# CS 636: Memory Consistency Models

**Swarnendu Biswas**

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2025-26-II

# 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 Tutorials. Journal of Computer, vol. 29, no. 12, pp. 66–76, Dec. 1996.

# Busy-Wait Paradigm

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

**Thread 1**

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

**Thread 2**

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

# Possible Errors

**Thread 1**

```
1 X = new Object();
2
3
4 done = true;
```

**Thread 2**

```
1
2 tmp = done;
3 while (!temp) {}
4
```

Infinite loop

**Thread 1**

```
1 done = true;
2
3
4 X = new Object();
```

**Thread 2**

```
1
2 while (!done) {}
3 X.compute();
4
```

NPE

# Reordering of Accesses by Hardware

Accesses are to **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 sore) |
| 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

Accesses are to **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 sore)

Load-load
- C

Load-store
- C

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

- Correct in a single-threaded context
- Non-trivial in a multithreaded context

# What values can a load return?

### Return the “last” write

- Uniprocessor: program order defines the “last” write
- Multiprocessor: operations from different cores/threads are not related by program order

# 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 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

# Issues with Memory Consistency

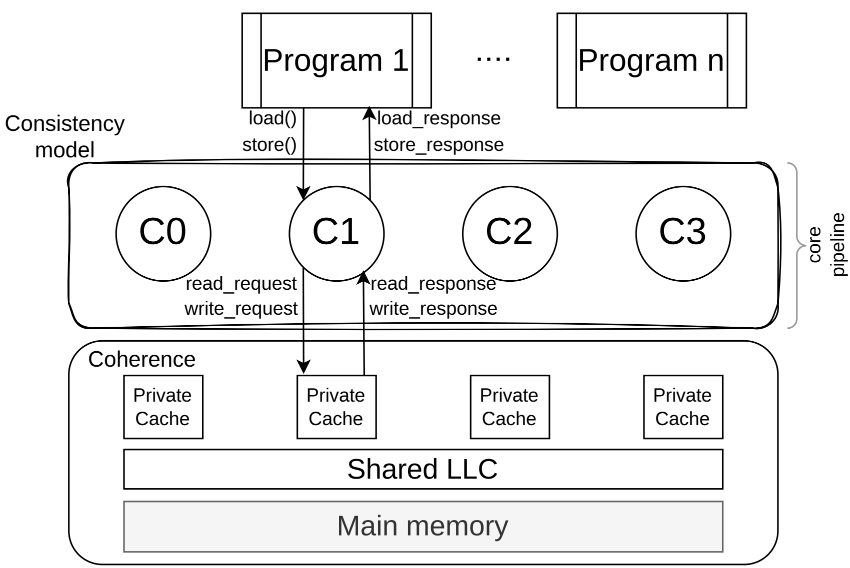
**Visibility**

When are the effects of one thread (e.g., updating a memory location) visible to another?

**Ordering**

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

# Memory Consistency vs Cache Coherence



# 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
- Propagates a new value to other cached copies
- Invalidation-based or update-based

# Sequential Consistency

# Sequential Consistency

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

Uniprocessor
- Memory operations execute in program order, and respect data and control dependences
  - Read from memory returns the value from the last write in program order
  - Compiler optimizations preserve these semantics

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

# Interleavings with SC

```
data = null;
flag = false;
```

**Core 1**

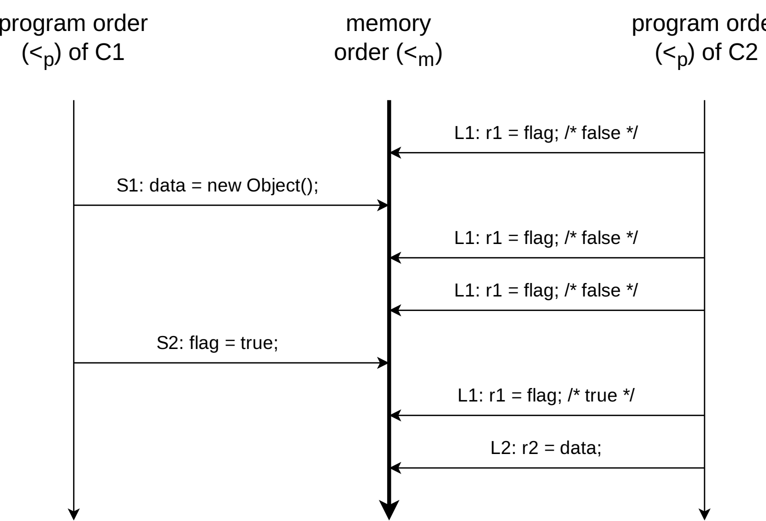
```
S1: data = new Object();
S2: flag = true;

```

**Core 2**

```
L1: r1 = flag;
B1: if (r1 != true) goto L1;
L2: r2 = data;
```

Should r2 always be set to the new `Object()` stored?

# Interleavings with SC



**Note**

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

Suppose we have two addresses a and b (a == b or a != b). L(a) is a load from a and S(a) is a store to a.

Constraints
(i) If L(a) $<_p$ L(b) $\Rightarrow$ L(a) $<_m$ L(b)
(ii) If L(a) $<_p$ S(b) $\Rightarrow$ L(a) $<_m$ S(b)
(iii) If S(a) $<_p$ S(b) $\Rightarrow$ S(a) $<_m$ S(b)
(iv) 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 the following requirements

| | |
|---|---|
| Program order | ▶ Previous memory operation completes before proceeding with the next memory operation in program order |
| Write atomicity | ▶ 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 |

# Dekker's Algorithm: Need for Program Order

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

**Core 1**

```
S1: ST flag1, 1
L1: LD r1, flag2
```

**Core 2**

```
S2: ST flag2, 1
L2: LD r2, flag1
```

Can both r1 and r2 be set to zero?

# Need for Write Atomicity

```
A = B = 0;
```

time ↓

**Core 1**

```
A = 1
```

**Core 2**

```
if (A == 1)
  B = 1
```

**Core 3**

```
if (B == 1)
  tmp = A
```

What should A return?

# Need for Write Atomicity

```
A = B = 0;
```

time

| Core 1 | Core 2 | Core 3 |
| --- | --- | --- |
| `A = 1` | | |

- Important to maintain a single sequential order among operations from all processors
- The effect of a write operation should be visible to all the processors at the same time (i.e., instantaneous)

What should A return?

# 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

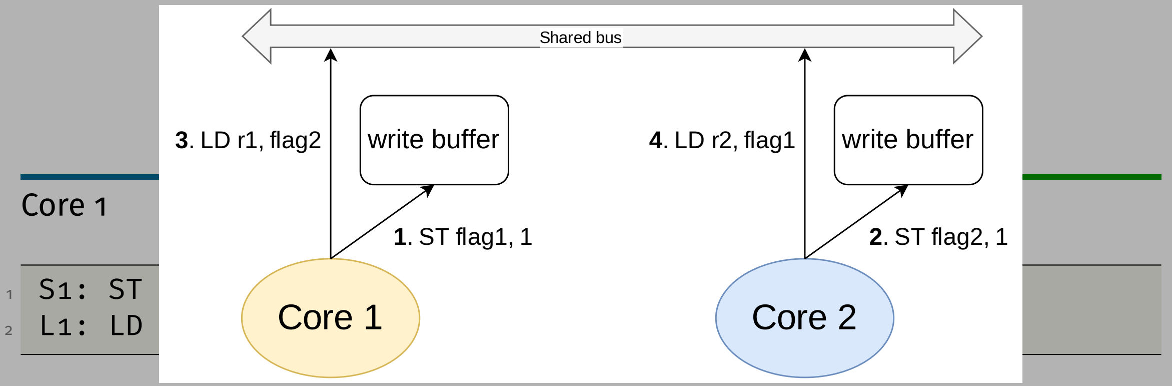
```
S1: ST flag1, 1
L1: LD r1, flag2
```

Core 2

```
S2: ST flag2, 1
L2: LD r2, flag1
```

# Importance of Maintaining Write-Read Order

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

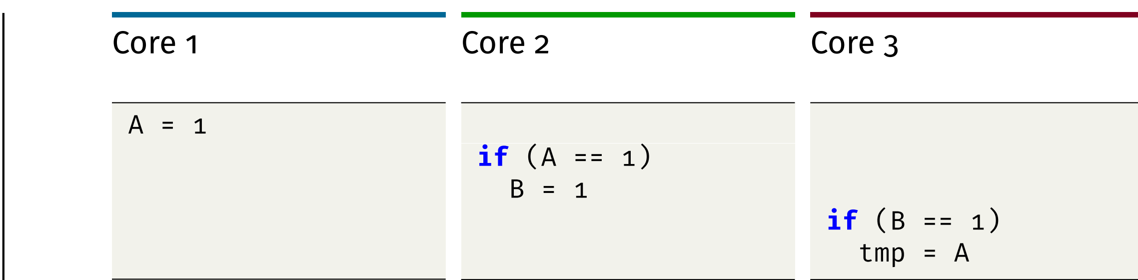


# SC in Architecture with Caches

- Replication of data requires a cache coherence protocol
- A coherence protocol propagates a new value to all other cached copies
  - Several definitions of cache coherence protocols exist
  - A memory model places bounds on **when** the value can be propagated to a given processor
- 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

- 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

Prohibit a read from returning a newly written value until all cached copies have acknowledged the receipt of the invalidation or update messages generated by the write

time

```
  B = 1
```

```
if (B == 1)
  tmp = A
```

# 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—it 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
```

Writes ... even if
the ne ... ate SC

- 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

# End-to-end SC

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

— Performance can take a hit
  - Naïve hardware
  - Maintaining program order can be expensive for writes

D. Marino et al. A Case for an SC-Preserving Compiler. PLDI'11.

# SC-Preserving Optimizations

**Redundant load**

Original

```
t = X; u = X;
```

⇒ Optimized

```
t = X; u = t;
```

**Forwarded load**

Original

```
X = t; u = X;
```

⇒ Optimized

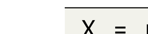
```
X = t; u = t;
```

**Dead store**

Original

```
X = t; X = u;
```

⇒ Optimized



```
X = u;
```

**Redundant store**

Original

```
t = X; X = t;
```

⇒ Optimized

```
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;
```

$\Rightarrow$

**Optimized**

```
L1: t = X*2;
L2: u = Y;
M3: v = t;
```

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

# Optimizations Forbidden in SC

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

**Original**

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

⇒

**Optimized**

```
L1: t = X*2;
L2: u = Y;
M3: v = t;
```

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

**Concurrent Thread**

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

# Problematic Optimizations with SC

Constant/copy propagation

**Original**

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

$\Rightarrow$

**Optimized**

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

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

**Dead store**

**Original**

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

⇒

**Optimized**

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

**Redundant store**

**Original**

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

⇒

**Optimized**

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

# Implementing SC with Compiler Support

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;
```

$\Rightarrow$

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

D. Marino et al. A Case for an SC-Preserving Compiler. PLDI'11.

# SC Semantics

**SC is not a strong memory model**

Does not guarantee data race freedom

**Thread 1**

```
a++;
```

**Thread 2**

```
a++;
```

**Thread 3**

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

**Thread 4**

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

# Hardware Memory Models

# Characterizing Hardware Memory Models

## Relax program order

- For example, Store $\rightarrow$ Load and 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

# Can both r1 and r2 be set to zero?

```
1  x = 0;
2  y = 0;
```

**Core 1**

```
1  S1: x = new Object();
2  L1: r1 = y;
```

**Core 2**

```
1  S2: y = new Object();
2  L2: r2 = x;
```

# Total Store Order

- Allows reordering stores to loads
  - 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)
- Requires write atomicity, can read own write early, not other's writes
- Conjecture: widely-used x86 memory model is equivalent to TSO

# TSO Formalism

Suppose we have two addresses a and b (a == b or a != b)

**Constraints**

1. If L(a) $<_p$ L(b) $\Rightarrow$ L(a) $<_m$ L(b)
2. If L(a) $<_p$ S(b) $\Rightarrow$ L(a) $<_m$ S(b)
3. If S(a) $<_p$ S(b) $\Rightarrow$ S(a) $<_m$ S(b)
4. ~~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) $<_p$ FENCE $\Rightarrow$ L(a) $<_m$ FENCE

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

If FENCE $<_p$ FENCE $\Rightarrow$ FENCE $<_m$ FENCE

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

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

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

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

# Possible Outcomes with TSO

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

**Core 1**

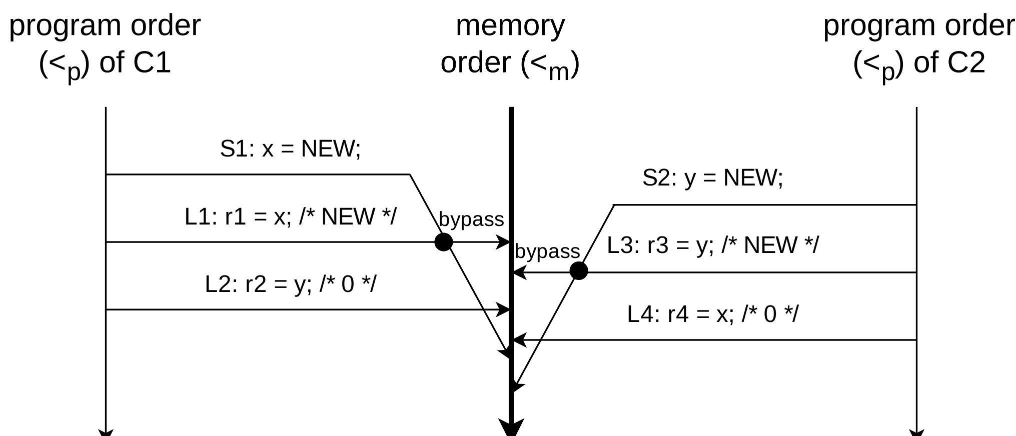
```
S1: x = NEW;
L1: r1 = x;
L2: r2 = y;
```

**Core 2**

```
S2: y = NEW;
L3: r3 = y;
L4: r4 = x;
```

Assume r2 and r4 are zero. Can r1 or r3 also be set to zero?

# Possible Outcomes with TSO




**Outcome**: r2 ==0, r4 == 0, r1 == NEW, and r3 == NEW

# RMW in TSO

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

- 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 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

Which is correct?

# Processor Consistency (PC)

**PC is similar to TSO, but does not guarantee write atomicity**

Writes may become visible to different processors in different order

```
A = B = 0;
```

time ↓

**Core 1**

```
A = 1
```

**Core 2**

```
if (A == 1)
  B = 1
```

**Core 3**

```
if (B == 1)
  tmp = A
```

# Partial Store Order (PSO)

**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

```
data1 = data2 = null;
flag = false;
```

**Core 1**

```
S1: data1 = new Object();
S2: data2 = new Object();
S3: flag = true;
```

**Core 2**

```
L1: r1 = flag;
B1: if (!r1) goto L1;
L2: r2 = data1;
L3: r3 = data2;
```

What order ensures r2 and r3 always see initialized objects?

# Reorder Operations Within a Synchronization Block

## Core 1

```
1  A1: acquire(lock);
2  // Loads L_1i arbitrarily
3  // interleaved with stores S_1j
4  R1: release(lock);
5
6
7
8
```

## Core 2

```
1
2
3
4
5
6  A2: acquire(lock);
7  // Loads L_2i arbitrarily
8  // interleaved with stores S_2j
9  R2: release(lock);
```

What order ensures correct handoff from critical section 1 to 2?

# Optimization Opportunities

(i) Non-FIFO coalescing write buffer

(ii) Support non-blocking reads
- Hide latency of reads
- Use lockup-free caches and speculative execution

(iii) 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

**Loads and stores are unordered excepting TSO rules are followed for ordering two accesses to the same address**

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

Constraints

1. If $L(a) <_p L^{'}(a) \Rightarrow L(a) <_m L^{'}(a)$
2. If $L(a) <_p S(a) \Rightarrow L(a) <_m S(a)$
3. If $S(a) <_p S^{'}(a) \Rightarrow S(a) <_m S^{'}(a)$

# Relaxed Consistency Rules

**Loads and stores are unordered excepting TSO rules are followed for ordering two accesses to the same address**

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

Constraints

1. If $L(a) <_p L^{'}(a) \Rightarrow L(a) <_m L^{'}(a)$
2. If $L(a) <_p S(a) \Rightarrow L(a) <_m S(a)$
3. If $S(a) <_p S^{'}(a) \Rightarrow S(a) <_m S^{'}(a)$

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

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

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

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

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

# Using Fences under Relaxed Consistency

```
1  data1 = null;
2  data2 = null;
3  flag = false;
```

**Core 1**

```
1  S1: data1 = new Object();
2  S2: data2 = new Object();
3  F1: FENCE
4  S3: flag = true;
5
6
7
8
```

**Core 2**

```
1
2
3
4  L1: r1 = flag;
5  B1: if (!r1) goto L1;
6  F2: FENCE
7  L2: r2 = data1;
8  L3: r3 = data2;
```

Are both fences required?

# Conservative Use of Fences under Relaxed Consistency

## Core 1

```
1  F11: FENCE
2  A11: acquire(lock);
3  F12: FENCE
4  // Loads L1i arbitrarily
5  // interleaved with stores S1j
6  F13: FENCE
7  R12: release(lock);
8  F14: FENCE
9
10
11
12
13
14
```

## Core 2

```
1
2
3
4
5
6
7  F21: FENCE
8  A21: acquire(lock);
9  F22: FENCE
10 // Loads L2i arbitrarily
11 // interleaved with stores S2j
12 F23: FENCE
13 R22: release(lock);
14 F24: FENCE
```

# 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

## Release consistency

- Relaxes WO further, distinguishes between acquire and release 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 SC between synchronization operations
- Acquire $\rightarrow$ all, all $\rightarrow$ release, and sync $\rightarrow$ sync

# Correct Implementation under Relaxed Consistency

## Core 1

```
1  F11: FENCE
2  A11: acquire(lock);
3  F12: FENCE
4  // Loads L1j arbitrarily
5  // interleaved with stores S1j
6  F13: FENCE
7  R12: release(lock);
8  F14: FENCE
9
10
11
12
13
14
```

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

## Core 2

```
1
2
3
4
5
6
7  F21: FENCE
8  A21: acquire(lock);
9  F22: FENCE
10 // Loads L2j arbitrarily
11 // interleaved with stores S2j
12 F23: FENCE
13 R22: release(lock);
14 F24: FENCE
```

# 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 | ✓ | ✓ | ✓ | ✓ | |
| $RC_{SC}$ | ✓ | ✓ | ✓ | ✓ | |
| $RC_{PC}$ | ✓ | ✓ | ✓ | ✓ | ✓ |

# Desirable Properties of a Memory Model

**Desirable properties: Programmability, Performance, and Portability**

- — Hard to satisfy all three

**Evaluating SC**

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

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

# Programming Language Memory Models

# Language Memory Models

- Data-Race-Free-0 (DRF0) model is conceptually similar to Weak Ordering (WO)
- 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 were 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?

# C++ Memory Model and Catch-Fire Semantics

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

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

**Thread 1**

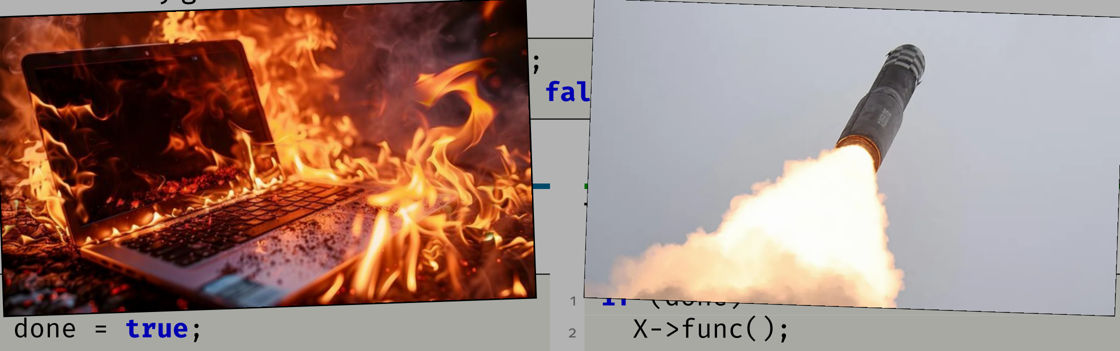
```
X = new X();
done = true;
```

**Thread 2**

```
if (done)
  X->func();
```

# C++ Memory Model and Catch-Fire Semantics

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



# Memory Operations in C++

**Synchronization** Lock, unlock, atomic load, atomic store, atomic RMW

**Data** Load, store

Compiler reordering is allowed for memory operations M1 and M2 if

- 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 Concurrent C++ Code Using Locks

```cpp
std::mutex mtx;
bool ready = false;
```

**Thread 1**

```cpp
mtx.lock();
prepareData();
ready = true;
mtx.unlock();
```

**Thread 2**

```cpp
mtx.lock();
if (ready)
  consumeData();
mtx.unlock();
```

# Using Atomics Available 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

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

**Thread 1**

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

**Thread 2**

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

# 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` in C++ and `volatile` in 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

- Producer thread creates data
- Producer thread stores to an atomic

## Consumer

- Consumer threads read from the atomic
- When the expected value is seen, data from the producer thread is visible to the consumers

> The different memory model modes indicate the strength of data sharing between threads

Memory model synchronization modes

# Memory Model Modes in C++

`memory_order_seq_cst`

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

### Thread 1

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

### Thread 2

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

Can this assert fail?

# Memory Model Modes in C++

`memory_order_seq_cst`

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

**Thread 1**

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

**Thread 2**

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

**Thread 3**

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

Can these asserts fail?

# Memory Model Modes in C++

## `memory_order_relaxed`: no happens-before edge

### Thread 1

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

### Thread 2

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

### Thread 3

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

Can these asserts fail?

# Memory Model Modes in C++

`memory_order_relaxed`: no happens-before edge

### Thread 1

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

### Thread 2

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

Can this assert fail?

# Memory Model Modes in C++

`memory_order_relaxed`: no happens-before edge

### Thread 1

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

### Thread 2

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

- 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++

`memory_order_acquire` and `memory_order_release`: introduces HB edges only between dependent variables

Thread 1

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

y is a regular data variable

Thread 2

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

Can this assert fail?

# Memory Model Modes in C++

### Thread 1

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

### Thread 2

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

### Thread 3

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

### Thread 4

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

Can these asserts pass? Can they fail?

# Memory Model Modes in C++

`memory_order_consume`: removes HB ordering on non-dependent variables

### Thread 1

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

### Thread 2

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

### Thread 3

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

Can these asserts fail?

# 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||a,

(ii) There is no intervening write c to x where b $\prec_{HB}$ c $\prec_{HB}$ a

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

**Thread 1**

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

**Thread 2**

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

```
assert (r1 != 0 || r2 != 0);
```

Can this assert fail?

# 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||a,

(ii) There is no intervening write c to x where b $\prec_{HB}$ c $\prec_{HB}$ a

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

Thread 1

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

Thread 2

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

```
assert (r1 == 0 || r2 == 0);
```

Can this assert fail?

# HBMM

```
1 x = 0;
2 y = 0;
```

**Thread 1**

```
1 r = x;
2 y = 1;
3 assert (r == 0);
4
5
```

Will the assertion pass or fail?

**Thread 2**

```
1
2
3
4 while (y == 0) {}
5 x = 1;
```

# HBMM

```
x = 0;
```

## Thread 1

```
1 x = 10;
2
```

## Thread 2

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

- Can anything go wrong with HBMM?
- What will be the behavior with `std::memory_order_relaxed` in C++?

# DRFo vs HBMM

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

**Thread 1**

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

**Thread 2**

```
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 only for data-race-free programs

# DRFo vs HBMM

## DRFo

- DRFo allows arbitrary behavior for racy executions
- DRFo is not strictly stronger than HBMM

## HBMM

- HBMM does not guarantee SC for DRF programs
- HBMM is not strictly stronger than DRFo

# HBMM

HBMM has the potential to generate out-of-thin-air (OOTA) values

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

**Thread 1**

```
x = y;
```

**Thread 2**

```
y = x;
```

Problematic for garbage-collected languages since the "out-of-thin-air" value could be an invalid pointer

- Introduces potential security loopholes

# Java Memory Model (JMM)

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

# Outcomes Possible with JMM

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

### Thread 1

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

### Thread 2

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

```
assert (r1 != 0 || r2 != 0);
```

Can these asserts fail?

# Outcomes Possible with JMM

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

**Thread 1**

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

**Thread 2**

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

```
assert (r1 == 0 || r2 == 0);
```

Can these asserts fail?

# Outcomes Possible with JMM

Racy initialization

```
obj = null;
```

**Thread 1**

```
1 obj = new Circle();
2
```

**Thread 2**

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

Can there be a NPE with JMM?

**Note**

JVMs may not exhibit all behaviors permissible under the JMM

# Outcomes Not Possible with JMM

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

Thread 1

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

Thread 2

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

```
assert (r1 != 42);
```

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

# JVMs do not comply with the JMM!

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

**Thread 1**

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

**Thread 2**

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

Can this assert fail under HBMM and JMM?

```
assert (r2 == 0);
```

# JVMs do not comply with the JMM!

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

**Thread 1**

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

**Thread 2**

```
r2 = y;
if (r2 ...
}
```

- 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

```
assert (r2 == 0);
```

# Lessons Learned

**Specifying semantics for racy programs is hard**

Simple optimizations may introduce unintended consequences

**SC for DRF is now the preferred baseline**

- Make sure your program is free of data races
- Compiler and architecture setup will guarantee SC execution

# References

V. Nagarajan et al. A Primer on Memory Consistency and Cache Coherence. Chapters 1–5, $2^{nd}$ edition, Springer Cham.

S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. Journal of Computer, vol. 29, no. 12, pp. 66–76, Dec. 1996.

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.