CS202: Advanced Operating Systems

Cache Coherence, Concurrency and Memory Consistency

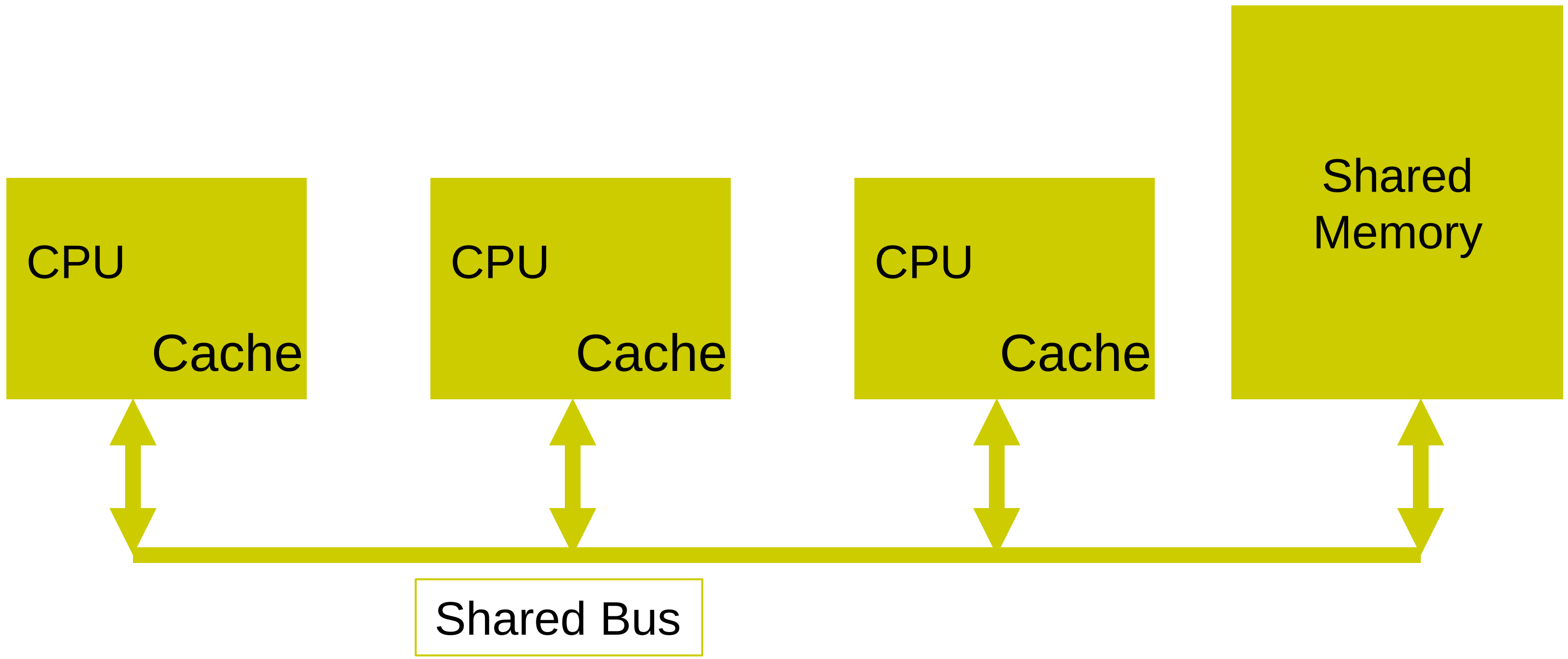
References:

- **Shared Memory Consistency Models: A Tutorial**, Sarita V. Adve & Kourosh Gharachorloo, September 1995
- **A primer on memory consistency and cache coherence**, Sorin, Hill and wood, 2011 (chapters 3 and 4)
- **Memory Models: A Case for Rethinking Parallel Languages and Hardware**, Adve and Boehm, 2010

Crash course on cache coherence

Bus-based Shared Memory Organization

Basic picture is simple :-



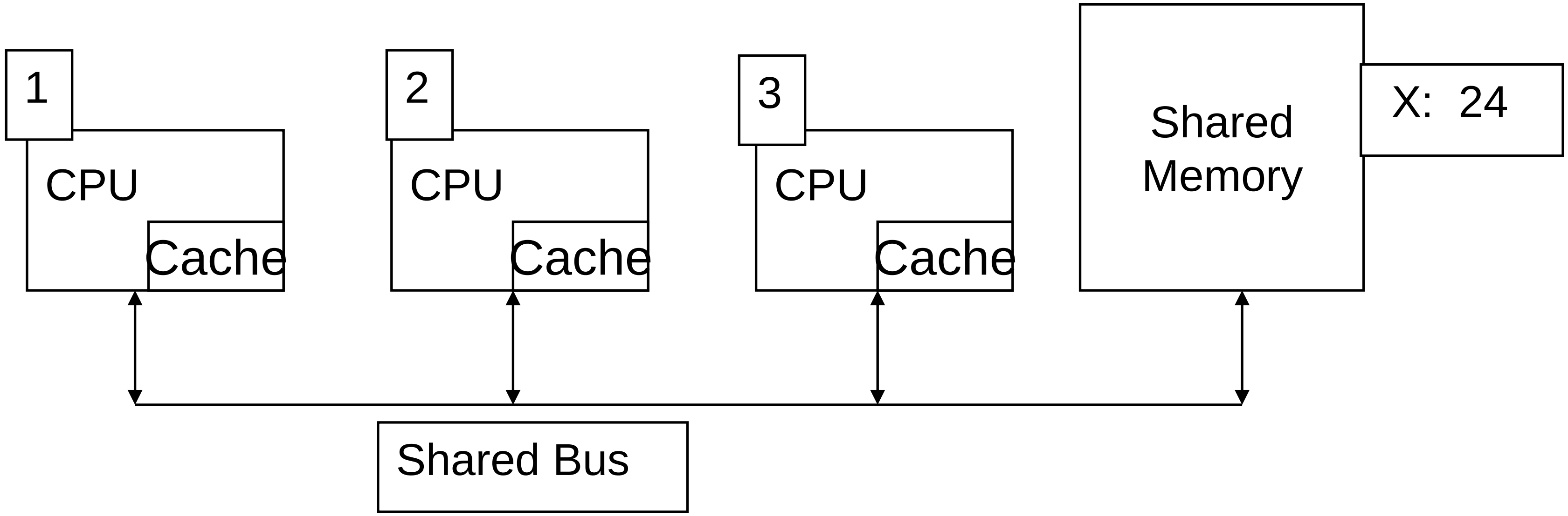
Organization

- Bus is usually simple physical connection (wires)
- Bus bandwidth limits no. of CPUs
- Could be multiple memory elements
- For now, assume that each CPU has only a single level of cache

Problem of Memory Coherence

- Assume just single level caches and main memory
- Processor writes to location in its cache
- Other caches may hold shared copies - these will be out of date
- Updating main memory alone is not enough
- What happens if two updates happen at (nearly) the same time?
  - Can two different processors see them out of order?

Example



Processor 1 reads X: obtains 24 from memory and caches it
Processor 2 reads X: obtains 24 from memory and caches it
Processor 1 writes 32 to X: its locally cached copy is updated
Processor 3 reads X: what value should it get?
Memory and processor 2 think it is 24
Processor 1 thinks it is 32

Notice that having write-through caches is not good enough

Cache Coherence

- Try to make the system behave as if there are no caches!
- How? Idea: Try to make every CPU know who has a copy of its cached data?
    - too complex!
- More practical:
    - Snoopy caches
        - Each CPU snoops memory bus
        - Looks for read/write activity concerned with data addresses which it has cached.
            - What does it do with them?
        - This assumes a bus structure where all communication can be seen by all.
- More scalable solution: ‘directory based’ coherence schemes

Snooping Protocols

- Write Invalidate
  - CPU with write operation sends invalidate message
  - Snooping caches invalidate their copy
  - CPU writes to its cached copy
    - Write through or write back?
  - Any shared read in other CPUs will now miss in cache and re-fetch new data.

Snooping Protocols

- Write Update
  - CPU with write updates its own copy
  - All snooping caches update their copy
- Note that in both schemes, problem of simultaneous writes is taken care of by bus arbitration - only one CPU can use the bus at any one time.
- Harder problem for arbitrary networks

Update or Invalidate?

- Which should we use?
- Bus bandwidth is a precious commodity in shared memory multi-processors
  - Contention/cache interrogation can lead to 10x or more drop in performance
  - (also important to minimize false sharing)
- Therefore, invalidate protocols used in most commercial SMPs

Cache Coherence summary

- Reads and writes are atomic
  - What does atomic mean?
    - As if there is no cache
- Some magic to make things work
  - Have performance implications
  - …and therefore, have implications on performance of programs

Memory Consistency

- Formal specification of memory semantics
- Guarantees as to how shared memory will behave on systems with multiple processors
- Ordering of reads and writes
- Essential for programmer (OS writer!) to understand

Why Bother?

- Memory consistency models affect *everything*
  - Programmability
  - Performance
  - Portability
- Model must be defined at all levels
- Programmers and system designers care

Uniprocessor Systems

- Memory operations occur:
  - One at a time
  - In program order
- Read returns value of last write
  - Only matters if location is the same or dependent
  - Many possible optimizations
- **Intuitive!**

How does a core reorder? (1)

- Store-store reordering:
  - Non-FIFO write buffer
- Load-load or load-store/store-load reordering:
  - Out of order execution

- Should the hardware prevent any of this behavior?

Multiprocessor: Example

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;	

Cont'd

TABLE 3.2: One Possible Execution of Program in Table 3.1.				
cycle	**Core C1**	**Core C2**	**Coherence state of data**	**Coherence state of flag**
1	S2: Store flag=SET		read-only for C2	read-write for C1
2		L1: Load r1=flag	read-only for C2	read-only for C2
3		L2: Load r2=data	read-only for C2	read-only for C2
4	S1: Store data=NEW		read-write for C1	read-only for C2

- S2 and S1 reordered
  - Why? How?

Example 2

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*/

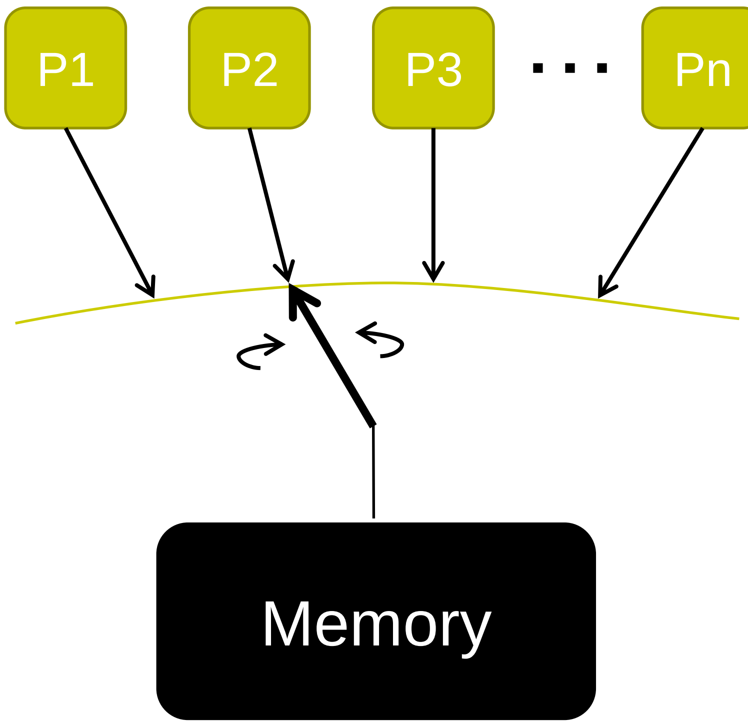
Intuitively, one might expect that there are three possibilities:

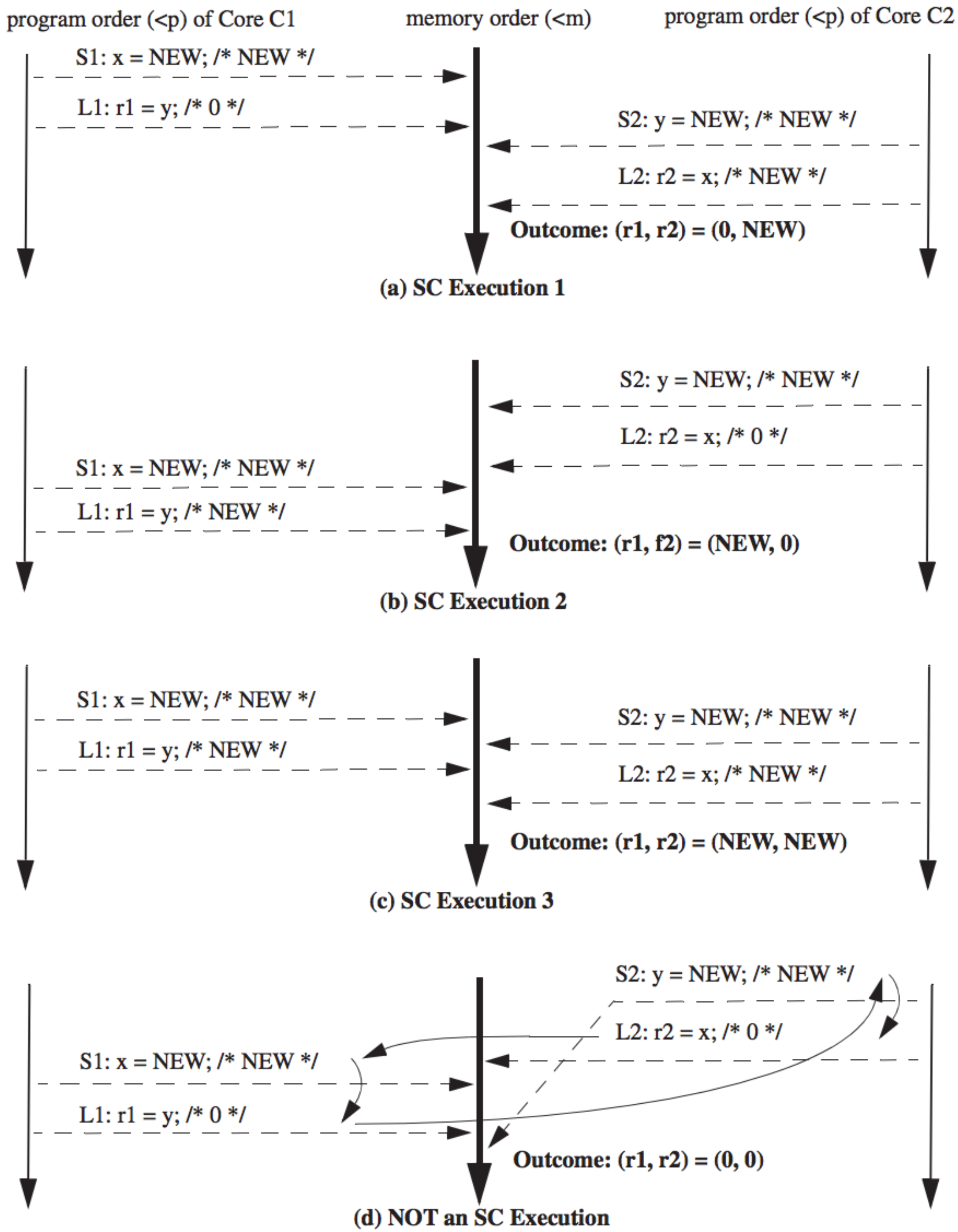
- (r1, r2) = (0, NEW) for execution S1, L1, S2, then L2
- (r1, r2) = (NEW, 0) for S2, L2, S1, and L1
- (r1, r2) = (NEW, NEW), e.g., for S1, S2, L1, and L2

Surprisingly, most real hardware, e.g., x86 systems from Intel and AMD, also allows (r1, r2) = (0, 0) because it uses first-in–first-out (FIFO) write buffers to enhance performance. As with the example in Table 3.1, all of these executions satisfy cache coherence, even (r1, r2) = (0, 0).

Sequential Consistency

- The result of any execution is the same as if all operations were executed on a single processor
- Operations on each processor occur in the sequence specified by the executing program




program order (<p) of Core C1
memory order (<m)
program order (<p) of Core C2
S1: x = NEW; /* NEW */
L1: r1 = y; /* 0 */
S2: y = NEW; /* NEW */
L2: r2 = x; /* NEW */
Outcome: (r1, r2) = (0, NEW)
(a) SC Execution 1
S2: y = NEW; /* NEW */
L2: r2 = x; /* 0 */
S1: x = NEW; /* NEW */
L1: r1 = y; /* NEW */
Outcome: (r1, f2) = (NEW, 0)
(b) SC Execution 2
S1: x = NEW; /* NEW */
L1: r1 = y; /* NEW */
S2: y = NEW; /* NEW */
L2: r2 = x; /* NEW */
Outcome: (r1, r2) = (NEW, NEW)
(c) SC Execution 3
S2: y = NEW; /* NEW */
L2: r2 = x; /* 0 */
S1: x = NEW; /* NEW */
L1: r1 = y; /* 0 */
Outcome: (r1, r2) = (0, 0)
(d) NOT an SC Execution

One execution sequence

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;	

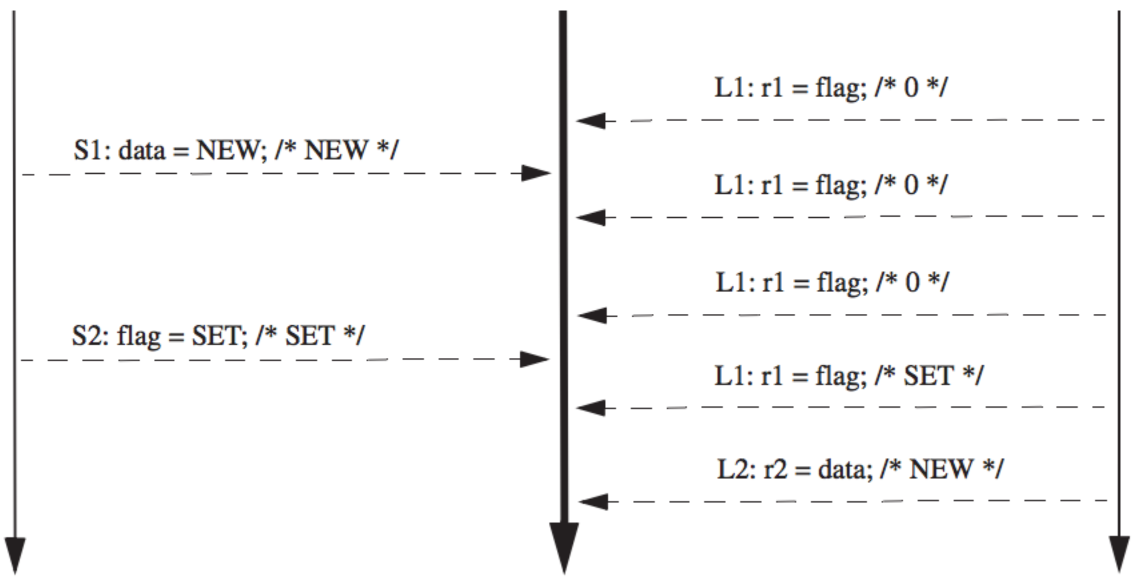


FIGURE 3.1: A Sequentially Consistent Execution of Table 3.1's Program.

S.C. Disadvantages

- Difficult to implement!
- Huge lost potential for optimizations
  - Hardware (cache) and software (compiler)
  - Be conservative: err on the safe side
  - Major performance hit

Relaxed Consistency

- **Program Order** relaxations *(different locations)*
  - W → R; W → W; R → R/W
- **Write Atomicity** relaxations
  - Read returns another processor's Write early
- Combined relaxations
  - Read your own Write *(okay for S.C.)*
- *Safety Net* – available synchronization operations
- *Note: assume one thread per core*

Synchronization is broken!

- How can we solve this problem?

- Answer: Memory Barrier/Fence
  - A special complier or CPU instruction that enforces an ordering constraint
  - Compiler: asm volatile ("" ::: "memory");
  - CPU: mfence/lfence

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;	