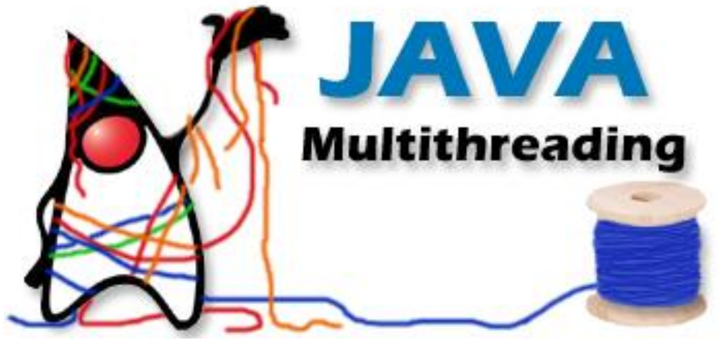
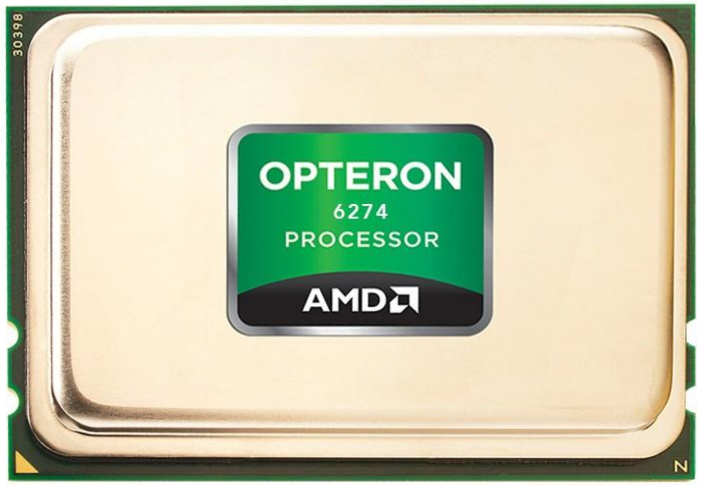
Memory Consistency Model

Swarnendu Biswas
UT Austin

Outline

- Data races
- Memory consistency models
- Sequential Consistency
- Hardware memory models
    - TSO, PSO, Relaxed consistency
- Language memory models
    - C++, Java





Today’s Trends

Data Race: Primary Source of Concurrency Errors

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

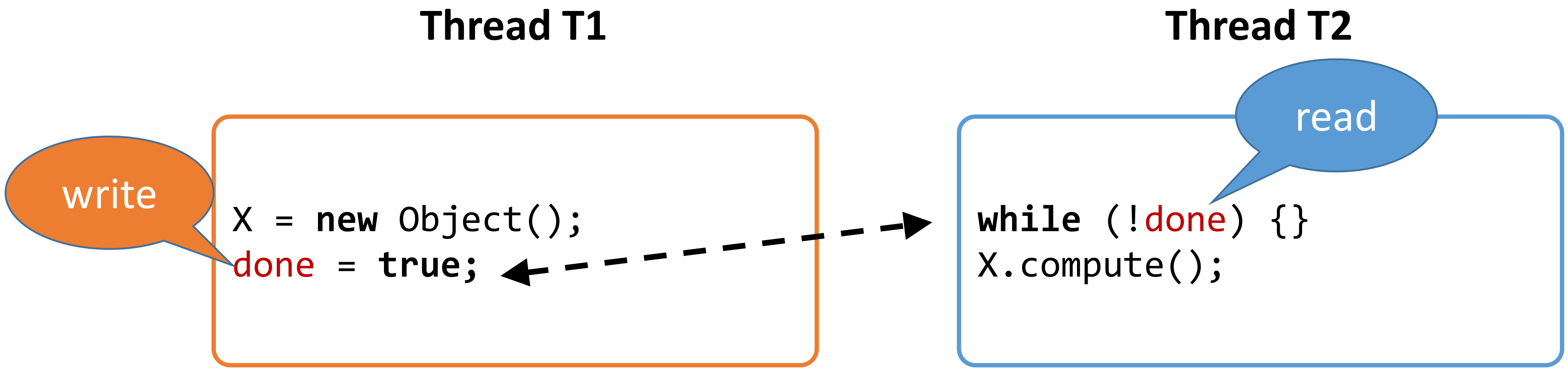
Thread T1

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

Thread T2

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

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



Thread T1

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

Thread T2

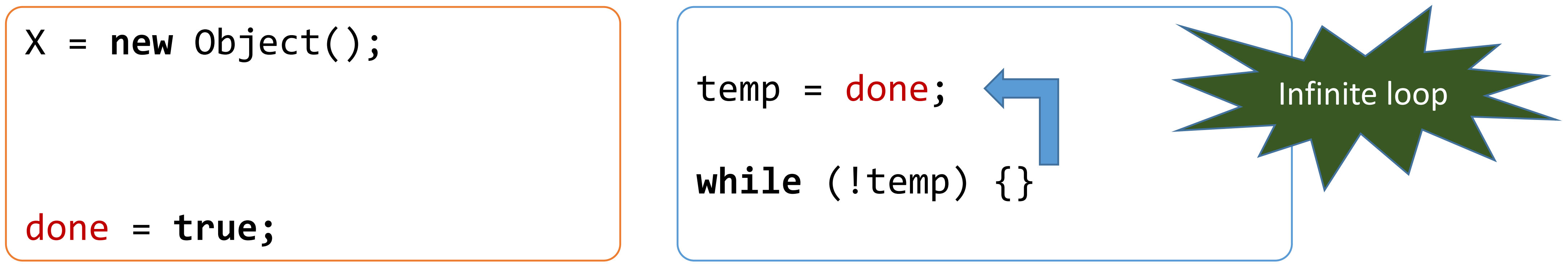
```
while (!done) {}
X.compute();
```

Data race

Conflicting accesses – two threads access the same shared variable where at least one access is a write

Concurrent accesses – accesses are not ordered by synchronization operations

Thread T1 **Thread T2**



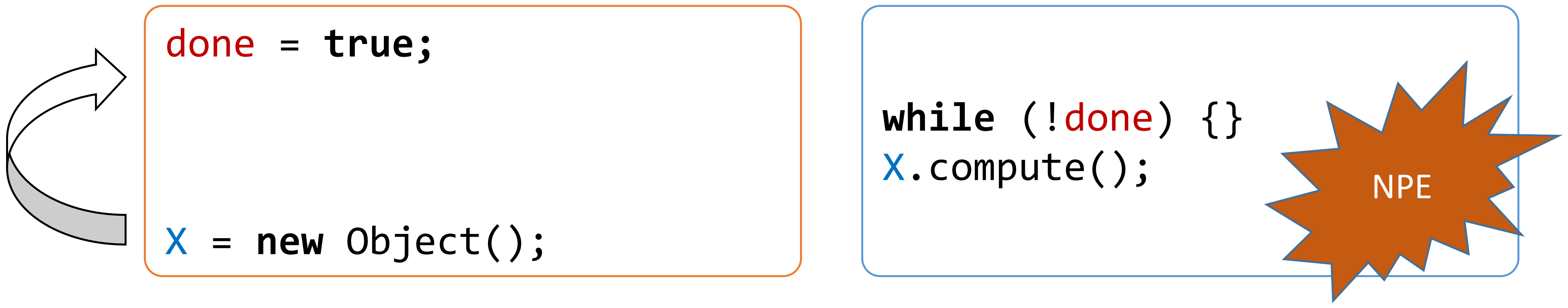
```
X = new Object();

done = true;
```

```
temp = done;

while (!temp) {}
```

Thread T1 **Thread T2**



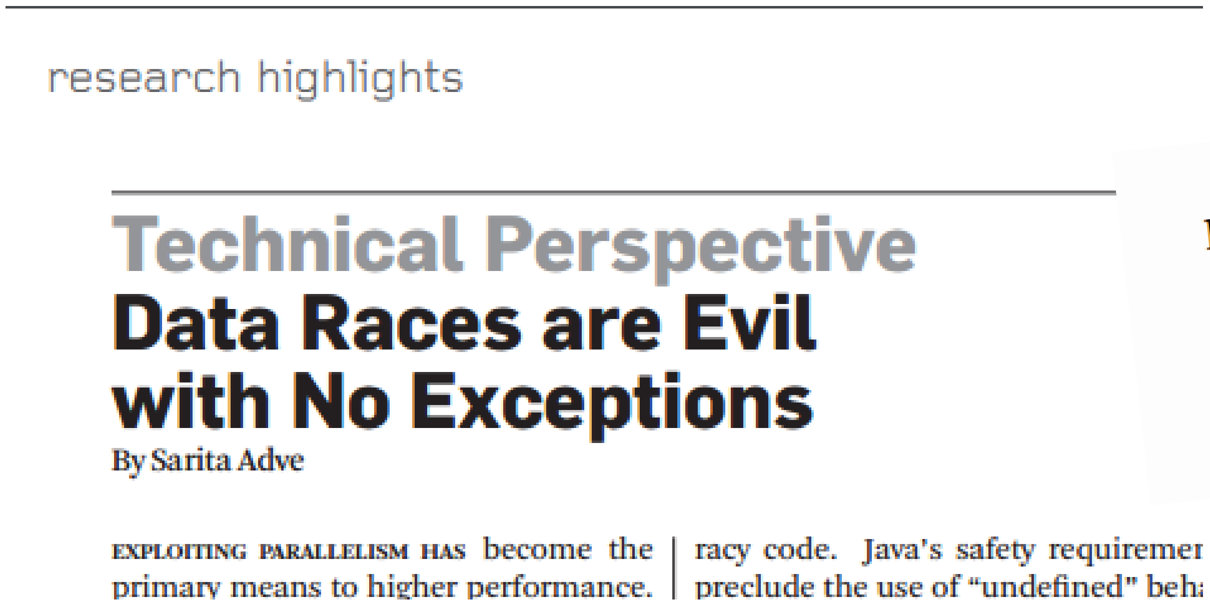
```
done = true;

X = new Object();
```

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

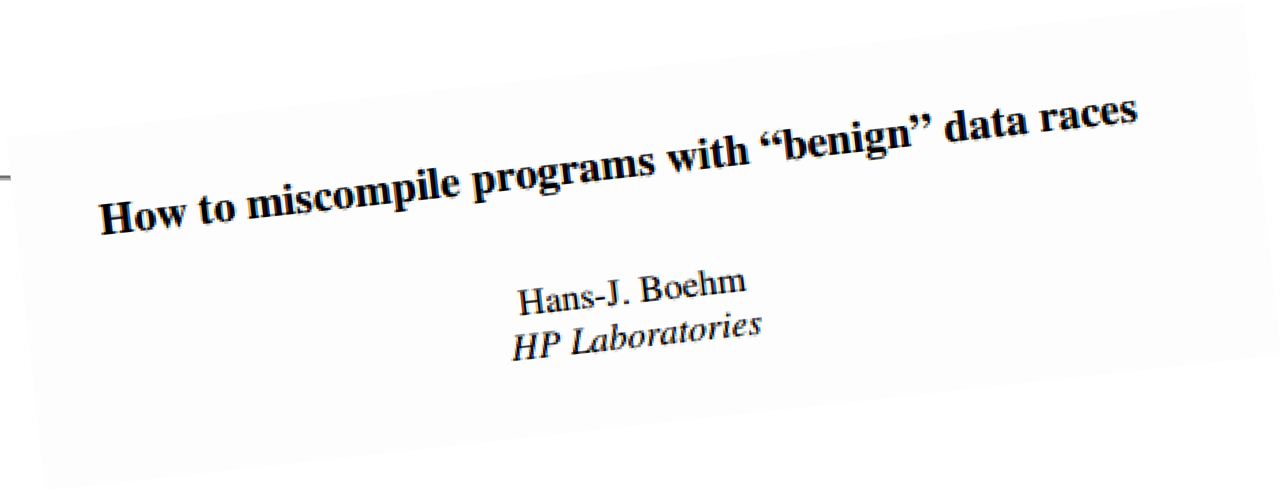
Data Races are Bad

Therac-25 accident & Northeast US Blackout & NASDAQ Facebook glitch


research highlights

Technical Perspective
Data Races are Evil with No Exceptions
By Sarita Adve

EXPLOITING PARALLELISM HAS become the primary means to higher performance. | racy code. Java's safety requiremer preclude the use of "undefined" beh


How to miscompile programs with "benign" data races

Hans-J. Boehm
HP Laboratories

Memory Consistency Model: What Value Can a Read Return?

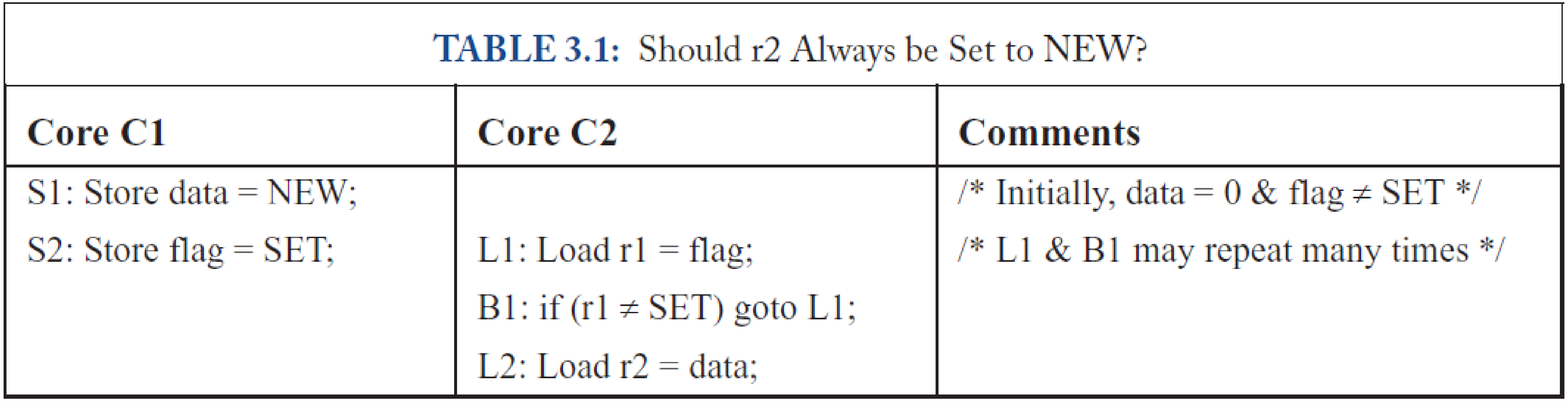


TABLE 3.1: Should r2 Always be Set to NEW?

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

How a Core Might Reorder Accesses?

- Store-store
- Load-load
- Store-load
- Load-store

Memory Consistency Model

- Specifies the allowed behaviors of multithreaded programs executing with shared memory
  - Both at the hardware-level and at the programming-language-level
- “What values can a load return?”
  - Return the “last” write
  - Uniprocessor: program order
  - Multiprocessor: ?
- There can be multiple correct behaviors

Memory Consistency Model

- 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?

Dekker’s Algorithm

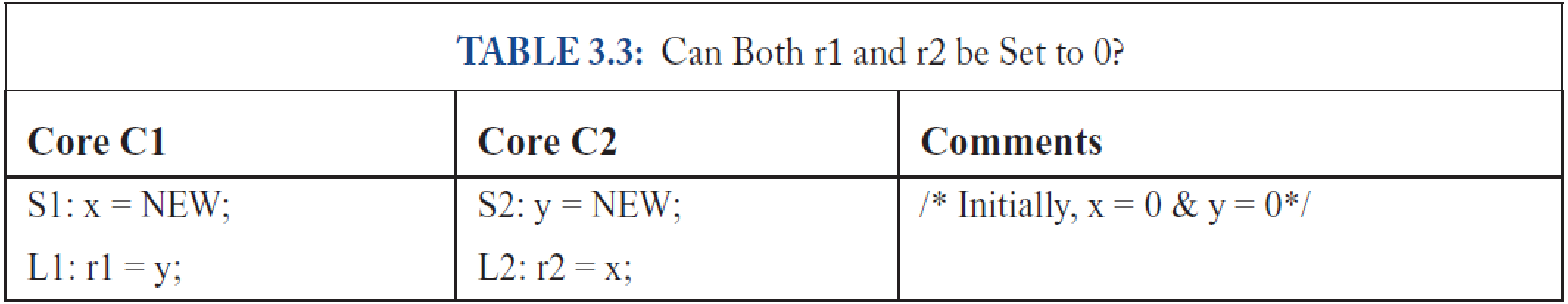


TABLE 3.3: Can Both r1 and r2 be Set to 0?

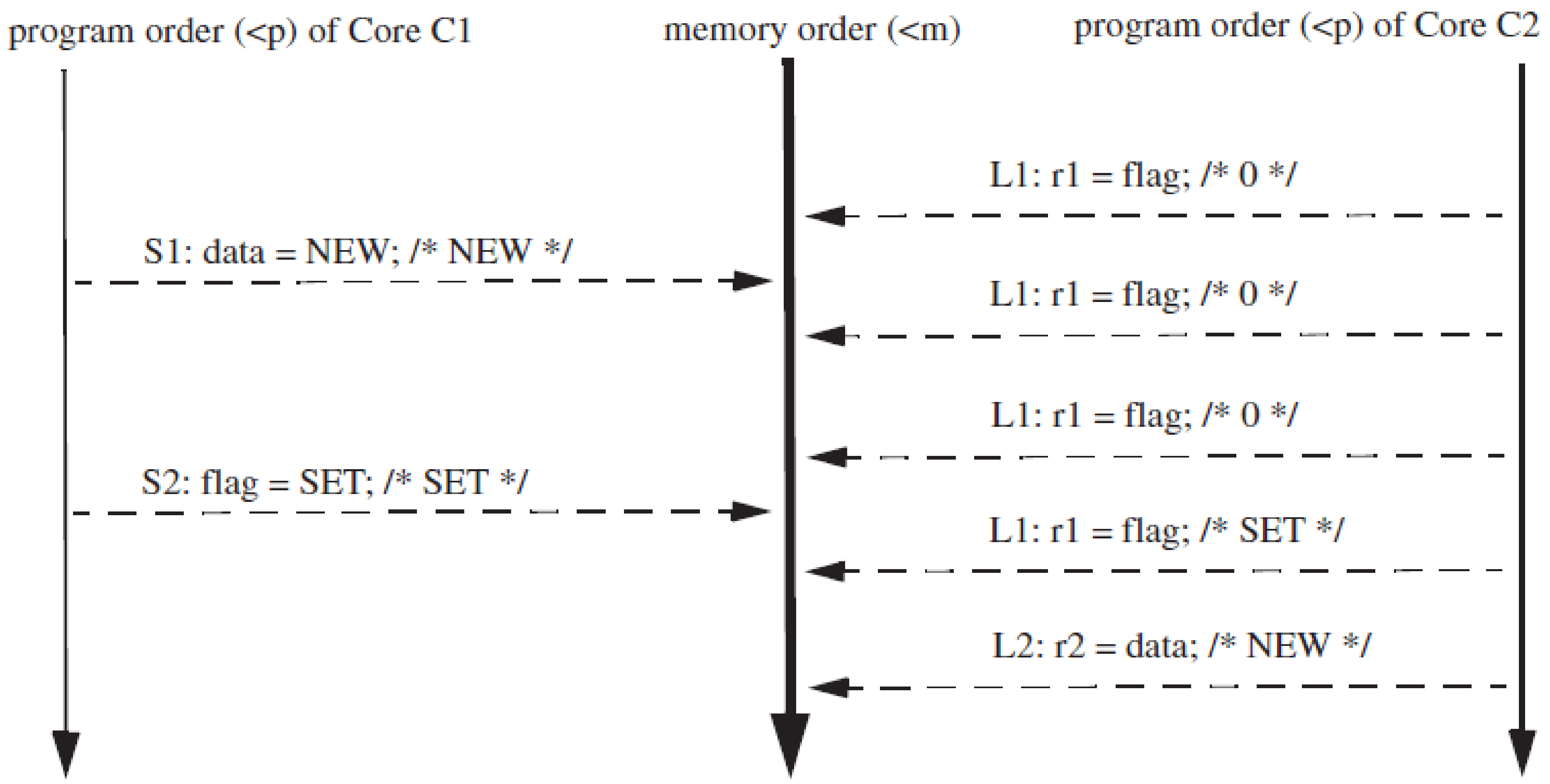
Core C1	Core C2	Comments

S1: x = NEW;
L1: r1 = y;
	S2: y = NEW;
	L2: r2 = x;
		/* Initially, x = 0 & y = 0*/

Sequential Consistency (SC)

- Uniprocessor - operations executed in order specified by the program
- Multiprocessor - all operations executed in order, and the operations of each individual core appear in program order

Earlier Example Under SC

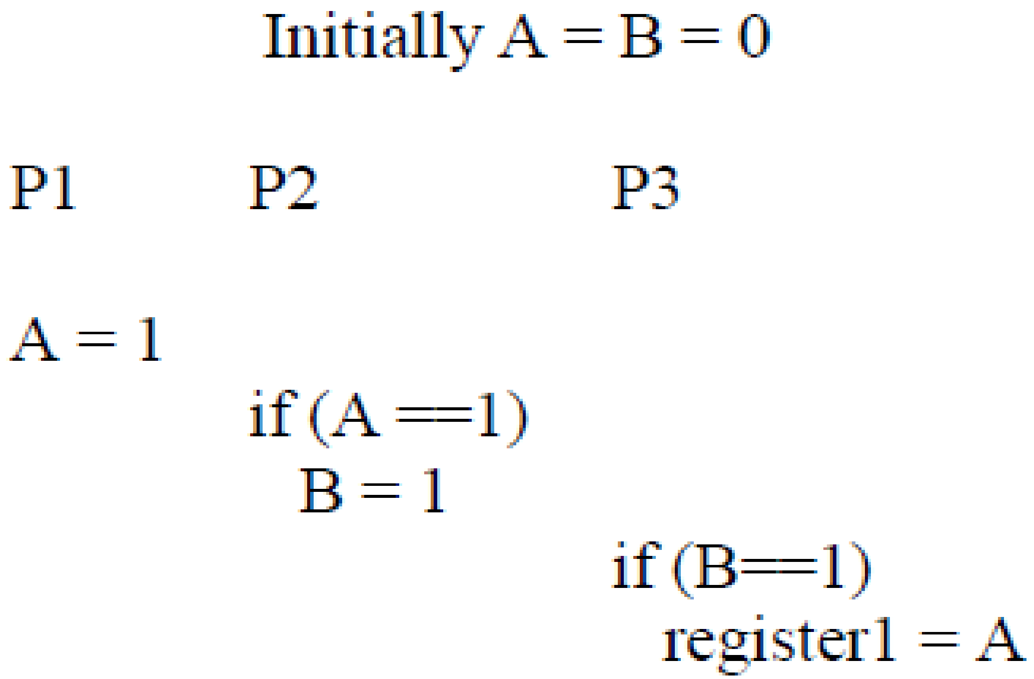


SC Rules

- **a = b or a != b**
  - if L(a) <p L(b) ⇒ L(a) <m L(b)
  - If L(a) <p S(b) ⇒ L(a) <m S(b)
  - If S(a) <p S(b) ⇒ S(a) <m S(b)
  - If S(a) <p L(b) ⇒ S(a) <m L(b)

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

SC Provides Write Atomicity



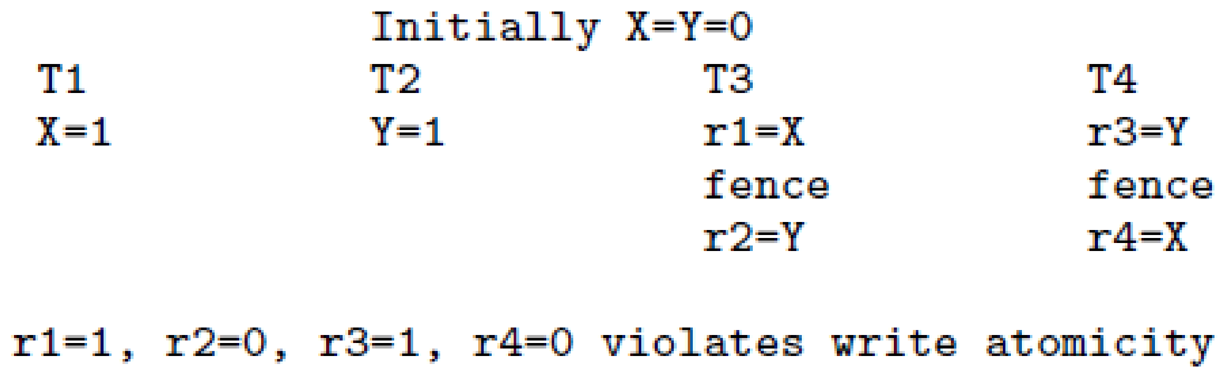
```
            Initially A = B = 0

P1          P2                  P3

A = 1
            if (A ==1)
               B = 1
                                if (B==1)
                                   register1 = A
```

Write Atomicity

- Relaxing write atomicity violates SC



```
                  Initially X=Y=0
 T1               T2              T3                T4
 X=1              Y=1             r1=X              r3=Y
                                  fence             fence
                                  r2=Y              r4=X

r1=1, r2=0, r3=1, r4=0 violates write atomicity
```

End-to-end SC

- Simple memory model that can be implemented both in hardware and in languages
- Performance
  - Naive hardware
    - Maintain program order - expensive for a write
      - E.g., write buffer can break Dekker's algorithm
    - Write atomicity
- Program semantics
  - SC does not guarantee data race freedom
  - Not a strong memory model

```
a++;
```

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

Cache Coherence

- Single writer multiple readers
- Memory updates are passed correctly, cached copies always contain the most recent data
- Virtually a synonym for SC

- 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*

Maintaining the Illusion of Write Atomicity

Initially A = B = C = 0

```
P1          P2       P3                          P4
A = 1;      A = 2;        while (B != 1) {;}              while (B != 1) {;}
B = 1;      C = 1;        while (C != 1) {;}            while (C != 1) {;}
                          tmp1 = A;   ~~1~~                 tmp2 = A;   ~~2~~
```

Memory Consistency vs Cache Coherence

- Cache Coherence does not define shared memory behavior
  - Goal is to make caches invisible
- Memory consistency can use cache coherence as a “black box”

Characterizing Hardware Memory Models

- Relax program order
  - Store → Load, Store → Store, etc.
  - Applicable to pairs of operations with different addresses
- Relax write atomicity
  - Read own write early
  - Read other’s write early
    - Applicable to only cache-based systems

Read Other’s Write Early Can Violate Write Atomicity

Initially A = B = 0

```
P1                    P2                    P3
A = 1                 while (A != 1) ;while (B != 1) ;
                      B = 1;                   tmp = A
```

P1	P2	P3
Write, A, 1		
	Read, A, 1	
	Write, B, 1	
		Read, B, 1
		Read, A, ~~0~~

Possible Interleavings Under SC

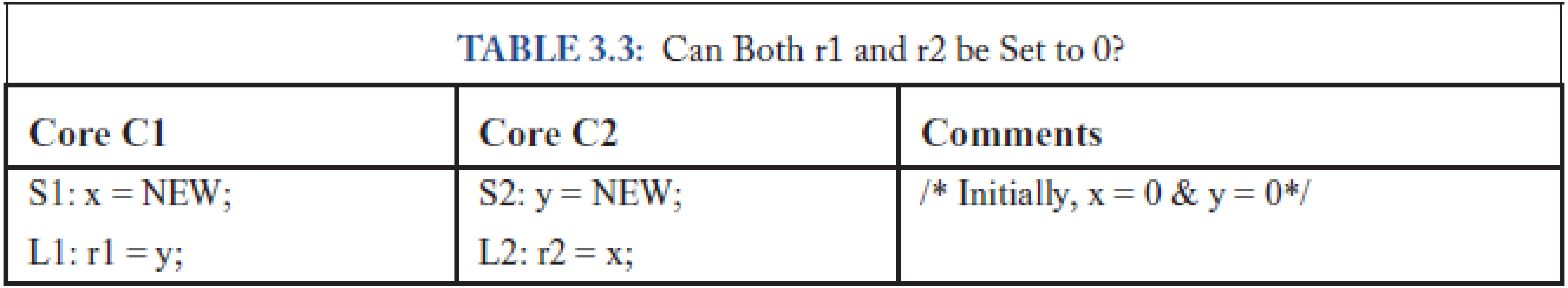


TABLE 3.3: Can Both r1 and r2 be Set to 0?

Core C1	Core C2	Comments

S1: x = NEW;
L1: r1 = y;
	S2: y = NEW;
	L2: r2 = x;
		/* Initially, x = 0 & y = 0*/

Total Store Order (TSO)

- Allows reordering stores to loads
- Can read own write early, not other’s writes

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

TSO Rules

- **a == b or a != b**
  - If L(a) <p L(b) ⇒ L(a) <m L(b)
  - If L(a) <p S(b) ⇒ L(a) <m S(b)
  - If S(a) <p S(b) ⇒ S(a) <m S(b)
  - ~~If S(a) <p L(b) ⇒ S(a) <m L(b)~~ /* Enables FIFO Write Buffer */
- Every load gets its value from the last store before it to the same address
- Needs a notion of a FENCE

TSO Rules (…contd)

- If L(a) <p FENCE ⇒ L(a) <m FENCE
- If S(a) <p FENCE ⇒ S(a) <m FENCE
- If FENCE <p FENCE ⇒ FENCE <m FENCE
- If FENCE <p L(a) ⇒ FENCE <m L(a)
- If FENCE <p S(a) ⇒ FENCE <m S(a)

- If S(a) <p FENCE ⇒ S(a) <m FENCE
- If FENCE <p L(a) ⇒ FENCE <m L(a)

RMW in TSO

- Load of a RMW cannot be performed until earlier stores are performed (i.e., exited 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

Partial Store 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

Opportunities to Reorder Memory Operations

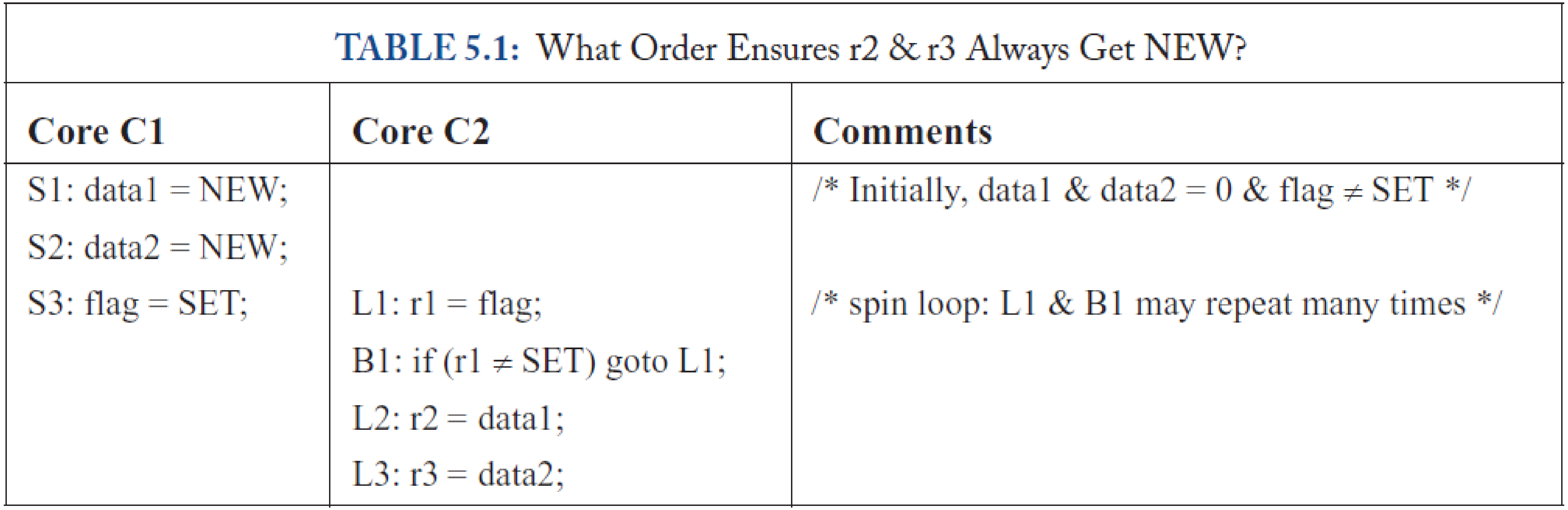


TABLE 5.1: What Order Ensures r2 & r3 Always Get NEW?

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

Reorder Operations Within a Synchronization Block

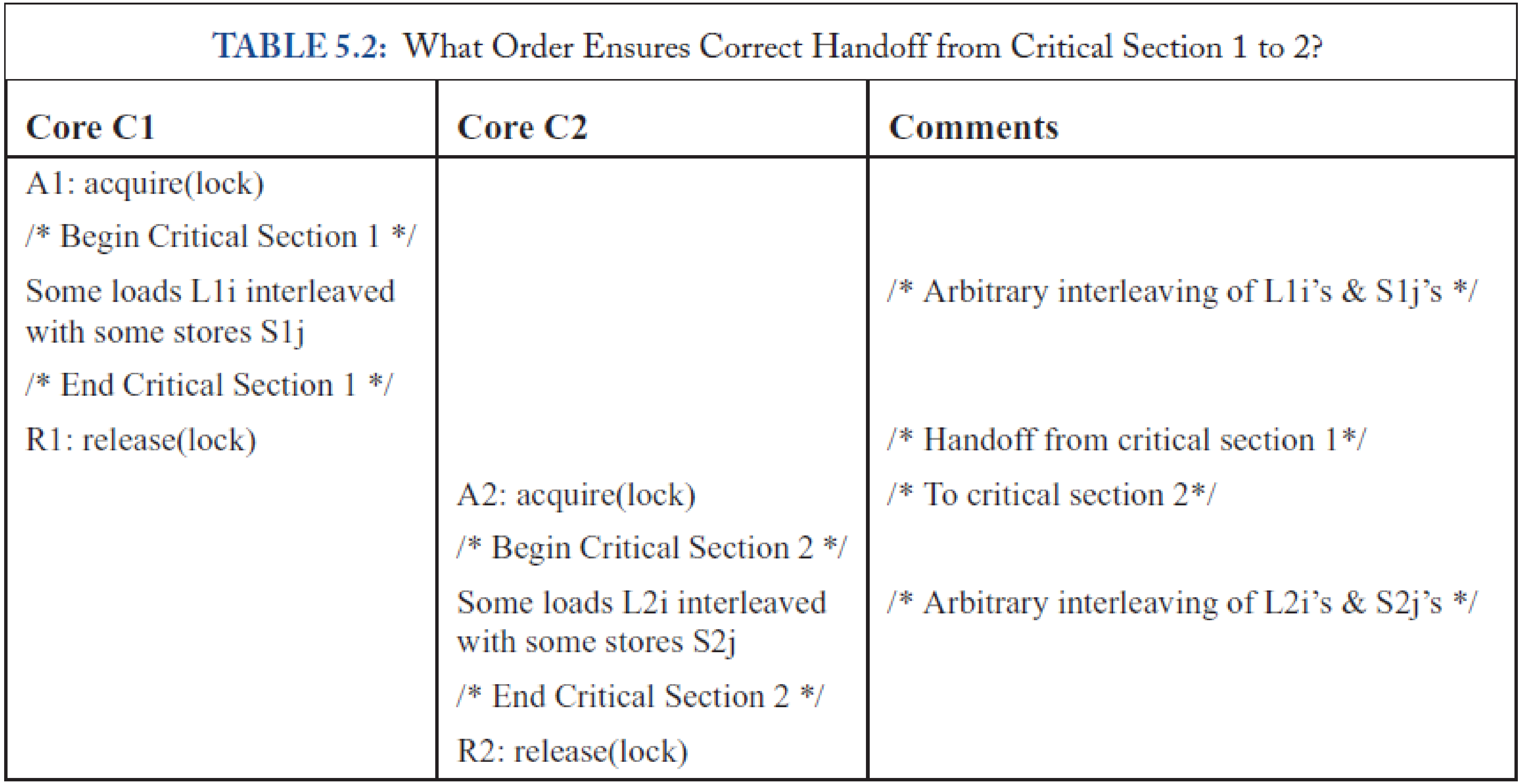


TABLE 5.2: What Order Ensures Correct Handoff from Critical Section 1 to 2?

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

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 FENCE ⇒ L(a) <m FENCE
- If S(a) <p FENCE ⇒ S(a) <m FENCE
- If FENCE <p FENCE ⇒ FENCE <m FENCE
- If FENCE <p L(a) ⇒ FENCE <m L(a)
- If FENCE <p S(a) ⇒ FENCE <m S(a)

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

- If L(a) <p L'(a) ⇒ L(a) <m L'(a)
- If L(a) <p S(a) ⇒ L(a) <m S(a)
- If S(a) <p S'(a) ⇒ 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

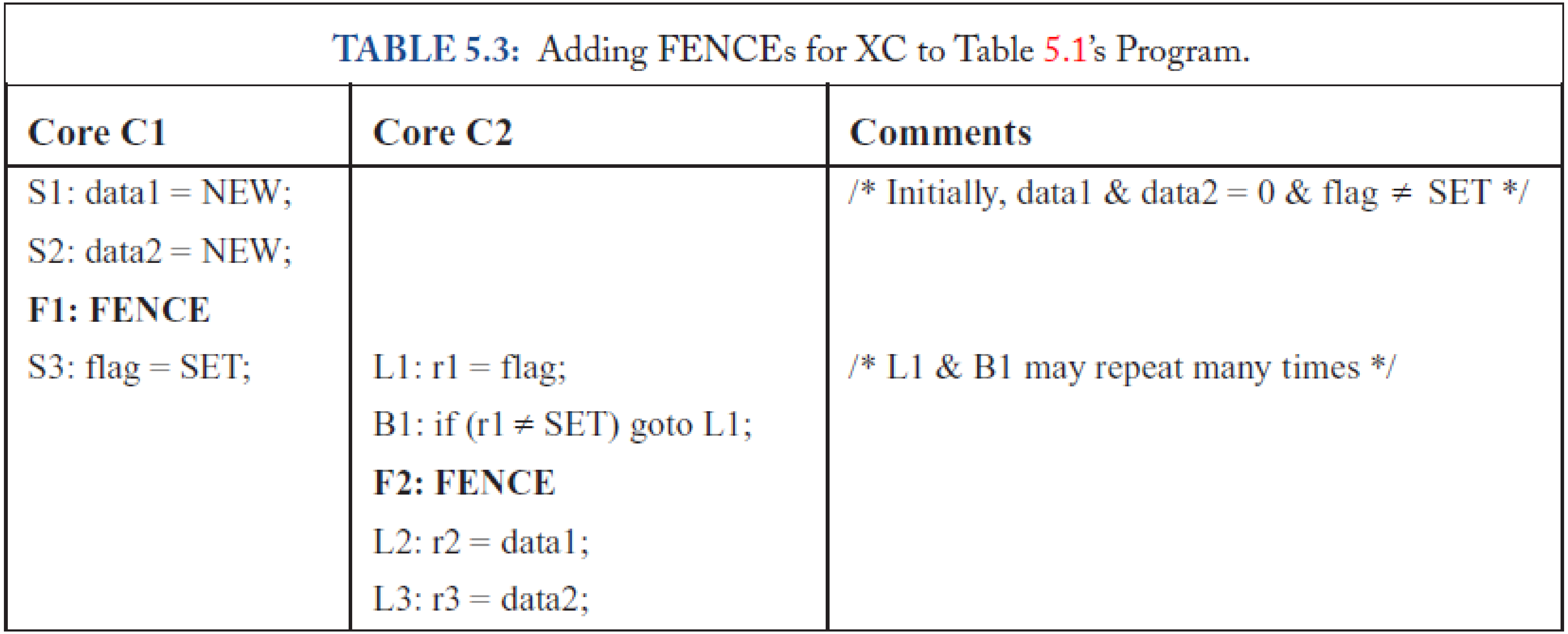


TABLE 5.3: Adding FENCEs for XC to Table 5.1's Program.

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

Correct Implementation Under Relaxed Consistency

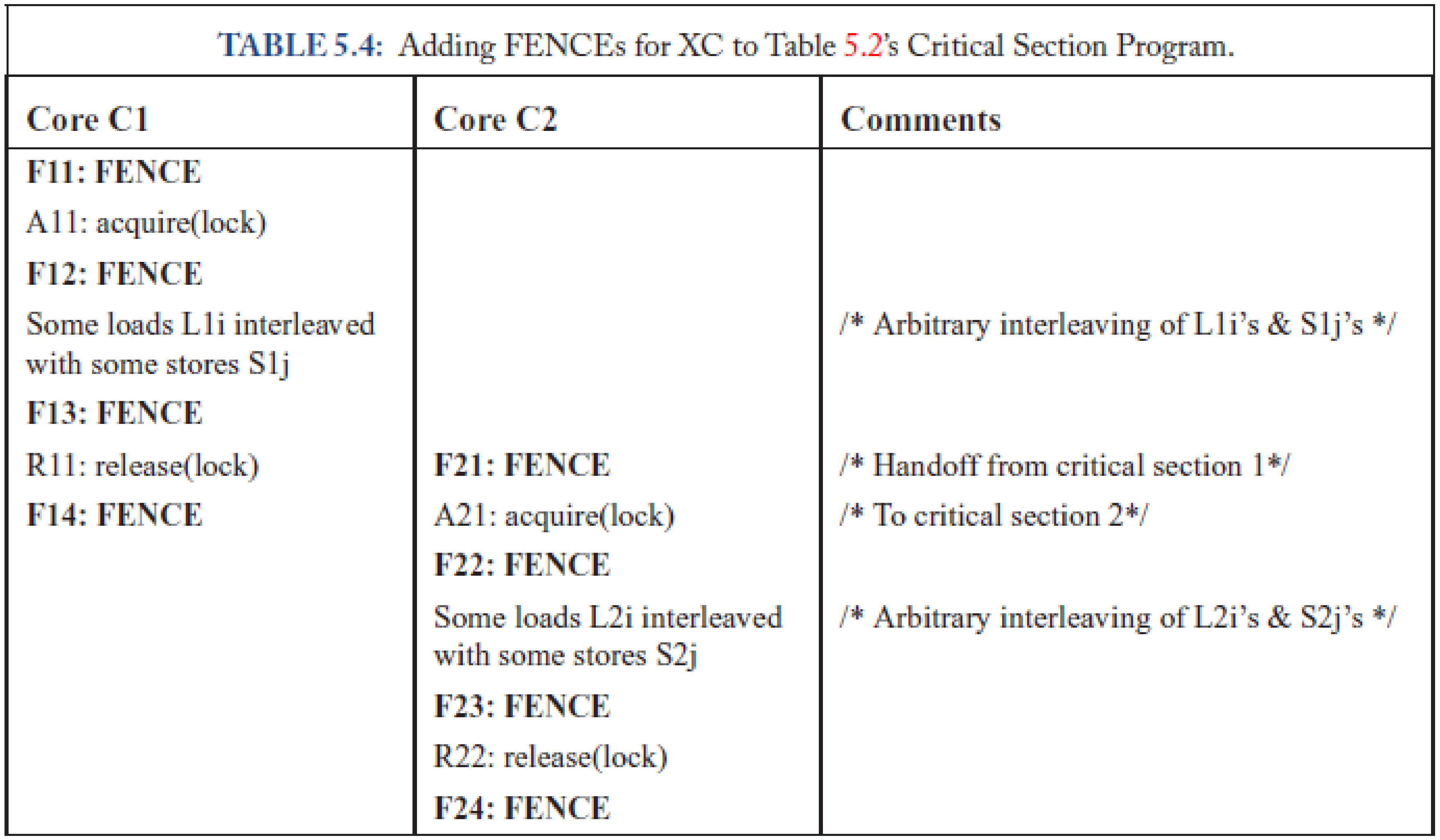


TABLE 5.4: Adding FENCEs for XC to Table 5.2's Critical Section Program.

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

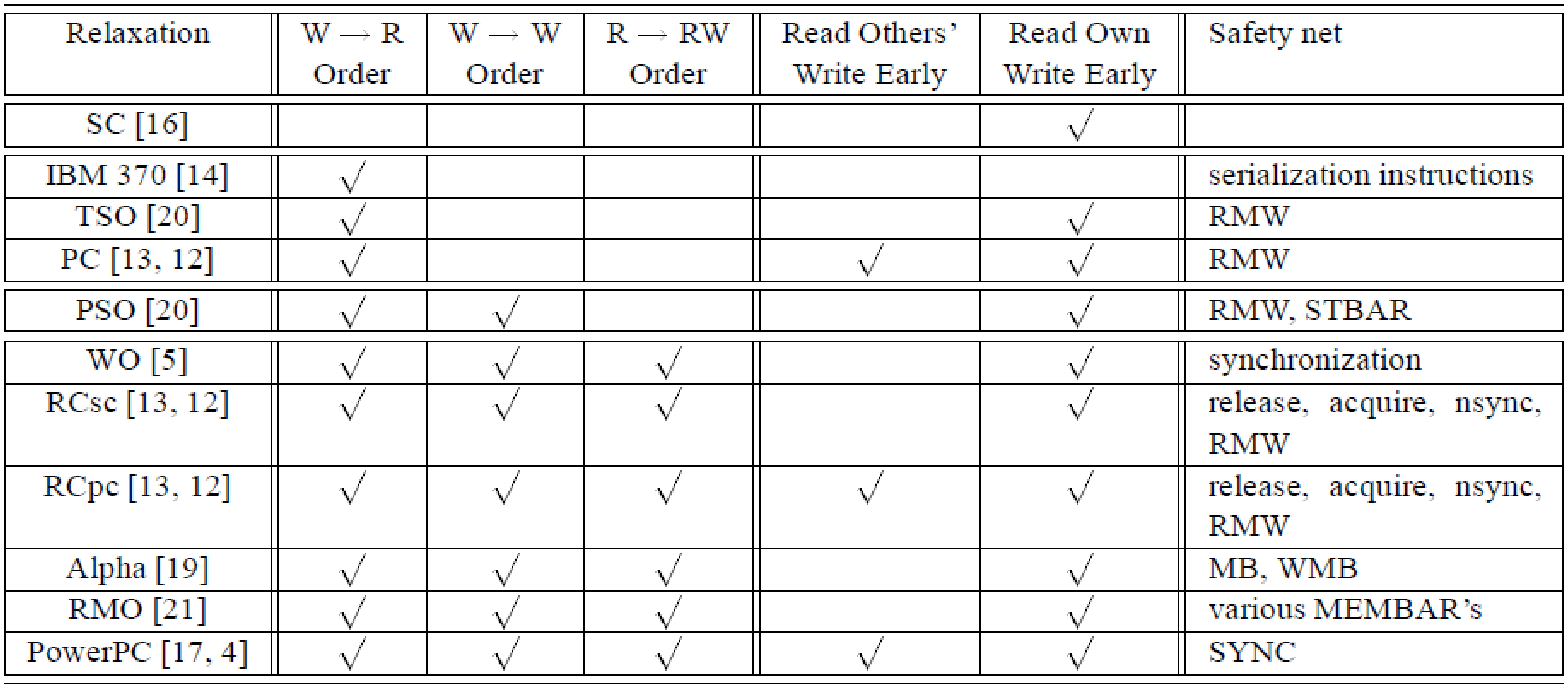
Relaxed Consistency Memory Models

- 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
  - Distinguishes between acquire and release synchronization operations
  - RCsc - maintains SC between synchronization operations
  - Acquire → all, all → release, and sync → sync

Relaxed Consistency Memory Models

- Why should we use them?
  - Performance
- Why should we not use them?
  - Complexity

Hardware Memory Models: One Slide Summary



Relaxation	W → R Order	W → W Order	R → RW Order	Read Others' Write Early	Read Own Write Early	Safety net
SC [16]					✓	
IBM 370 [14]	✓					serialization instructions
TSO [20]	✓				✓	RMW
PC [13, 12]	✓			✓	✓	RMW
PSO [20]	✓	✓			✓	RMW, STBAR
WO [5]	✓	✓	✓		✓	synchronization
RCsc [13, 12]	✓	✓	✓		✓	release, acquire, nsync, RMW
RCpc [13, 12]	✓	✓	✓	✓	✓	release, acquire, nsync, RMW
Alpha [19]	✓	✓	✓		✓	MB, WMB
RMO [21]	✓	✓	✓		✓	various MEMBAR's
PowerPC [17, 4]	✓	✓	✓	✓	✓	SYNC

DRF0 Model

- Conceptually similar to WO
- Assumes no data races
- Allows many optimizations in the compiler and hardware

Language Memory Models

- Developed much later
- Most are based on the data-race-free-0 (DRF0) model

Why do we need one?

- Isn’t the hardware memory model enough?

C++ Memory Model

- Adaptation of the DRFO memory model
    - SC for data race free programs
- C/C++ simply ignore data races
    - No safety guarantees in the language
- Memory operation
    - Synchronization: lock, unlock, atomic load, atomic store, atomic RMW
    - Data: Load, Store

C++ Memory Model

- Compiler reordering **allowed** for memory operations M1 and M2 when:
  - 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

Write Correct C++ Code

- Mutually exclusive execution 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;
```

```
prepareData();

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

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

Using Atomics

- "Data race free" variable by definition: `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, not atomic (can be part of a data race)

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

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

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

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
};
```

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?

Relaxed Ordering

```
// Thread 1:
r1 = y.load(memory_order_relaxed);
x.store(r1, memory_order_relaxed);
```

```
// Thread 2:
r2 = x.load(memory_order_relaxed); // C
y.store(42, memory_order_relaxed); // D
```

Is r1 == r2 == 42 possible?

Relaxed Ordering

```
// Thread 1:
r1 = x.load(memory_order_relaxed);
If (r1 == 42) {
  y.store(r1, memory_order_relaxed);
}
```

```
// Thread 2:
r2 = y.load(memory_order_relaxed);
If (r2 == 42) {
  x.store(42, memory_order_relaxed);
}
```

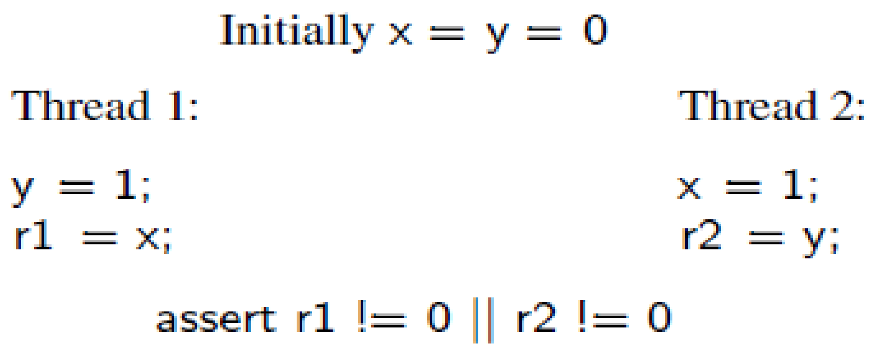
Is r1 == r2 == 42 possible?

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
- Atomic (C++)/ 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

Java Memory Model (JMM)

- First high-level language to incorporate a memory model
- Provides memory- and type-safety, so has to define some semantics for data races



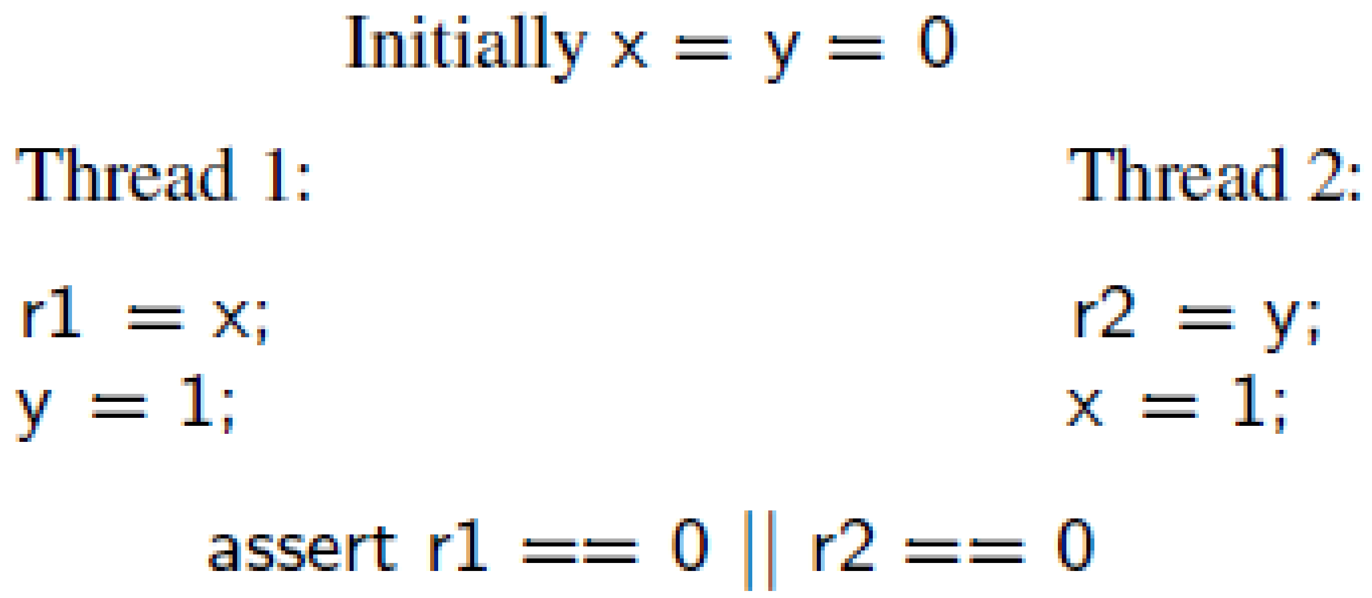
```
           Initially x = y = 0

Thread 1:                         Thread 2:

y  = 1;                           x  = 1;
r1  = x;                          r2  = y;

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

JMM (…contd)



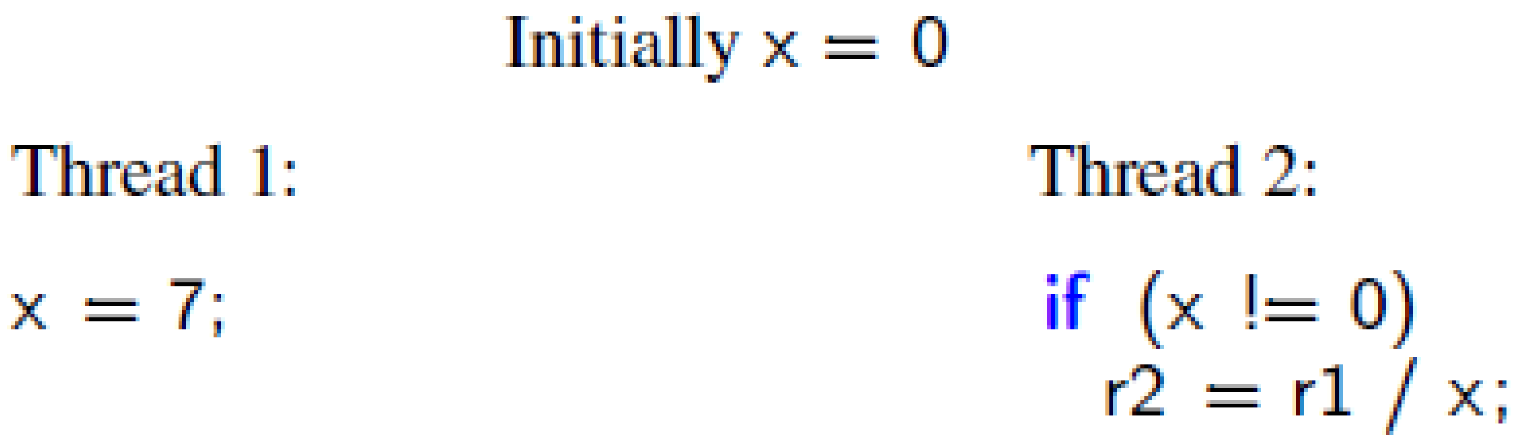
```
Initially x = y = 0

Thread 1:                    Thread 2:

r1 = x;                      r2 = y;
y = 1;                       x = 1;

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

Happens-before Memory Model (HBMM)



Initially x = 0

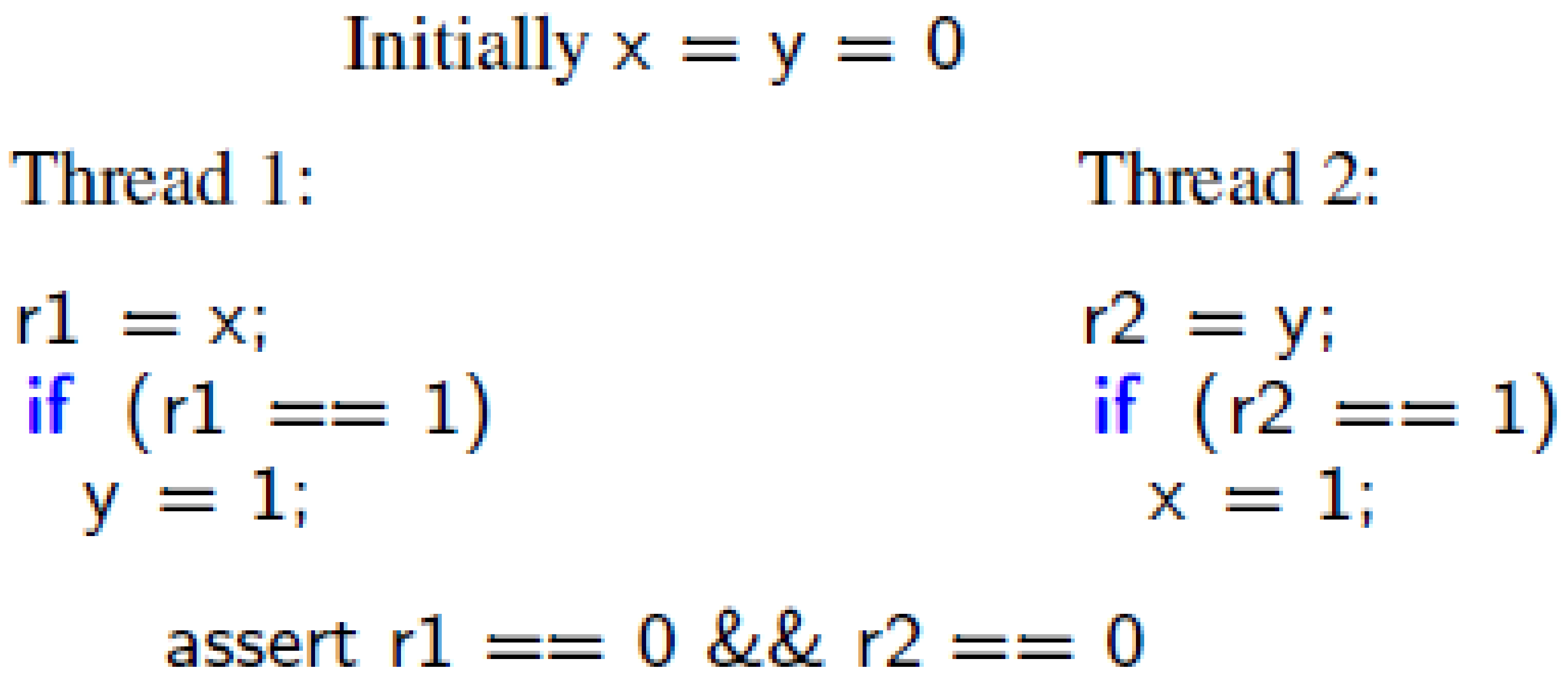
Thread 1:

```
x = 7;
```

Thread 2:

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

HBMM (...contd)



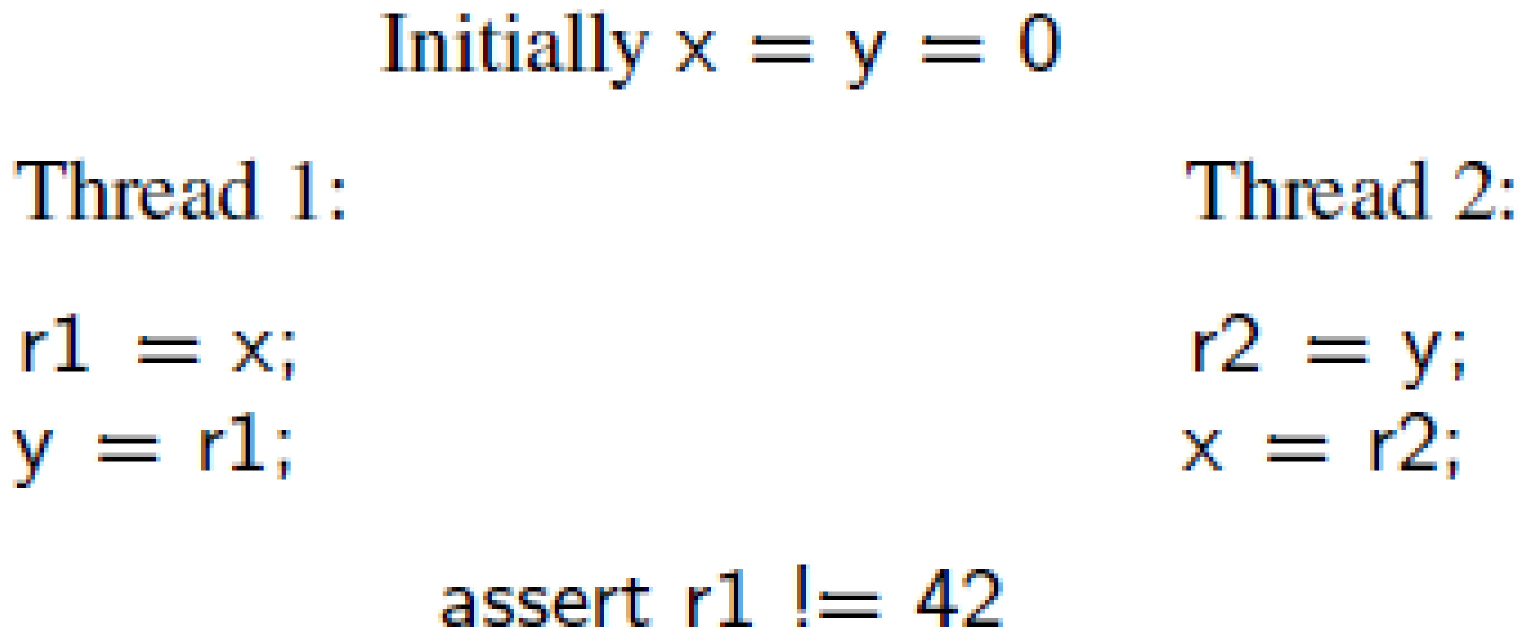
Initially x = y = 0

```
Thread 1:                 Thread 2:

r1 = x;                   r2 = y;
if (r1 == 1)              if (r2 == 1)
  y = 1;                    x = 1;

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

JMM is Stronger than DRF0 and HBMM



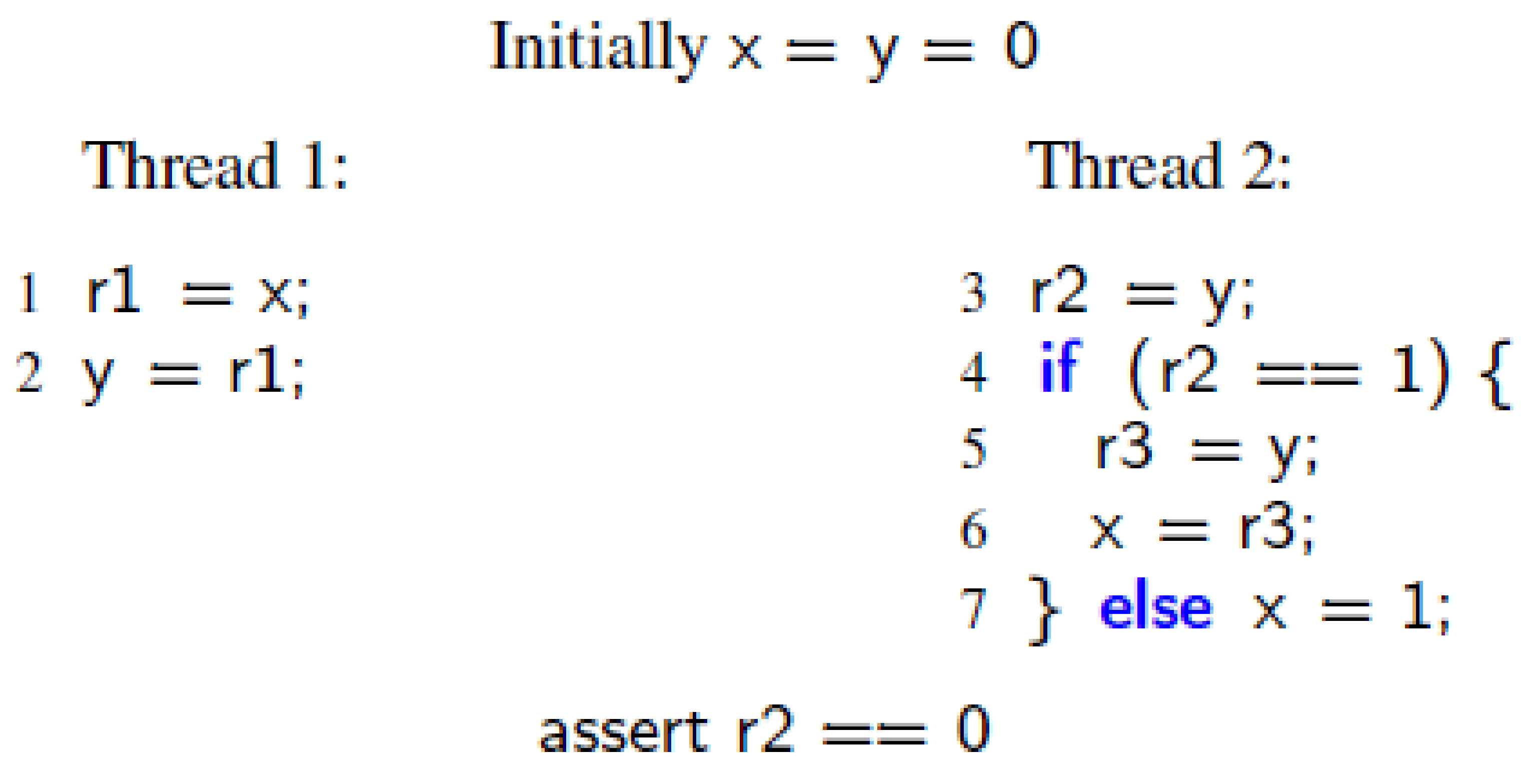
Initially x = y = 0

```
Thread 1:                      Thread 2:

r1 = x;                        r2 = y;
y = r1;                        x = r2;
```

assert r1 != 42

JVMs Do Not Comply with the JMM



Initially x = y = 0

```
Thread 1:              Thread 2:

r1 = x;                r2 = y;
y = r1;                if (r2 == 1) {
                         r3 = y;
                         x = r3;
                       } else x = 1;
```

assert r2 == 0

What Constitutes a Good Memory Model?

- Programmability
- Performance
- Portability
- Precision

Lessons Learnt

- SC for DRF is the minimal baseline
  - Make sure the program is free of data races
  - System guarantees SC execution
- Specifying semantics for racy programs is hard
- Simple optimizations may introduce unintended consequences

Memory Consistency Model

Swarnendu Biswas
UT Austin