CS 610: OpenMP Memory Model

Swarnendu Biswas

Semester 2022-2023-I

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 Tutorials. 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();
```

Thread T1

```
X = new Object();

done = true;
```

Thread T2

```
temp = done;

while (!temp) {}
```

Infinite loop

Thread T1

```
done = true;

X = new Object();
```

Thread T2

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

NPE

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

Reordering of Accesses by Hardware

Different addresses!

Store-store

Correct in a single-threaded context

Non-trivial in a multithreaded context

Store-load

What values can a load return?

- Return the “last” write
- Uniprocessor: program order
- 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?

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**.

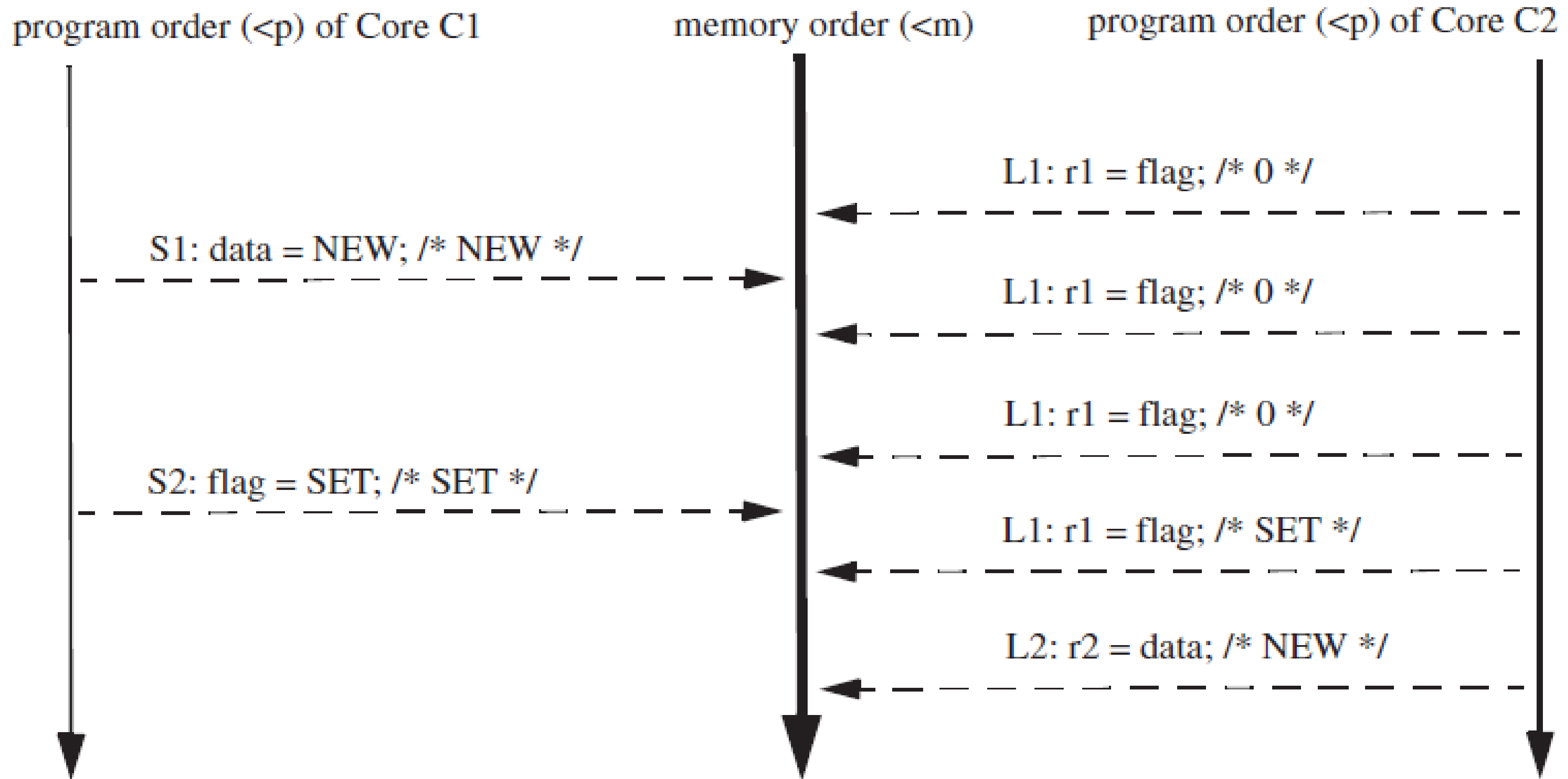
Leslie Lamport

Interleavings with SC

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;	

Interleavings with SC



SC Formalism

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

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

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

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	Cache Coherence
• **Defines** shared memory behavior	• **Does not define** shared memory behavior
• Related to **all** shared-memory locations	• Specific to a **single** shared-memory location
• Policy on **when** new value is propagated to other cores	• **Propagate** new value to other cached copies • Invalidation-based or update-based
• Memory consistency implementations can use cache coherence as a "black box"	

Existing Memory Consistency Models

Hardware	Programming Languages
• Sequential Consistency (SC)	• Java
• Total Store Order (TSO)	• C++ and OpenMP
• Partial Store Order (PSO)	• …
• Weak Ordering (WO)	
• …	

Total Store Order

- 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 $L(a) <_p L(b) \Rightarrow L(a) <_m L(b)$
- 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 $L(a) <_p FENCE \Rightarrow L(a) <_m FENCE$

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

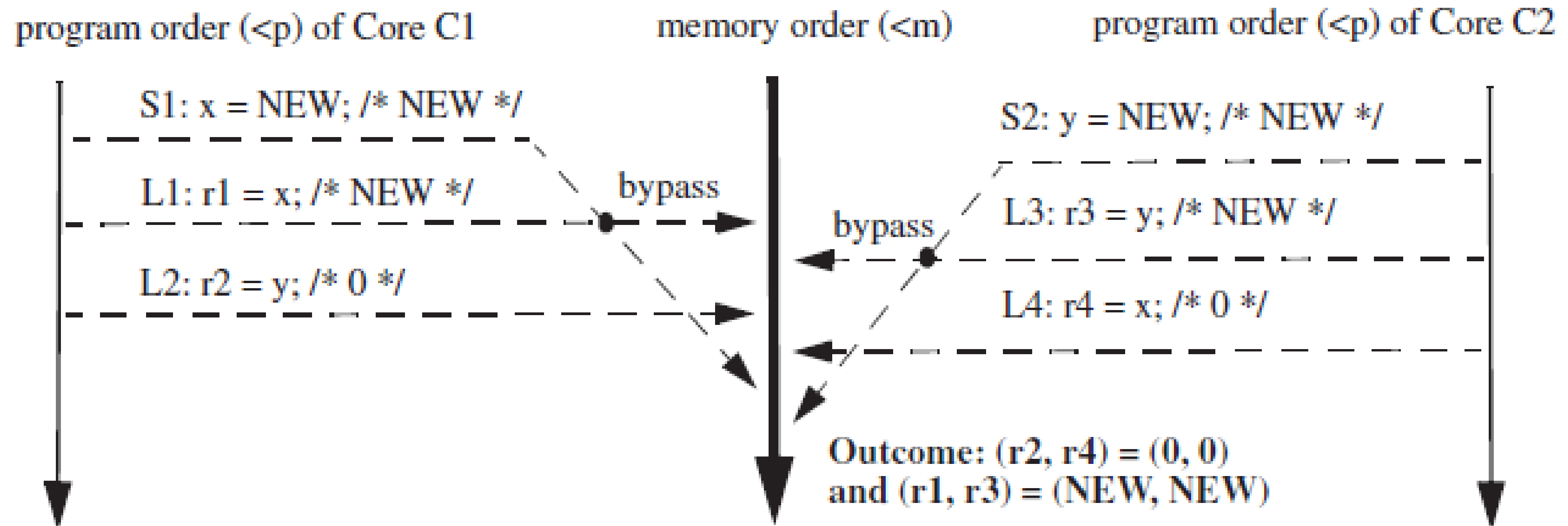
If $FENCE <_p L(a) \Rightarrow FENCE <_m L(a)$

Possible Outcomes with TSO

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

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

Possible Outcomes with TSO



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

Desirable Properties of a Memory Model

Hard to satisfy all three properties

- Programmability
- Performance
- Portability

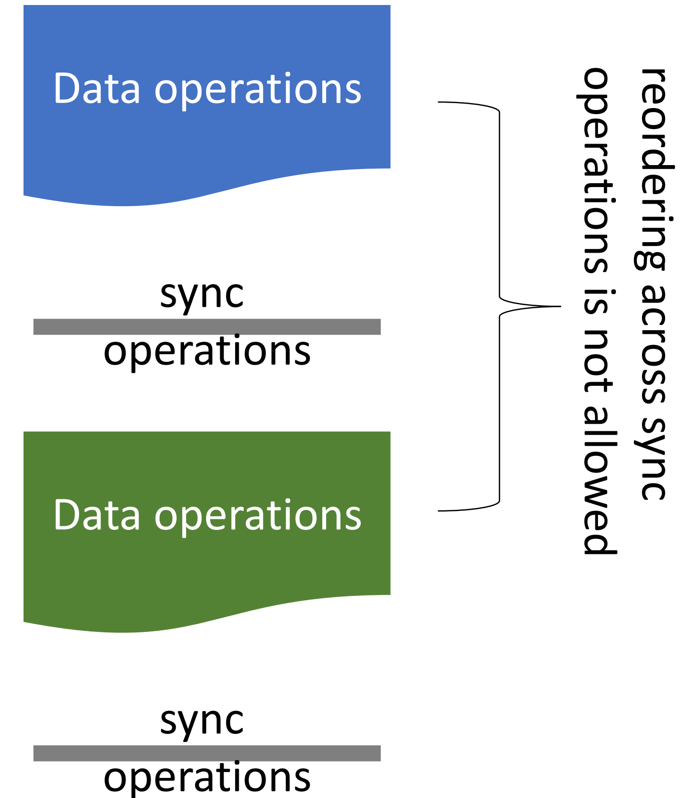
Think of SC

Pros
- Intuitive
- Serializability of instructions

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

Relaxed Consistency Memory Model in OpenMP

- OpenMP supports a relaxed consistency shared memory model
  - Closely related to the weak ordering model
- Threads can maintain a temporary view of shared memory that is not consistent with other threads
- These temporary views are made consistent only at certain points in the program
- The operation that enforces consistency is called the `flush` operation



Semantics of the `flush` Operation

- A `flush` is a sequence point at which a thread is guaranteed to see a consistent view of memory
  - All previous read/writes by this thread have completed and are visible to other threads
  - No subsequent read/writes by this thread have occurred
- A `flush` operation is analogous to a fence in other shared memory APIs

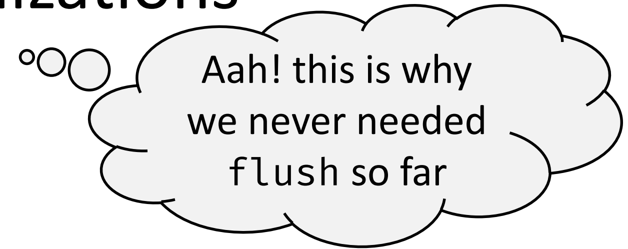
Potential Benefits with Relaxed Consistency

- Relaxed memory model allows flexibility to OpenMP implementations
- Write to A
  - May complete immediately
  - May complete after the execution marked “…; …”

```
A = 1


…

…


#pragma omp flush(A)
```

Flush and Synchronization

- A flush operation is implied by OpenMP synchronizations
  - at entry/exit of parallel, critical, and ordered regions
  - at implicit and explicit barriers
  - at entry/exit of parallel worksharing regions
  - during lock APIs
  - ….



- If you are mixing reads and writes of a variable across multiple threads, you cannot assume the reading threads see the results of the writes unless:
  - The writing threads follow the writes with a construct that implies a flush
  - The reading threads precede the reads with a construct that implies a flush

Reordering Example

```
1. a = …;
2. b = …;
3. c = …;

4. #pragma omp flush(c)
5. #pragma omp flush(a, b)

6. …= a…b…;
7. …c…;
```

- 1 and 2 may not be moved after 5
- 4 and 5 maybe interchanged at will
- 6 may not be moved before 5

Fixing Dekker's Algorithm

Where should I add a `flush`?

Dekker's Algorithm

```
flag1 = 1
if (flag2 == 0) {
   // Critical Section
}
```

```
flag2 = 1
if (flag1 == 0) {
   // Critical Section
}
```

Usage of `flush`

```
#pragma omp parallel sections
{
  // Producer
#pragma omp section
  {
    // produce data
    flag = 1;
  }
  // Consumer
#pragma omp section
  {
    while (flag == 0 ) {}
    // consume data
  }
}
```

```
#pragma omp parallel sections
{
#pragma omp section
  {
    // produce data
#pragma omp flush
#pragma omp atomic write
    flag = 1;
#pragma omp flush(flag)
  }
#pragma omp section
  {
    while (1) {
#pragma omp flush(flag)
#pragma omp atomic read
      flag_read = flag
      if (flag_read) break;
    }
#pragma omp flush
    // consume data
  }
}
```

OpenMP Optimizing Compiler

- Can reorder operations freely inside a parallel region
  - No guarantees about the ordering of operations during a parallel region excepting around flush operations
  - Parallel region contains implicit flushes
  - Cannot move operations outside of the parallel region or around synchronization operations
  - Presence of flush operations make the OpenMP memory model a variant of weak ordering

References

- S. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorials. WRL Research Report, 1995.
- D. Sorin et al. A Primer on Memory Consistency and Cache Coherence.