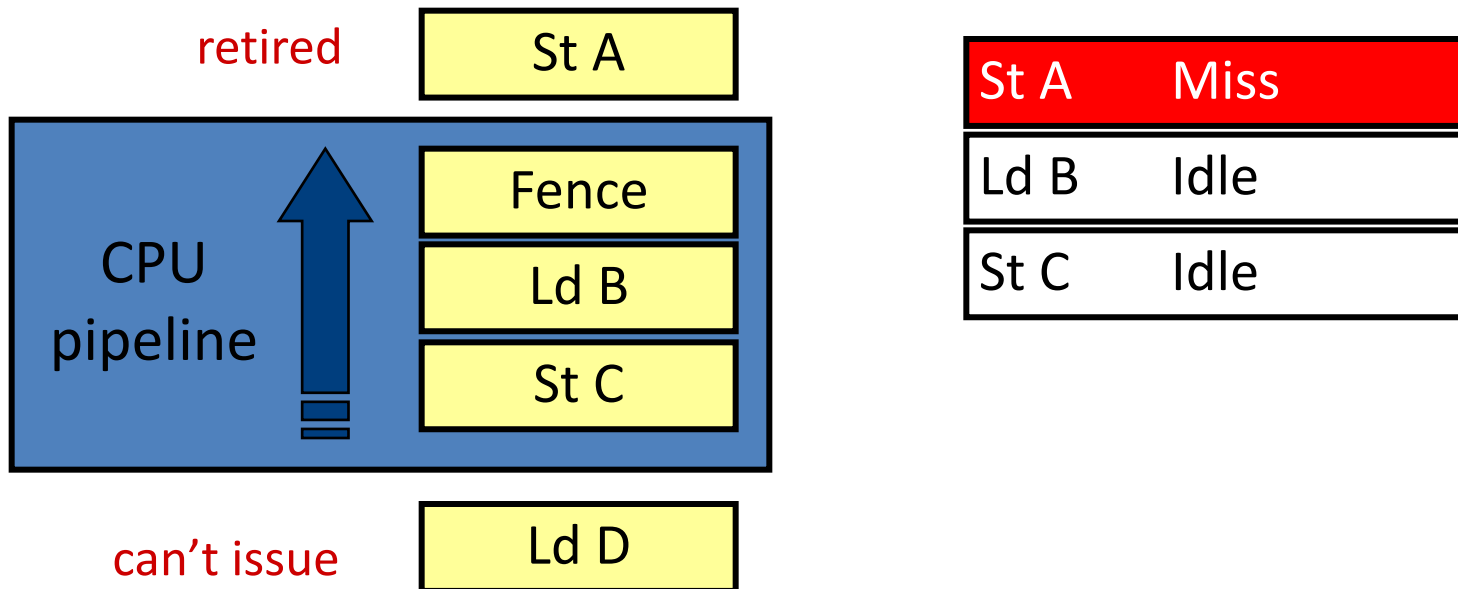
- Assuming Y is a load to memory location a,

- $$\text{Value}(La) = \text{Value}(\text{Max}_{<_m} \{ S \mid Sa <_m La \text{ or } Sa <_p La \} )$$

- where  $\text{Max}_{<_m}\{..\}$  selects the most recent element with respect to the memory order and where  $\text{Value}()$  yields the value of a particular memory transaction.



# Execution in RMO w/ fence



- “Fence” indicates ordering affects correctness
  - ❑ Retirement stalls at Fence
  - ❑ Typically, accesses after fence don't issue

# RMO: Programmer's Perspective

- Programmer must specify MEMBARS wherever they are needed
  - ❑ This is hard; Specifying *minimum* barriers is harder
    - See Vsync [Oberhauser et al. ASPLOS 2021]
  - ❑ Below: lock and unlock (from v9 ref. manual; w/o branch delays)
  - ❑ RMO also does not preserve same address ld-ld ordering!

LockWithLDSTUB(lock)

```
retry:      ldstub [lock],%10
            tst %10
            be out
loop:       ldub [lock],%10
            tst %10
            bne loop
            ba,a retry
out: membar #LoadLoad | #LoadStore
```

UnLockWithLDSTUB(lock)

```
membar #StoreStore      !RMO and PSO only
membar #LoadStore      !RMO only
stub %g0,[lock]
```

# RMO: Compiler's Perspective

- **Sweet, sweet freedom!**

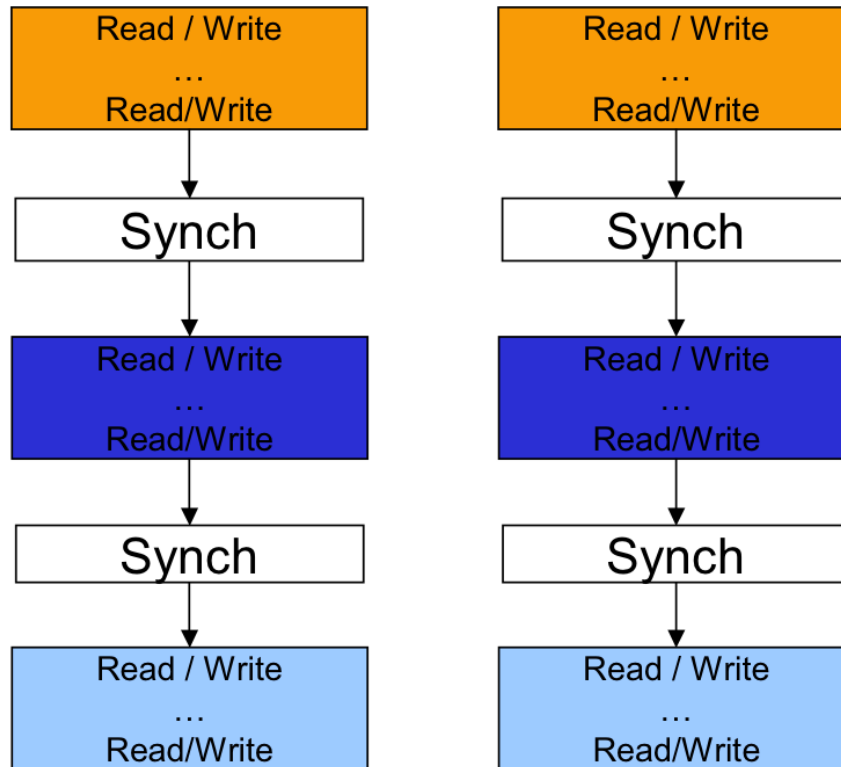
- ❑ Compiler may freely re-order between fences
- ❑ Also across fences if the fence allows it (partial fences)
- ❑ Note: still can't reorder stores to same address
- ❑ Programmer's problem if the fences are wrong...

# RMO: HW Perspective

- Unordered, coalescing post-retirement store buffer
  - ▣ Just like PSO
- Out-of-order execution!
  - ▣ Need a standard uniprocessor store queue
  - ▣ Fence instruction implementation
    - Easy – stall at issue
    - Hard – retire to store buffer and track precise constraints

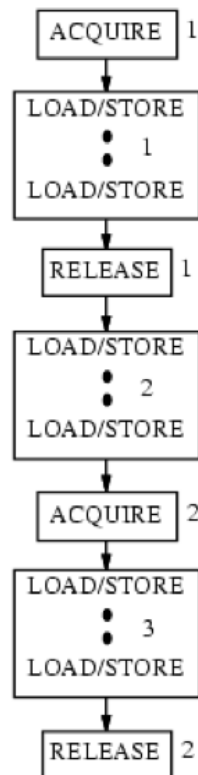
# Weak Ordering (WO)

- [Dubois et al. ISCA 1986]
- Loads and stores can be labeled as “sync”
  - ❑ No reordering allowed across sync instructions

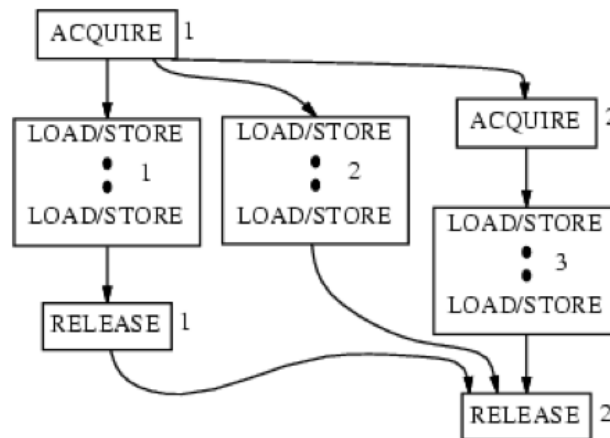


# Release Consistency (RC)

- Specialize loads and stores as “acquires” and “releases”
  - ❑ Load at top of critical section is an acquire
  - ❑ Store at bottom of critical section is a release
  - ❑ Allows reordering **into** critical sections, but not out



Weak Consistency (WCsc)



Release Consistency (RCpc)

u  
↓  
v  
v cannot perform with respect to  
any other processor until u is  
performed

# Release Consistency (RC)

- Two flavours: RCsc and RCpc
  - ❑ In RCsc, special accesses are sequentially consistent with respect to each other
  - ❑ In RCpc, special accesses are processor consistent with respect to each other
- The difference has significant ramifications!
  - ❑ In RCpc, even adding enough synchronization to eliminate data races does not make the program behave like SC!
  - ❑ In RCpc, ordering is not maintained between releases and subsequent acquires
    - Why do this?
  - ❑ In RCpc, releases do not become visible to all cores at the same time
    - Why do this?

# Data Races

A program has a **data race** if it has an execution in which two accesses to the **same address** on **different threads** where **at least one is a write** and **at least one is not a synchronization access** are not ordered by synchronization accesses

Note: The definition of races can vary depending on the context.

Core 0

A: x = 1;

B: y = 1;

Core 1

C: r1 = y;

D: z = 1;

Core 0

A: x = 1;

B: y = 1;

Core 1

C: r1 = y;

D: bne r1, 1, C

E: r2 = x;



# WO/RC: Programmer Perspective

- A new way of thinking: **programmer-centric memory models**
  - ❑ If you annotate syncs correctly, your program will behave like SC
    - WHY?
  - ❑ E.g., Data-Race-Free-0 (DRF0) [Adve and Hill ISCA 1990]
    - Accesses are either normal or sync
    - Sync accesses are sequentially consistent, and order normal accesses
    - Data races among normal accesses are prohibited
    - **DRF0 programs appear as if they ran under SC**
    - Similar idea in [Gharachorloo et al. ISCA 1990]: “Properly-[Labelled]” Programs
  - ❑ SC-for-DRF forms the basis for programming language memory models

# The Ramifications of SC-for-DRF

- Hardware can freely reorder instructions in between acquire and release operations
  - ❑ No one else can tell!
- Compilers can freely reorder code in between acquire and release operations
  - ❑ Again, no one else can tell!
- So does this solve all memory consistency problems?
  - ❑ Not quite!
  - ❑ The hardware and compiler still need to maintain ISA-level MCM and PL MCM guarantees
  - ❑ What about cases where we want some atomics that are not sequentially consistent? (for performance reasons)
  - ❑ Next time: programming language MCMs