

# CS 636: Memory Consistency Models

Swarnendu Biswas

Semester 2020-2021-II  
CSE, IIT Kanpur

---

Content influenced by many excellent references, see References slide for acknowledgements.

# Correctness of Shared-memory Programs

“To write correct and efficient shared memory programs, programmers need a precise notion of how memory behaves with respect to read and write operations from multiple processors”

---

S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. WRL Research Report, 1995.

# Busy-Wait Paradigm

```
Object X = null;  
boolean done= false;
```

## Thread T1

```
X = new Object();  
done = true;
```

## Thread T2

```
while (!done) {}  
if (X != null)  
    X.compute();
```

# What Value Can a Read Return?

```
X = 0  
done = 0
```

## Core C1

```
S1: store X, 10  
S2: store done, 1
```

## Core C2

```
L1: load r1, done  
B1: if (r1 != 1) goto L1  
L2: load r2, X
```

# Reordering of Accesses by Hardware

Different addresses!

Store-store

Load-load

Load-store

Store-load

How?

# Reordering of Accesses by Hardware

Different addresses!

## Store-store

- Non-FIFO write buffer (first store misses in the cache while the second hits or the second store can coalesce with an earlier store)

## Load-load

- Cache hits, dynamic scheduling, execute out of order

## Load-store

- Cache hits, out-of-order core

## Store-load

- FIFO write buffer with bypassing, out-of-order core

# Reordering of Accesses by Hardware

Different addresses!

## Store-store

- Non-FIFO write buffer (first store misses in the cache while the second hits or the

Correct in a single-threaded context

Non-trivial in a multithreaded context

## Store-load

- FIFO write buffer with bypassing, out-of-order core

# What values can a load return?

## Return the “last” write

- Uniprocessor: program order defines the “last” write
- Multiprocessor: ?



# Memory Consistency Model

Set of rules that govern how systems process memory operation requests from multiple processors

- Determines the order in which memory operations appear to execute

Specifies the allowed behaviors of multithreaded programs executing with shared memory

- Both at the hardware-level and at the programming-language-level
- There can be multiple correct behaviors

# Importance of Memory Consistency Models

Determines what optimizations are correct

Contract between the programmer and the hardware

Influences ease of programming and program performance

Impacts program portability

# Dekker's Algorithm

```
flag1 = 0  
flag2 = 0
```

## Core C1

```
S1: store flag1, 1  
L1: load r1, flag2
```

## Core C2

```
S2: store flag2, 1  
L2: load r2, flag1
```

Can both r1 and r2 be set to zero?

# Issues with Memory Consistency

## Visibility

- When does a value update become visible to others?

## Ordering

- When can operations of any given thread appear out of order to another thread?

# Sequential Consistency

A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in **some sequential order**, and the operations of each individual processor appear in **the order specified by the program**.

# Sequential Consistency (SC)

## Uniprocessor

- Operations execute in the order specified by the program

## Multiprocessor

- All operations execute in order, and the operations of each individual core appear in program order

# Uniprocessor Memory Model

- Memory operations occur in program order
  - Only maintain data and control dependences
- Read from memory returns the value from the last write in program order
- Compiler optimizations preserve these semantics

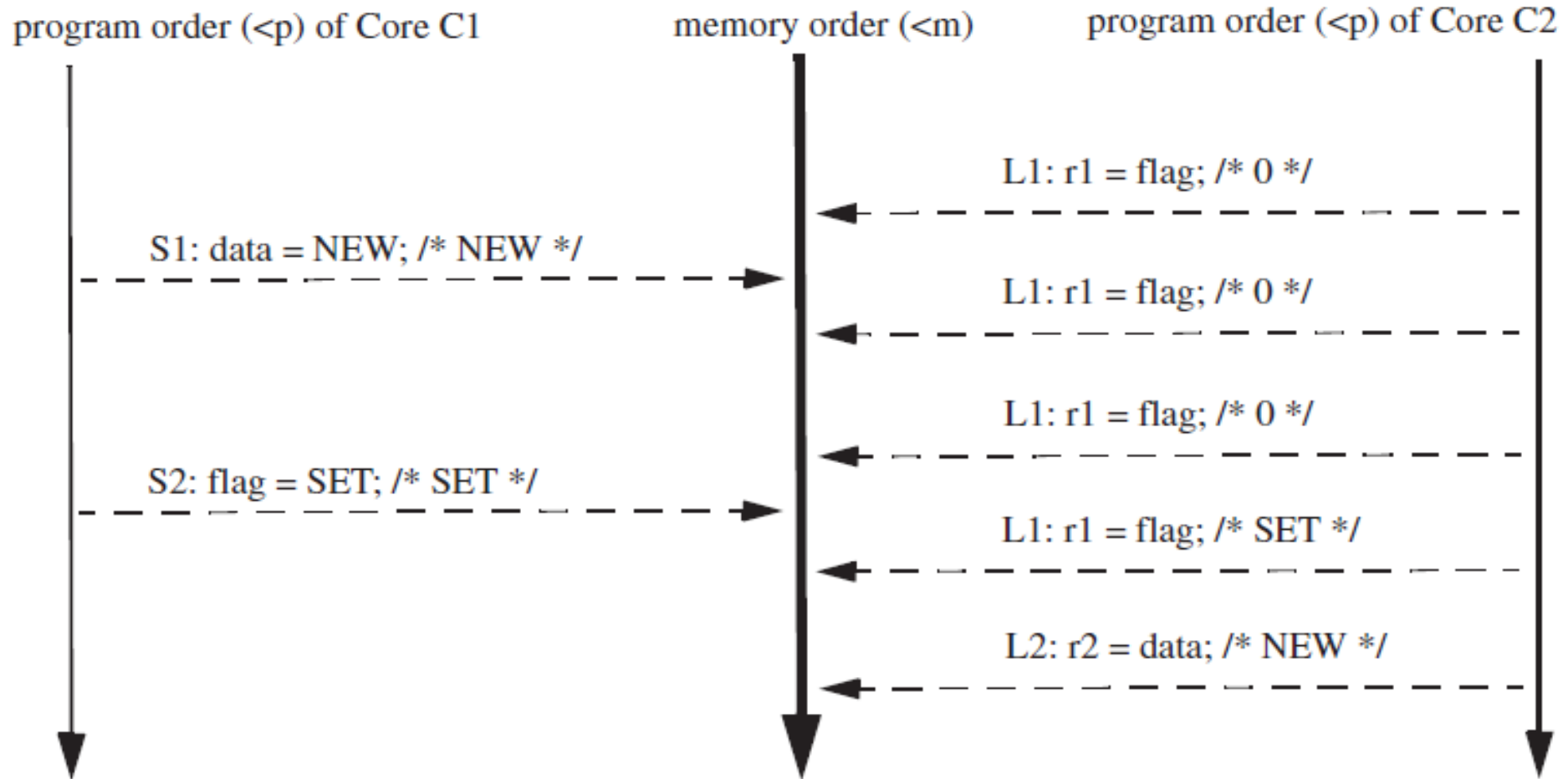
# Interleavings with SC

**TABLE 3.1:** Should r2 Always be Set to NEW?

<b>Core C1</b>	<b>Core C2</b>	<b>Comments</b>
S1: Store data = NEW; S2: Store flag = SET;	L1: Load r1 = flag; B1: if (r1 ≠ SET) goto L1; L2: Load r2 = data;	/* Initially, data = 0 & flag ≠ SET */ /* L1 & B1 may repeat many times */



# Interleavings with SC



# SC Formalism

Every load gets its value from the last store before it (in global memory order) to the same address

# SC Rules

Suppose we  
have two  
addresses a  
and b

- $a == b$  or  $a != b$

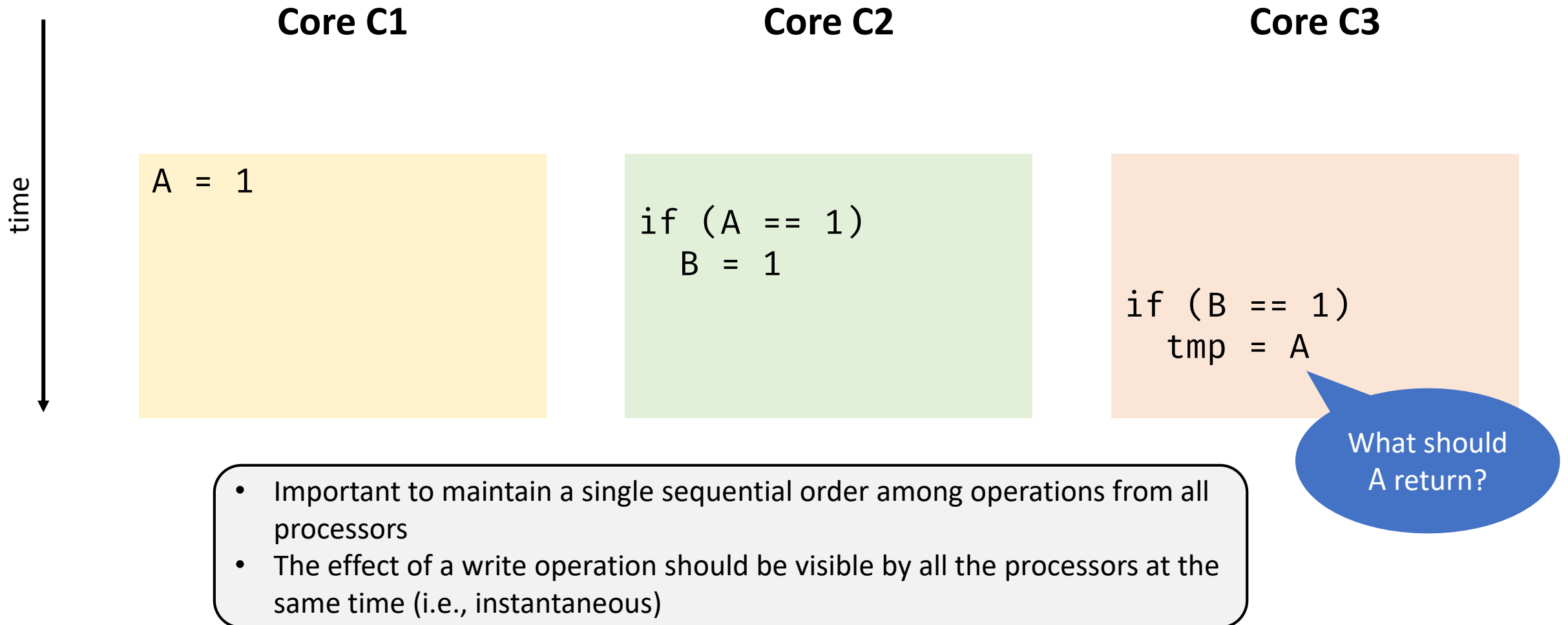
Constraints

- if  $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
- If  $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$
- If  $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$
- If  $S(a) <_p L(b) \Rightarrow S(a) <_m L(b)$

# Challenges in Implementing SC

- Is preserving program order on a per-location basis sufficient?
- Hardware implementations of SC need to satisfy
  1. Program order requirement
    - Previous memory operation completes before proceeding with the next memory operation in program order
  2. Write atomicity requirement
    - Writes to the same location should be serialized, i.e., writes to the same location should be visible in the same order to all processors

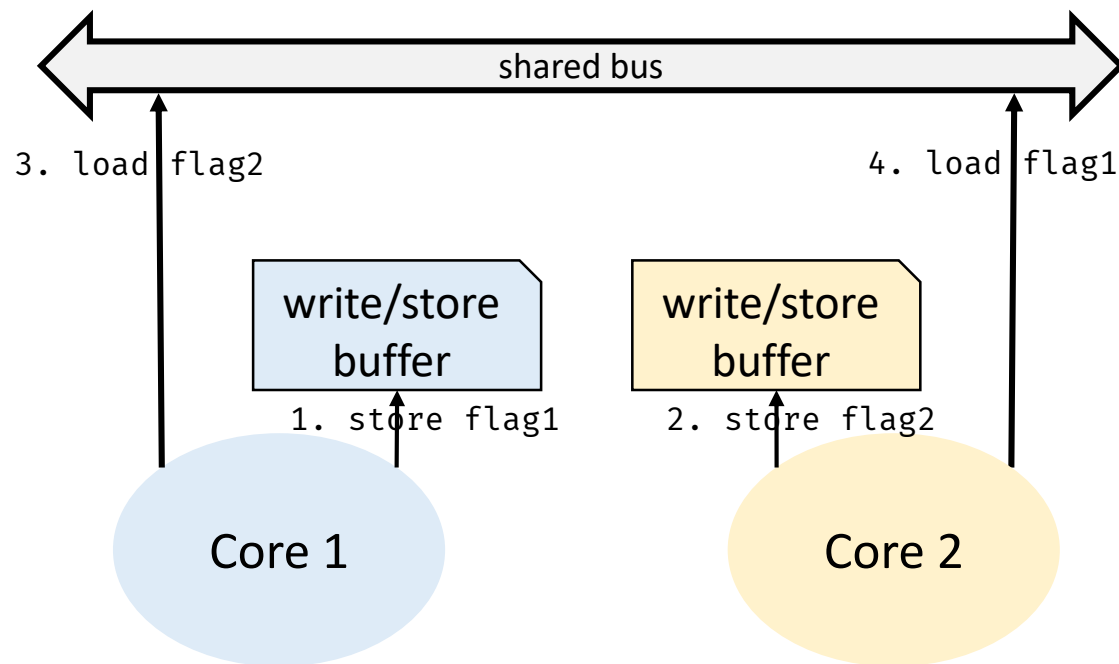
# Need for Write Atomicity



# Importance of Maintaining Write-Read Order

- Assume a bus-based system with no caches
- Includes a write buffer with bypassing capabilities

flag1 = 0  
flag2 = 0



**Core 1**

S1: store flag1, 1  
L1: load r1, flag2

**Core 2**

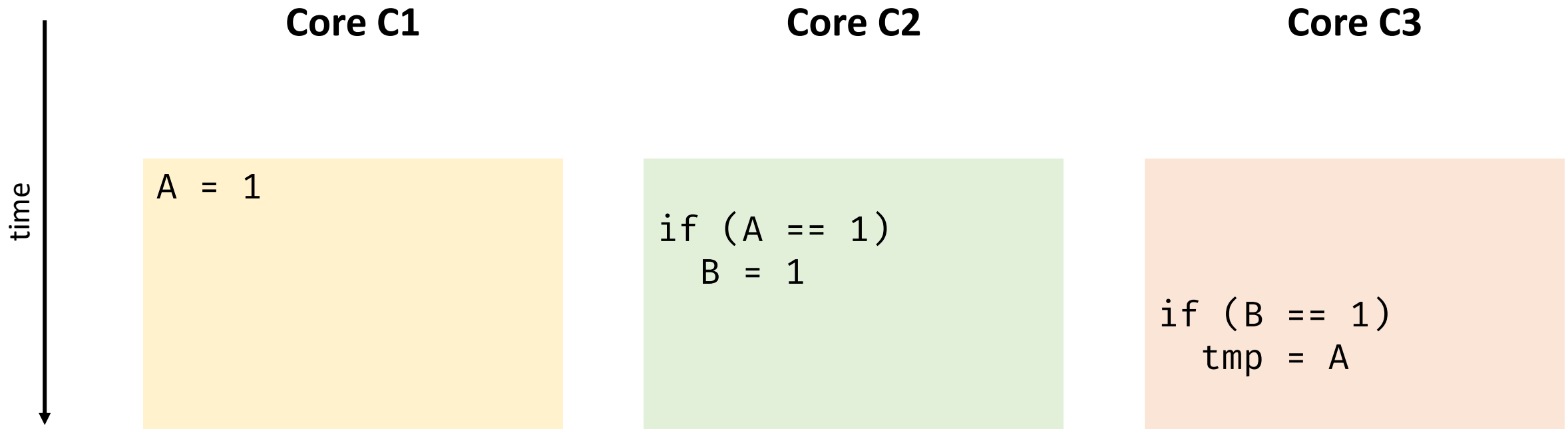
S2: store flag2, 1  
L2: load r2, flag1

# SC in Architecture with Caches

- Replication of data requires a cache coherence protocol
  - Several definitions of cache coherence protocols exist
- Propagating new values to multiple other caches is non-atomic

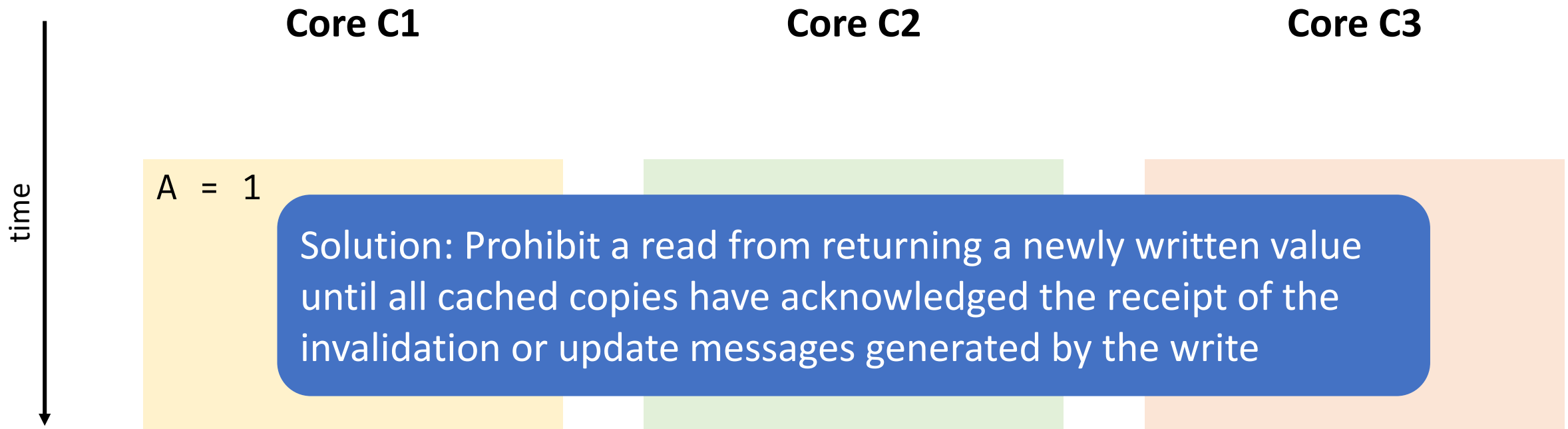
# Providing Write Atomicity with Caches

- Consider a system with caches, and assume that all variables are cached by all the cores
- SC can be violated with a network with no ordering guarantees





# Providing Write Atomicity with Caches



# Serialization of Writes

## Core 1

```
A = 1  
B = 1
```

## Core 2

```
A = 2  
C = 1
```

## Core 3

```
while (B != 1) {}  
while (C != 1) {}  
tmp1 = A
```

## Core 4

```
while (B != 1) {}  
while (C != 1) {}  
tmp2 = A
```

Writes to A in Cores 1 and 2 should not reach Cores 3 and 4 out of order even if the network is out of order or does not provide guarantees

- That would violate SC

# Serialization of Writes

**Core 1**

```
A = 1  
B = 1
```

**Core 2**

```
A = 2  
C = 1
```

**Core 3**

```
while (B != 1) {}  
while (C != 1) {}  
tmp1 = A
```

**Core 4**

```
while (B != 1) {}  
while (C != 1) {}  
tmp2 = A
```

Solution: Cache coherence must serialize writes to the same memory location

Writes to the same memory location must be seen in the same order by all

# Cache Coherence

Single writer multiple readers (SWMR)

Memory updates are passed correctly, cached copies always contain the most recent data

Virtually a synonym for SC, but for a single memory location

Alternate definition based on relaxed ordering

- A write is **eventually** made visible to all processors
- Writes to the **same** location appear to be seen in the same order by all processors (serialization)
  - SC - \*all\*

# Memory Consistency vs Cache Coherence

## Memory Consistency

- **Defines** shared memory behavior
- Related to **all** shared-memory locations
- Policy on **when** new value is propagated to other cores
- Memory consistency implementations can use cache coherence as a “black box”

## Cache Coherence

- **Does not define** shared memory behavior
- Specific to a **single** shared-memory location
- **Propagate** new value to other cached copies
  - Invalidation-based or update-based

# End-to-end SC

Simple memory model that can be implemented both in hardware and in languages

Performance can take a hit

- Naive hardware
- Maintain program order - expensive for a write

# SC-Preserving Optimizations

- Redundant load

$t = X; u = X; \Rightarrow t = X; u = t;$

- Forwarded load

$X = t; u = X; \Rightarrow X = t; u = t;$

- Dead store

$X = t; X = u; \Rightarrow X = u;$

- Redundant store

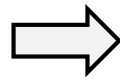
$t = X; X = t; \Rightarrow t = X;$

# Optimizations Forbidden in SC

- Loop invariant code motion
- Common sub-expression elimination
- ...

**Original**

```
L1: t = X*2  
L2: u = Y  
L3: v = X*2
```



**Optimized**

```
L1: t = X*2  
L2: u = Y  
O3: v = t
```

CSE reorders the memory accesses to Y and the second read from X (relaxes L->L constraint, performs an eager load)



```
X = 0
Y = 0
```

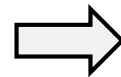
# Optimizations Forbidden in SC

- Loop invariant code motion
- Common sub-expression elimination
- ...

`u == 1 && v == 0`  
is not possible in the  
original code

**Original**

```
L1: t = X*2
L2: u = Y
L3: v = X*2
```



**Optimized**

```
L1: t = X*2
L2: u = Y
O3: v = t
```

**Concurrent  
Thread**

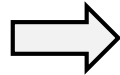
```
C1: X = 1
C2: Y = 1
```

# Problematic Optimizations with SC

**Original**

**Optimized**

L1: X = 1  
L2: P = Q  
L3: t = X



L1: X = 1  
L2: P = Q  
L3: t = 1

Constant/copy  
propagation

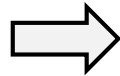
Eager load optimizations involve S->L and L->L reordering. These optimizations perform a load earlier than would have been performed without the optimizations.

# Problematic Optimizations with SC

**Original**

**Optimized**

L1: X = 1  
L2: P = Q  
L3: X = 2



L1: ;  
L2: P = Q  
L3: X = 2

Dead store

L1: t = X  
L2: P = Q  
L3: X = t



L1: t = X  
L2: P = Q  
L3: ;

Redundant store

# Implementing SC with Compiler Support

- Idea: Implement a compiler pass (e.g., in LLVM) to deal with non-SC preserving optimizations

```
L1: t = X*2
L2: u = Y
L3: v = X*2    →    L1: t = X*2
                  L2: u = Y
                  L3: v = t
                  C3: if (X modified since L1)
                  L3:   v = X*2
```

# SC Semantics

## Program semantics

- SC does not guarantee data race freedom
- Not a **strong** memory model

```
a++;
```

```
buffer[index]++;
```

# Questions

- How would you implement an RMW instruction with SC?
- Are memory models only relevant in systems with support for caches?
- Is memory consistency not needed in presence of cache coherence?
- Do memory models only impact hardware design?

# Hardware Memory Models

---

# Characterizing Hardware Memory Models

## Relax program order

- Store  $\rightarrow$  Load, Store  $\rightarrow$  Store, ...
- Applicable to pairs of operations with different addresses

## Relax write atomicity

- Read other core's write early
- Applicable to only cache-based systems

## Relax both program order and write atomicity

- Read own write early



# Possible Interleavings Under SC and TSO

**TABLE 3.3:** Can Both r1 and r2 be Set to 0?

<b>Core C1</b>	<b>Core C2</b>	<b>Comments</b>
S1: x = NEW; L1: r1 = y;	S2: y = NEW; L2: r2 = x;	<i>/* Initially, x = 0 &amp; y = 0*/</i>

# Total Store Order

Allows reordering stores to loads

- Operational model of TSO uses a store buffer for each processor

Requires write atomicity, can read own write early, not other's writes

- A read is not allowed to return the value of another processor's write until it is made visible to all other processors (as in SC)

Conjecture: widely-used x86 memory model is equivalent to TSO

# TSO Rules

**a == b or a != b**

- If  $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
- If  $L(a) <_p S(b) \Rightarrow L(a) <_m S(b)$
- If  $S(a) <_p S(b) \Rightarrow S(a) <_m S(b)$
- ~~If  $S(a) <_p L(b) \Rightarrow S(a) <_m L(b)$  /\* Enables FIFO Write Buffer \*/~~

Every load gets its value from the last store before it to the same address

# Support for FENCE Operations in TSO

If  $L(a) <_p \text{FENCE} \Rightarrow L(a) <_m \text{FENCE}$

If  $S(a) <_p \text{FENCE} \Rightarrow S(a) <_m \text{FENCE}$

If  $\text{FENCE} <_p \text{FENCE} \Rightarrow \text{FENCE} <_m \text{FENCE}$

If  $\text{FENCE} <_p L(a) \Rightarrow \text{FENCE} <_m L(a)$

If  $\text{FENCE} <_p S(a) \Rightarrow \text{FENCE} <_m S(a)$

If  $S(a) <_p \text{FENCE} \Rightarrow S(a) <_m \text{FENCE}$

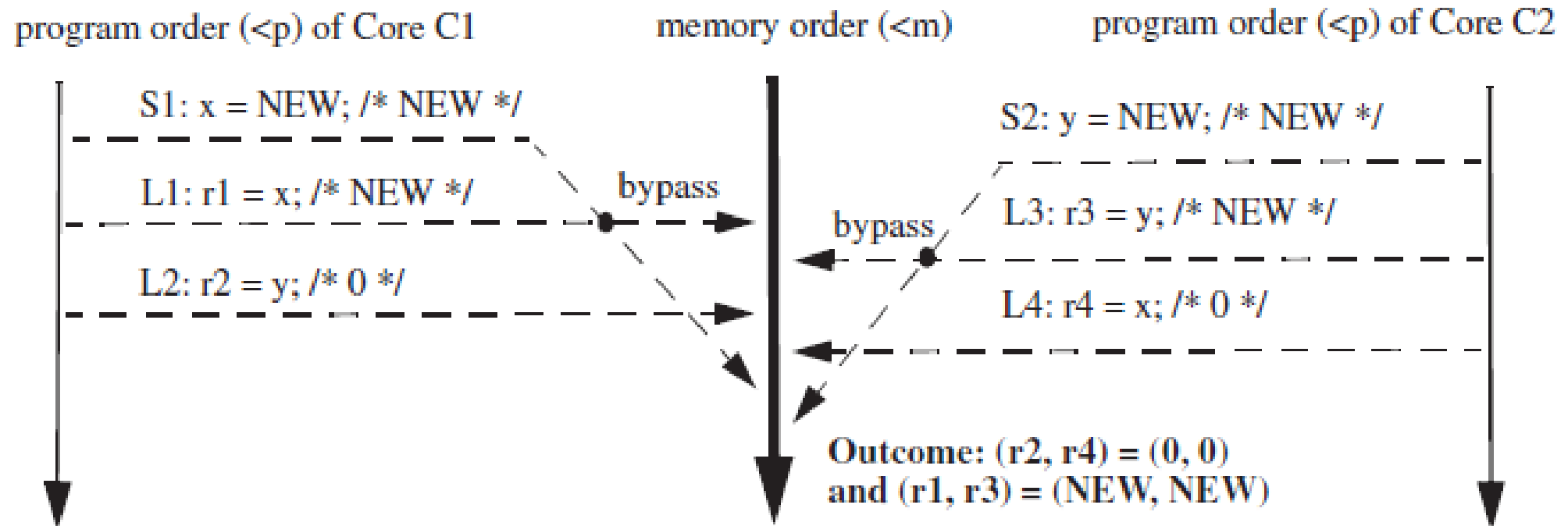
If  $\text{FENCE} <_p L(a) \Rightarrow \text{FENCE} <_m L(a)$

# Possible Outcomes with TSO

**TABLE 4.3:** Can r1 or r3 be Set to 0?

<b>Core C1</b>	<b>Core C2</b>	<b>Comments</b>
S1: x = NEW; L1: r1 = x; L2: r2 = y;	S2: y = NEW; L3: r3 = y; L4: r4 = x;	/* Initially, x = 0 & y = 0*/  /* Assume r2 = 0 & r4 = 0 */

# Possible Outcomes with TSO



# RMW in TSO

Load of a RMW **cannot** be performed until earlier stores are performed (i.e., exited the write buffer). Why?

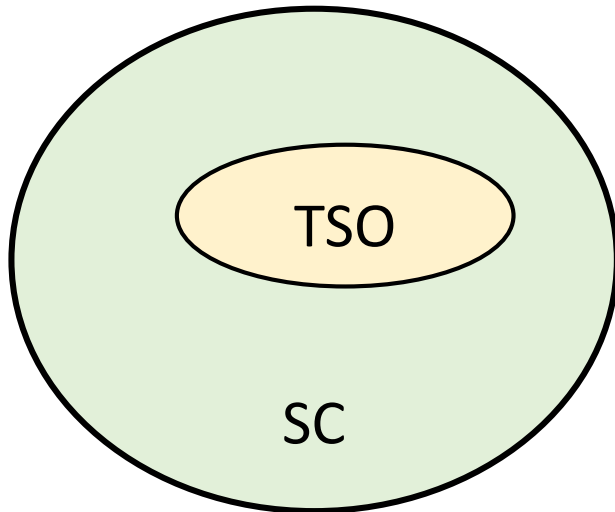
- Effectively drains the write buffer

Load requires read–write coherence permissions, not just read permissions

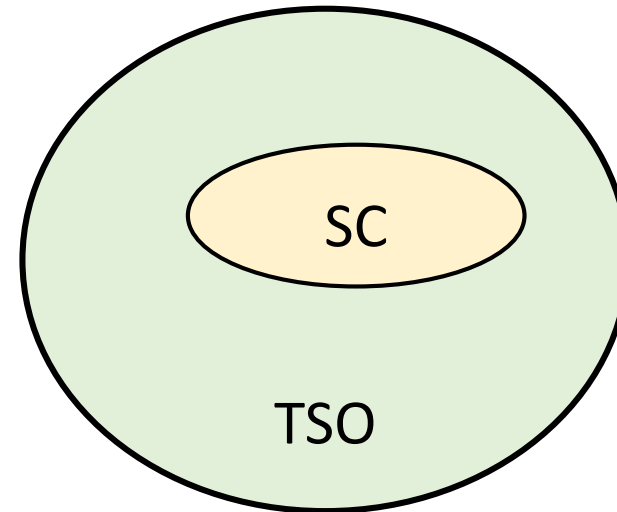
To guarantee atomicity, the cache controller may not relinquish coherence permission to the block between the load and the store

# Relationship between SC and TSO

**Correct?**



**Correct?**





# Relationship among Memory Models

- A memory model Y is strictly more relaxed (weaker) than a memory model X if all X executions are also Y executions, but not vice versa
- If Y is more relaxed than X, then all X implementations are also Y implementations
- Two memory models may be incomparable if both allow executions precluded by the other

# Processor Consistency (PC) and Partial Store Order (PSO)

- PC is similar to TSO, but does not guarantee write atomicity
  - Writes may become visible to different processors in different order
- PSO allows reordering of store to loads and stores to stores
  - Writes to **different** locations from the same processor can be pipelined or overlapped and are allowed to reach memory or other cached copies out of program order
  - Can read own write early, not other's writes
  - Write-write reordering is present in many architectures, including Alpha, IA64, and POWER

# Opportunities to Reorder Memory Operations

**TABLE 5.1:** What Order Ensures r2 & r3 Always Get NEW?

<b>Core C1</b>	<b>Core C2</b>	<b>Comments</b>
S1: data1 = NEW; S2: data2 = NEW; S3: flag = SET;	L1: r1 = flag; B1: if (r1 ≠ SET) goto L1; L2: r2 = data1; L3: r3 = data2;	/* Initially, data1 & data2 = 0 & flag ≠ SET */  /* spin loop: L1 & B1 may repeat many times */

# Reorder Operations Within a Synchronization Block

**TABLE 5.2:** What Order Ensures Correct Handoff from Critical Section 1 to 2?

Core C1	Core C2	Comments
A1: acquire(lock) /* Begin Critical Section 1 */ Some loads L1i interleaved with some stores S1j /* End Critical Section 1 */ R1: release(lock)	A2: acquire(lock) /* Begin Critical Section 2 */ Some loads L2i interleaved with some stores S2j /* End Critical Section 2 */ R2: release(lock)	/* Arbitrary interleaving of L1i's & S1j's */  /* Handoff from critical section 1*/ /* To critical section 2*/  /* Arbitrary interleaving of L2i's & S2j's */

# Optimization Opportunities

Non-FIFO coalescing write buffer

Support non-blocking reads

- Hide latency of reads
- Use lockup-free caches and speculative execution

Simpler support for speculation

- Need not compare addresses of loads to coherence requests
- For SC, need support to check whether the speculation is correct

# Relaxed Consistency Rules

If  $L(a) <_p \text{ FENCE} \Rightarrow L(a) <_m \text{ FENCE}$

If  $S(a) <_p \text{ FENCE} \Rightarrow S(a) <_m \text{ FENCE}$

If  $\text{FENCE} <_p \text{ FENCE} \Rightarrow \text{FENCE} <_m \text{ FENCE}$

If  $\text{FENCE} <_p L(a) \Rightarrow \text{FENCE} <_m L(a)$

If  $\text{FENCE} <_p S(a) \Rightarrow \text{FENCE} <_m S(a)$

# Relaxed Consistency Rules

Maintain TSO rules for ordering two accesses to the **same address only**

- If  $L(a) <_p L'(a) \Rightarrow L(a) <_m L'(a)$
- If  $L(a) <_p S(a) \Rightarrow L(a) <_m S(a)$
- If  $S(a) <_p S'(a) \Rightarrow S(a) <_m S'(a)$

Every load gets its value from the last store before it to the **same address**

# Correct Implementation under Relaxed Consistency

**TABLE 5.3:** Adding FENCEs for XC to Table 5.1's Program.

Core C1	Core C2	Comments
S1: data1 = NEW; S2: data2 = NEW; <b>F1: FENCE</b> S3: flag = SET;	L1: r1 = flag; B1: if (r1 ≠ SET) goto L1; <b>F2: FENCE</b> L2: r2 = data1; L3: r3 = data2;	<i>/* Initially, data1 &amp; data2 = 0 &amp; flag ≠ SET */</i>  <i>/* L1 &amp; B1 may repeat many times */</i>

Are both FENCEs required?



# Correct Implementation under Relaxed Consistency

TABLE 5.4: Adding FENCES for XC to Table 5.2's Critical Section Program.

Core C1	Core C2	Comments
<b>F11: FENCE</b> A11: acquire(lock) <b>F12: FENCE</b> Some loads L1i interleaved with some stores S1j <b>F13: FENCE</b> R11: release(lock) <b>F14: FENCE</b>	<b>F21: FENCE</b> A21: acquire(lock) <b>F22: FENCE</b> Some loads L2i interleaved with some stores S2j <b>F23: FENCE</b> R22: release(lock) <b>F24: FENCE</b>	<i>/* Arbitrary interleaving of L1i's &amp; S1j's */</i>  <i>/* Handoff from critical section 1*/</i> <i>/* To critical section 2*/</i>  <i>/* Arbitrary interleaving of L2i's &amp; S2j's */</i>

# Examples of Relaxed Consistency Memory Models

## Weak ordering

- Distinguishes between **data and synchronization operations**
- A synchronization operation is not issued until all previous operations are complete
- No operations are issued until the previous synchronization operation completes

# Correct Implementation under Relaxed Consistency

Which fences are needed to ensure correct ordering and visibility between C1 and C2?

**TABLE 5.4:** Adding FENCES for XC

Core C1	Core C2
<b>F11: FENCE</b> A11: acquire(lock)	
<b>F12: FENCE</b> Some loads L1i interleaved with some stores S1j	<i>/* Arbitrary interleaving of L1i's &amp; S1j's */</i>
<b>F13: FENCE</b> R11: release(lock)	<b>F21: FENCE</b> <i>/* Handoff from critical section 1*/</i> A21: acquire(lock) <i>/* To critical section 2*/</i>
<b>F14: FENCE</b>	<b>F22: FENCE</b> <i>/* Arbitrary interleaving of L2i's &amp; S2j's */</i> Some loads L2i interleaved with some stores S2j
	<b>F23: FENCE</b> R22: release(lock)
	<b>F24: FENCE</b>

# Relaxed Consistency Memory Models

## Release consistency

- Relaxes WO further, distinguishes between **acquire** and **release** synchronization operations
- All previous acquire operations must be performed before an ordinary load or store access is allowed to perform
- Previous accesses have to complete before a release is performed
- $RC_{sc}$  – maintains maintains SC between synchronization operations
- Acquire  $\rightarrow$  all, all  $\rightarrow$  release, and sync  $\rightarrow$  sync

# Relaxed Consistency Memory Models

---

Why should we  
use them?

**Performance**

---

Why should we  
not use them?

**Complexity**

---

# Hardware Memory Models: One Slide Summary

Model	W->R	W->W	R->RW	Read Own Write Early	Read Other's Write Early
SC				✓	
TSO	✓			✓	
PC	✓			✓	✓
PSO	✓	✓		✓	
WO	✓	✓	✓	✓	
RC <sub>SC</sub>	✓	✓	✓	✓	
RC <sub>PC</sub>	✓	✓	✓	✓	✓

# Desirable Properties of a Memory Model

Hard to satisfy all three properties

- Programmability
- Performance
- Portability

## Think of SC

Pros

- Intuitive when we think of uniprocessor executions
- Serializability of instructions

Cons

- No atomicity of regions
- Inhibits many compiler transformations
- Almost all recent architectures violate SC

# Programming Language Memory Models

---



# Data-Race-Free-0 (DRF0) Model

- Conceptually similar to Weak Ordering
- Assumes no data races
  - **DRF0 ensures SC for data-race-free programs**
  - No guarantees for racy programs
- Allows many optimizations in the compiler and hardware

# Language Memory Models

Developed much later than hardware models

- Recent standardizations are largely driven by languages

Most language models are based on DRF0

Why do we need one? Is the hardware memory model not enough?

# Discussion about C++

# C++ Memory Model

- Adaptation of the DRF0 memory model
  - Provides SC for data-race-free programs
- **C/C++ simply ignore data races**
  - No safety guarantees in the language

# Catch-Fire Semantics in C++

```
X* x = NULL;  
bool done= false;
```

**Thread T1**

```
x = new X();  
done = true;
```

**Thread T2**

```
if (done) {  
    x->func();  
}
```

# Catch-Fire Semantics in C++

```
X* x = NULL;  
bool done= false;
```

Thread T1

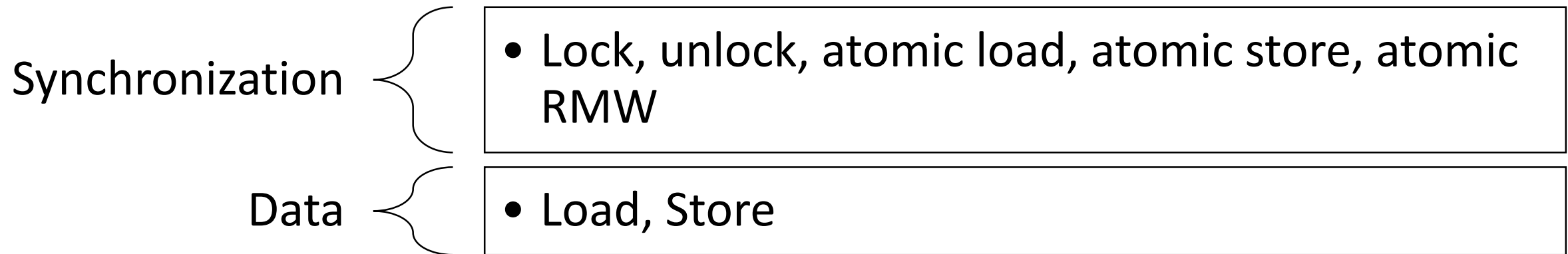
Thread T2

```
x = new X();  
done = true;
```

```
if (done) {
```



# Memory Operations in C++



# Reordering of Memory Operations in C++

---

Compiler  
reordering  
**allowed** for  
M1 and M2

M1 is a data operation and M2 is a read synchronization operation

---

M1 is write synchronization and M2 is data

---

M1 and M2 are both data with no synchronization between them

---

M1 is data and M2 is the write of a lock operation

---

M1 is unlock and M2 is either a read or write of a lock

---



# Writing Correct C++ Code

- Mutual exclusion of critical code blocks

```
std::mutex mtx;  
{  
    mtx.lock();  
    // access shared data here  
    mtx.unlock();  
}
```

- Mutex provides inter-thread synchronization
  - Unlock() synchronizes with calls to lock() on the **same mutex object**

# Synchronize Using Locks

```
std::mutex mtx;  
bool dataReady = false;
```

```
{  
    mtx.lock();  
    prepareData();  
    dataReady = true;  
    mtx.unlock();  
}
```

```
{  
    mtx.lock();  
    if (dataReady) {  
        consumeData();  
    }  
    mtx.unlock();  
}
```

# Synchronize Using Locks

```
std::mutex mtx;  
bool dataReady = false;
```

```
{  
    mtx.lock();  
    prepareData();  
    dataReady = true;  
    mtx.unlock();  
}
```

```
bool b;  
{  
    mtx.lock();  
    b = dataReady;  
    mtx.unlock();  
}  
if (b) {  
    consumeData();  
}
```

# Using Atomics from C++11

- “Data race free” by definition
  - E.g., `std::atomic<int>`
  - A store synchronizes with operations that load the stored value
  - Similar to `volatile` in Java
- C++ `volatile` is different!
  - Does not establish inter-thread synchronization
  - Can be part of a data race

```
reg_var1 = reg_var2;
```



```
atm_var1.store(atm_var2.load());
```

---

```
std::mutex mtx;  
std::atomic<bool> ready(false);
```

```
prepareData();  
ready.store(true);
```

```
if (ready.load()) {  
    consumeData();  
}
```

# Visibility and Ordering

## Visibility

- When are the effects of one thread visible to another?

## Ordering

- When can operations of any given thread appear out of order to another thread?

# Ensuring Visibility

- Writer thread releases a lock
  - Flushes all writes from the thread's working memory
- Reader thread acquires a lock
  - Forces a (re)load of the values of the affected variables
- `std::atomic` (C++) and `volatile` (Java)
  - Values written are made visible immediately before any further memory operations
  - Readers reload the value upon each access
- Thread join
  - Parent thread is guaranteed to see the effects made by the child thread

# Memory Order of Atomics

- Specifies how regular, non-atomic memory accesses are to be ordered around an atomic operation
- **Default** is sequential consistency

## **atomic.h**

```
enum memory_order {  
    memory_order_relaxed,  
    memory_order_consume,  
    memory_order_acquire,  
    memory_order_release,  
    memory_order_acq_rel,  
    memory_order_seq_cst  
};
```

# Memory Model Synchronization Modes

- Producer thread creates data
- Producer thread stores to an atomic
- Consumer threads read from the atomic
- When the expected value is seen, data from the producer thread is complete and visible to the consumer thread

The different memory model modes indicate how strong this data-sharing bond is between threads

---

<http://gcc.gnu.org/wiki/Atomic/GCCMM/AtomicSync>



# Memory Model Modes in C++

```
x = 0;  
y = 0;
```

- `memory_order_seq_cst`

## Thread T1

```
y = 1;  
x.store(2);
```

## Thread T2

```
if (x.load() == 2)  
    assert (y == 1)
```

Can this assert fail?

# Memory Model Modes in C++

```
x = 0;  
y = 0;
```

- `memory_order_seq_cst`

```
y.store(20);  
x.store(10);
```

```
if (x.load() == 10)  
    assert (y.load() == 20);  
y.store(10);
```

```
if (y.load() == 10)  
    assert (x.load() == 10)
```

Can these  
asserts fail?

```
x = 0;  
y = 0;
```

# Memory Model Modes in C++

- `memory_order_relaxed`: no happens-before edges

```
y.store(20, memory_order_relaxed);  
x.store(10, memory_order_relaxed);
```

Different  
threads

```
if (x.load(memory_order_relaxed) == 10)  
    assert (y.load(memory_order_relaxed) == 20);  
y.store(30, memory_order_relaxed);
```

```
if (y.load(memory_order_relaxed) == 30)  
    assert (x.load(memory_order_relaxed) == 10)
```

Can these asserts fail?

# Memory Model Modes in C++

```
x = 0;  
y = 0;
```

- `memory_order_relaxed`
- Removes HB edges

```
x.store(10, memory_order_relaxed);  
x.store(20, memory_order_relaxed);
```

```
y = x.load(memory_order_relaxed);  
z = x.load(memory_order_relaxed);  
assert (y <= z);
```

Can this assert fail?

- In the absence of HB edges, a thread should not rely on the exact ordering of instructions in another thread
- Once a value of a variable from Thread 1 is observed in Thread 2, Thread 2 cannot see an earlier value

# Memory Model Modes in C++

```
x = 0;  
y = 0;
```

- `memory_order_acquire` and `memory_order_release`
- Only introduces HB edges between dependent variables

y is a regular  
data variable

```
y = 20;  
x.store(10, memory_order_release);
```

```
if (x.load(memory_order_acquire) == 10)  
    assert (y == 20);
```

Can this assert  
fail?

```
x = 0;  
y = 0;
```

# Memory Model Modes in C++

- `memory_order_acquire` and `memory_order_release`

```
y.store(20, memory_order_release);
```

```
x.store(10, memory_order_release);
```

```
assert (y.load(memory_order_acquire) == 20 && x.load(memory_order_acquire) == 0);
```

```
assert (y.load(memory_order_acquire) == 0 && x.load(memory_order_acquire) == 10);
```

What will happen with the asserts?

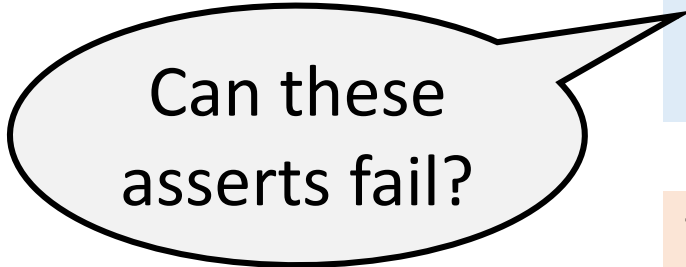
# Memory Model Modes in C++

```
x = 0;  
y = 0;
```

- `memory_order_consume`
- Removes HB ordering on non-dependent shared variables

```
n = 1;  
m = 1;  
p.store(&n, memory_order_release);
```

```
t = p.load(memory_order_acquire);  
assert (*t == 1 && m == 1);
```



Can these  
asserts fail?

```
t = p.load(memory_order_consume);  
assert (*t == 1 && m == 1);
```

# Memory Model Modes in C++

```
x = 0;  
y = 0;
```

- `memory_order_consume`
- Removes HB ordering on non-dependent shared variables

Performance gain depends on how much state the hardware has to flush in order to synchronize

- A consume operation may potentially execute faster

```
n = 1;  
m = 1;  
p.store(&n, memory_order_release);
```

```
t = p.load(memory_order_acquire);  
assert (*t == 1 && m == 1);
```

```
t = p.load(memory_order_consume);  
assert (*t == 1 && m == 1);
```



# Discussion about Java

# Happens-Before Memory Model (HBMM)

- Read operation  $a = rd(t, x, v)$  may return the value written by any write operation  $b = wr(t, x, v)$  provided
  - i.  $b$  does not happen after  $a$ , i.e.,  $b \prec_{HB} a$  or  $b \parallel a$
  - ii. There is no intervening write  $c$  to  $x$  where  $b \prec_{HB} c \prec_{HB} a$

# HBMM

```
x = 0;  
y = 0;
```

```
y = 1;  
r1 = x;
```

```
x = 1;  
r2 = y;
```

```
assert r1 != 0 || r2 != 0
```

---

```
r1 = x;  
y = 1;
```

```
r2 = y;  
x = 1;
```

```
assert r1 == 0 || r2 == 0
```

Can these asserts fail?

# HBMM

```
x = 0;  
y = 0;
```

```
r = x;  
y = 1;  
assert (r == 0);
```

Will the assertion  
pass or fail?

```
while (y == 0) {}  
x = 1;
```

```
x = 0;
```

# HBMM

```
x = 10;
```

```
if (x != 0)  
    r2 = r1 / x;
```

Can anything go wrong?

```
x = 0;  
y = 0;
```

# HBMM

- Potential for out-of-thin-air values

```
x = y;
```

```
y = x;
```

- Problematic for garbage-collected languages since the “out of thin air” value `val` could be an invalid pointer
- Introduces potential security loopholes

```
x = 0;  
y = 0;
```

# DRFO vs HBMM

```
r1 = x;  
if (r1 == 1)  
    y = 1;
```

```
r2 = y;  
if (r2 == 1)  
    x = 1;
```

```
assert r1 == 0 && r2 == 0
```

Is there a data race on x and y?

Remember that DRFO provides SC for data-race-free programs.

# DRFO vs HBMM

DRFO allows **arbitrary** behavior for racy executions

- DRFO is not strictly stronger than HBMM

HBMM does not guarantee SC for DRF programs

- HBMM is not strictly stronger than DRFO
- Example on next slide



# Java Memory Model (JMM)

- First high-level language to incorporate a relaxed memory model
- JMM provides SC for data-race-free executions (like DRF0)
- Provides memory- and type-safety, so **has to** define some semantics for data races
  - JMM prohibits out-of-thin-air values

```
obj = null  
x = 0;  
y = 0;
```

# Outcomes Possible with JMM

- **Racy** Initialization

```
obj = new Circle();
```

```
if (obj != null)  
    obj.draw();
```

Can there be a  
NPE with JMM?

JVMs may not exhibit all behaviors  
permissible under the JMM

# Outcomes Possible with Java

```
x = 0;  
y = 0;
```

```
y = 1;  
r1 = x;
```

```
x = 1;  
r2 = y;
```

```
assert r1 != 0 || r2 != 0
```

---

```
r1 = x;  
y = 1;
```

```
r2 = y;  
x = 1;
```

```
assert r1 == 0 || r2 == 0
```

Can these asserts fail?

```
x = 0;  
y = 0;
```

# Outcomes Not Possible with JMM

```
r1 = x;  
y = r1;
```

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

```
assert r1 != 42
```

- HBMM permits an execution in which each load reads say 42
- DRF0 allows any arbitrary behavior
- JMM is strictly stronger than DRF0 and HBMM

```
x = 0;  
y = 0;
```

# JVMs do not comply with the JMM!!!

```
r1 = x;  
y = r1;
```

```
r2 = y;  
if (r2 == 1) {  
    r3 = y;  
    x = r3;  
} else {  
    x = 1;  
}
```

```
assert r2 == 0
```

Can this assert fail under HBMM and JMM?

- HBMM allows OOTA values
- JMM only permits executions in which load of y sees 0
- JVM's JIT optimizing compiler can simplify the code in the right thread

# Lessons Learnt

SC for DRF is now the preferred baseline

- Make sure your program is free of data races
- System guarantees SC execution

Specifying semantics for racy programs is hard

Simple optimizations may introduce unintended consequences

# References

- S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. WRL Research Report, 1995.
- D. Sorin et al. A Primer on Memory Consistency and Cache Coherence
- D. Marino et al. A Case for an SC-Preserving Compiler. PLDI 2011.
- C. Flanagan and S. Freund. Adversarial Memory for Detecting Destructive Races. PLDI 2010.
- M. Cao et al. Prescient Memory: Exposing Weak Memory Model Behavior by Looking into the Future. ISMM 2016.