CS152/252A Fall 2026

Memory Consistency & Synchronization

Discussion 9

1 Cache Coherency: MESI & MOESI

Recap. The MSI protocol keeps private caches coherent by giving every cache line one of three states: Modified (M), Shared (S), or Invalid (I).

1.1 Under MSI, Core 0 loads `A` while no other cache holds it, then stores 7 to `A`. What bus transactions does the store cause? Which of them is wasted, and how could we avoid it?

MESI adds an **Exclusive (E)** state for exactly this case. A line can be in E only if both hold:

- **Only copy**: this cache is the only one with the line, and every other cache has it in I.
- **Clean**: the line matches main memory, so evicting it needs no writeback.

On a read miss, a cache enters E if no other cache reports holding the line (snooping caches assert a *shared* signal on the bus when they have a copy), and S otherwise. A write hit to an E line moves to M with no bus transaction, since there is no one to invalidate:

Step	**P0**	**P1**
Initial	I	I
P0 reads `X`	E	I
P0 writes `X`	M	I

MOESI adds a fifth **Owned (O)** state, which lets a dirty line be shared without first writing it back to memory:

- When another core reads a line held in M, MSI and MESI write the data back to memory and drop to S. Under MOESI, the holder supplies the data and moves to O instead, and memory is left stale. This saves the writeback.
- At most one cache can hold a line in O, and any other copies are in S.
- The owner supplies the data to caches that request the line, and must write the line back to memory if it evicts it.

In the MOESI state diagram, a **probe** is a coherence request sent to a cache to check whether it holds a particular line and, if necessary, update the line's state or supply data:

- **Local request**: your own core reads or writes the line.
- **Probe read**: another core requests to read the line.
- **Probe write**: another core requests write ownership of the line.

1.2 `X` starts as 0 in memory, and both caches start in I. All operations access the same cache line, and there are no evictions. Give the states (P0, P1) after each step under each protocol.

Step	**MSI**	**MESI**	**MOESI**
Initial	I, I	I, I	I, I
P0 reads `X`			
P0 writes `X = 1`			
P1 reads `X`			

1.3 *[Extra] HW / exam practice.* Three processors, `P0`–`P2`, run the MOESI protocol, and every access below goes to the same cache line `X`. DRAM initially holds `X = 6`. For each step, give the state of `X` in each cache, the bus operation, whether the step causes bus data traffic, whether the access misses in the local cache, and the value of `X` in DRAM.

- **Bus operations**: `BusRd` is a read miss, `BusRdX` is a write miss, and `BusInv` is a write hit to an S or O line (invalidate other copies).
- **Bus data traffic**: 1 for `BusRd`, `BusRdX`, or a dirty writeback, otherwise 0.
- **Read/write misses**: 1 if the access misses in the local cache, otherwise 0.

Step	P0, P1, P2	Bus op	Bus data traffic	Read/write misses	DRAM `X`
Initial	I, I, I	–	0	0	6
`P0` read `X`	E, I, I	`BusRd`	1	1	6
`P0` write `X = 7`					
`P1` read `X`					
`P2` read `X`					
`P0` write `X = 8`					
`P1` write `X = 9`					
`P2` read `X`					
`P1` evicts `X`					

2 Directory-Based Coherence

With a **snooping** protocol, every cache eavesdrops on every coherence request broadcast on the bus. That is hard to scale: every miss goes to every core, and the constant broadcasts hurt performance.

A **directory** is the alternative. It acts as a bookkeeper: for each line, it tracks which caches hold a copy (for example, a state field and one sharer bit per processor), and contacts only those caches when the line is requested or modified.

2.1 What is a drawback of using a directory-based cache?

2.2 If a cache evicts a line it holds in S, why might it need to notify the directory? What happens if it doesn't?

3 Synchronization

This section doesn't cover everything. Review barriers and exponential backoff from lecture on your own.

In a multiprocessor, we need a way to explicitly hand off ownership of shared data between cores. For example, consider a lock built in software from ordinary loads and stores:

```
if (lock == 0)
    lock = 1;
/* critical section */
```

Time	Core 0	Core 1
t0	Sees `lock == 0`	
t1		Sees `lock == 0`
t2	Sets `lock = 1`	
t3		Sets `lock = 1`

Both cores enter the critical section! The fix is hardware support for an **atomic** operation: one that happens all at once, with no other memory operation interleaved. For example, **atomic swap** (also called **atomic exchange**) writes 1 to `lock` and returns the old value, with nothing in between:

- If it returns 0, we acquired the lock: no one held it, and we wrote 1 to it.
- If it returns 1, we didn't acquire the lock: someone else holds it, and we wrote 1 over 1.

```
while (swap(lock, 1) == 1);  // spin
/* critical section */
```

Time	Core 0	Core 1
t0	`swap` returns 0	
t1		`swap` returns 1
t2	In critical section	
t3		Spins on `lock`

There is no gap between checking the lock and setting it, so only one core gets in. Other atomic instructions include:

- **Atomic swap**: swap a register and a memory value (RISC-V `amoswap`).
- **Test-and-set**: read a memory value and, if it is 0, set it to 1.
- **Fetch-and-increment**: return the old value, and increment it in memory.
- **Compare-and-swap (CAS)**: if memory holds an expected value, replace it with a new value. Either way, return the old value.
- **Load-reserved / store-conditional (LR/SC)**: a pair of instructions. The store fails if anyone wrote the address since the load, or if a context switch occurred. More on this below.

Higher-level primitives such as mutexes and semaphores are built on top of these.

3.1 Imagine 10,000 threads fighting for one lock. What are some potential problems?

3.2 How might we solve these problems?

Ticket locks. A ticket lock is a FIFO lock built from two counters:

- `next_ticket`: the next number to hand out.
- `now_serving`: the number whose turn it is.

Acquire: atomically take a ticket with fetch-and-increment, then spin until `now_serving` matches it.
Release: increment `now_serving`.

It works like a deli counter. You grab a ticket from `next_ticket`, then wait and watch the `now_serving` display for your number. When it's your turn, you enter the critical section and do your thing, and when you leave, you increment `now_serving` to call the next person. (Unlike at Costco, nobody calls numbers for you: each customer calls the next one.)

3.3 *(3 points)* Consider the atomic fetch-and-increment operation, which performs the following function atomically:

```
// atomic
int FetchAndIncrement(int *location) {
    int value = *location;
    *location = value + 1;
    return value;
}
```

Briefly describe (in at most 3 sentences) how one might implement a **ticket** lock using fetch-and-increment.

3.4 *Fun question.* Consider this compare-and-swap instruction:

```
cas rd, rs2, (rs1)
    old = mem[rs1]      // load the current value
    if (old == rd)      // rd holds the expected value
        mem[rs1] = rs2  // rs2 holds the desired value
    rd = old            // return the old value in rd
```

How would you implement it in the classic five-stage pipeline? Do you need to add anything to the pipeline? Does supporting atomics justify your changes?

Load-reserved / store-conditional (LR/SC) splits an atomic read-modify-write into two instructions:

- `lr.w` (**load-reserved**, also called load-link): read this value, and watch this address for me.
- `sc.w` (**store-conditional**): write my new value, but only if the address hasn't been written since my `lr.w`. Unlike a normal store, it returns a value: 0 if it succeeded, nonzero if it failed.

```
while (lock != 0);
lock = 1;
/* critical section */
```

```
      li   t0, 1
try:  lr.w t1, (a0)      # t1 = lock
      bnez t1, try       # lock held
      sc.w t2, t0, (a0)  # try to set
lock = 1
      bnez t2, try       # SC failed
      # critical section
```

Time	Core 0	Core 1
t0	`lr.w` reads 0	
t1		`lr.w` reads 0
t2	`sc.w` succeeds	
t3		`sc.w` fails!
t4	Has the lock	Retries (`bnez t2`)

3.5 At t3, how does Core 1 know that the lock has been written since its `lr.w`?

3.6 Two cores, A and B, both run the atomic increment below on the same counter. The counter starts at 0, and both caches start in I. Give each core's MSI state for the counter's line after each step, and describe what happens.

```
try:  lr.w t1, (a0)       # read counter
      addi t1, t1, 1      # add 1
      sc.w t2, t1, (a0)   # try to write it back
      bnez t2, try        # SC failed: retry
```

Step	Core A	Core B	Notes
Initial	I	I	–
A `lr.w`			
B `lr.w`			
B `sc.w`			
A `sc.w`			
A `lr.w`			
A `sc.w`			

4 Synchronization Exam Questions

4.1 *(2 points)* If we run a program on a multiprocessor system with an underlying shared main memory and no caches, are atomic instructions still required to enable synchronization?

4.2 *(1 point)* Which of the following are hardware challenges when implementing synchronization? Select all that apply.

- ☐ Ensuring atomicity of read-modify-write operations
- ☐ Maintaining coherence of shared data across cores
- ☐ Reducing instruction count
- ☐ Eliminating all memory accesses

5 Memory Consistency

5.1 Core A holds a lock and runs the code on the left, and Core B runs the code on the right. `balance` starts at 0. Assuming a working cache coherence implementation, is it possible for Core B to print 0?

```
// Core A (holds the lock)
balance = 100;
lock = 0;
```

```
// Core B
while (exchange(lock, 1) == 1);
print(balance);
```

Memory consistency vs. cache coherence.

- **Coherence** describes the legal values that a single memory address can return.
- **Consistency** describes properties across *all* memory addresses: when a write to `X` becomes visible to other processors, relative to reads and writes to other addresses.

Coherence alone does not imply any particular memory consistency model!

Sequential consistency (SC), the intuitive starting point: a multiprocessor is sequentially consistent if the result of any execution matches *some* single sequential order of all processors' operations, with each processor's operations appearing in program order.

Consider the following two threads, executing on two different cores. Memory locations `A`, `B`, and `C` are initialized to zero, and we are interested in the final values of `P1.x2`, `P1.x3`, and `P2.x2`.

P1:	**P2:**
`li x1, 1`	`li x1, 2`
`I1 lw x2, A`	`J1 sw x1, B`
`I2 sw x1, C`	`J2 lw x2, C`
`I3 lw x3, B`	`J3 sw x1, A`

5.2 Under sequential consistency, give all possible sets of values of `P1.x2`, `P1.x3`, and `P2.x2`, and a possible instruction ordering that causes each one.

Operation order	**`P1.x2`**	**`P1.x3`**	**`P2.x2`**

Relaxing memory consistency models. The strictness of SC suggests two major targets for relaxation:

(a) **Program order requirement**: when can one thread's memory operations be reordered with respect to one another?

(b) **Write atomicity requirement**: when can a processor see the effect of a write relative to other processors in the system?

There are four types of program-order constraints: (1) Write → Read, (2) Write → Write, (3) Read → Write, and (4) Read → Read.

5.3 Can you think of a microarchitectural optimization made possible by relaxed ordering?

5.4 Back to the `balance` example at the start of this section. Now assume we forbid W → W relaxation. Is it still possible for Core B to print 0?

Relaxing write atomicity comes in two flavors:
(a) Can a processor see its own write before others can?
(b) Can processor A see processor B's write before it is visible to the rest of the memory system?

A system is **multi-copy atomic** if once a processor's write becomes visible to another processor, it becomes visible to all processors. In a **non-multi-copy-atomic** system, a write by one processor can become visible to a subset of the other processors before it becomes visible to all of them, for example with hierarchical shared buffers.

5.5 Can you think of a microarchitectural optimization that would make flavor (1) true? What about flavor (2)?

5.6 In general, what is the difference between a weak and a strong memory consistency model? What are the pros and cons of each?

Categorizing relaxed models. A ✓ means the relaxation is allowed by straightforward implementations of that model (Adve & Gharachorloo, *Shared Memory Consistency Models: A Tutorial*, 1996):

Model	W → R	W → W	R → RW	Read others' write early	Read own write early	Safety net
SC					✓	
IBM 370	✓					Serialization instructions
TSO	✓				✓	RMW
PC	✓			✓	✓	RMW
PSO	✓	✓			✓	RMW, STBAR
WO	✓	✓	✓		✓	Synchronization
RCsc	✓	✓	✓		✓	Release, acquire, nsync, RMW
RCpc	✓	✓	✓	✓	✓	Release, acquire, nsync, RMW
Alpha	✓	✓	✓		✓	MB, WMB
RMO	✓	✓	✓		✓	Various MEMBARs
PowerPC	✓	✓	✓	✓	✓	SYNC

RISC-V currently has two memory models: the default, **weak memory ordering (RVWMO)**, and the optional **total store order (RVTSO)** extension.

6 Relaxed Orderings

Back to the two-thread program. Memory locations `A`, `B`, and `C` are initialized to zero.

```
        P1:                P2:
     li x1, 1           li x1, 2
I1   lw x2, A      J1   sw x1, B
I2   sw x1, C      J2   lw x2, C
I3   lw x3, B      J3   sw x1, A
```

6.1 If we relax Write → Read ordering, which instruction(s) can be reordered?

6.2 Give the new possible set(s) of values if we relax Write → Read ordering, and a possible instruction ordering that causes each one.

Operation order	P1.x2	P1.x3	P2.x2

6.3 If we also relax Write → Write ordering, which additional instruction(s) can be reordered?

6.4 Give the new possible set(s) of values if we also relax Write → Write ordering, and a possible instruction ordering that causes each one.

Operation order	P1.x2	P1.x3	P2.x2

6.5 If we also relax Read → Read and Read → Write ordering, which additional instruction(s) can be reordered?

6.6 Give the new possible set(s) of values if we relax all orderings, and a possible instruction ordering that causes each one.

Operation order	P1.x2	P1.x3	P2.x2

7 Fences

The RISC-V `fence` instruction constrains memory ordering. It comes in several fine-grained variants:

Ordering enforced	**Instruction**
Write ⟶ Read	`fence w,r`
Write ⟶ Write	`fence w,w`
Read ⟶ Write	`fence r,w`
Read ⟶ Read	`fence r,r`

Constraints can be combined: `fence r,rw` is the same as `fence r,r` followed by `fence r,w`. `fence` with no arguments is a full fence, like `fence rw,rw`.

7.1 *(Sp19)* Under a fully relaxed memory consistency model, insert fences where necessary to ensure this code functions correctly. Use the least restrictive fences for full credit. Assume each thread executes its code only once.

Processor 1:

```
      li   a0, 1
      sw   data, request
      sw   a0, go
spin: lw   a1, done
      beqz a1, spin
      lw   a2, result
      sw   zero, done
```

Processor 2:

```
spin: lw   a1, go
      beqz a1, spin
      lw   a2, request
      sw   zero, go
      ...  a3 = process request
      sw   a3, result
      li   a0, 1
      sw   a0, done
```

8 True or False

8.1 Sequential consistency is guaranteed if all processors have in-order pipelines.

8.2 A high-level language with a sequentially consistent memory model can be implemented on an ISA with a weaker memory model if fence instructions are provided.

8.3 Suppose an ISA specifies a non-multi-copy-atomic memory model, but a particular hardware implementation provides sequential consistency. Will software written for this ISA execute correctly on this machine?