Module 15 – Memory Consistency

(ECE M116C- CS M151B) Computer Architecture Systems

Nader Sehatbakhsh
Department of Electrical and Computer Engineering
University of California, Los Angeles

Next Problem: *Consistency!*

Consistency Problem

- if initially `M[Z] = M[Y] = 0, x3= 100, x4=200`

```
Core1: lw x1, Z(x0)          Core2: lw x2, Y(x0)
...                      ...
Core1: sw x3, Y(x0)          Core2: sw x4, Z(x0)
```

→ What are the potential values for `x1` and `x2`?

Wait, but why?

- RAW?
- Correctness?
- LSQ?
    - If the addresses are not the same, why wait?
    - Compiler can reorder, too!

Consistency Problems

- SW and LW (to different addresses) within one core can be reordered and that will mess up assumptions in OTHER cores.

- The timing between cores might be off! (even without reordering).

Memory Consistency vs. Coherency

- Consistency is about ordering of parallel accesses between *different* addresses.
- Coherency is about ordering of parallel accesses to the *same* address.

So, how to fix this?

The Timing Problem

- Timing between two cores is not the hardware or reordering problem, so it's programmer's job to fix that using synchronization strategies.

- We will discuss this later!

The Reordering Problem

- That is the optimization done by hardware and/or compiler and programmer cannot do anything about this!

- We need to fix this!

Memory Model

- We first need to know what are the possibilities.

→ Memory Model: Interactions between threads and the shared memory. In other words, *what are the possible values for a load?*

Sequential Consistency

1. Processors see their own loads and stores in program order.

2. Processors see others' loads and stores in program order.

3. All processors see the same global load/store ordering.

SC: *what programmers expected.*

Sequential Consistency

- Simplest scenario!
- SC makes multicore system indistinguishable from multi-programmed in-order uniprocessor.

Sequential Consistency

- Simplest scenario!
- SC makes multicore system indistinguishable from multi-programmed in-order uniprocessor.
- Too slow!
    - Too many restrictions (what are they?)

Memory Operation Ordering

- Four types of memory operation orderings
  - W→R: write must complete before subsequent read.
  - R→R: read must complete before subsequent read.
  - R→W: read must complete before subsequent write.
  - W→W: write must complete before subsequent write.

- This is for ***different*** addresses, so no RAW hazards!

What can we do?

- We really need to be able to re-order memory accesses.
  - Most lines are not even shared!
  - Store does not need to even wait
  - We have the ability to do so!

What can we do?

★ We need to compromise, and use a “weaker” or “relaxed” model.

→ *Not all loads and stores need to be in-order.*

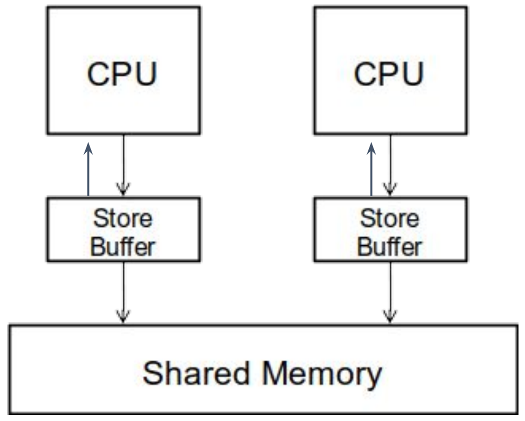
Memory Operation Ordering

- Four types of memory operation orderings
  - W→R: write must complete before subsequent read.
  - R→R: read must complete before subsequent read.
  - R→W: read must complete before subsequent write.
  - W→W: write must complete before subsequent write.
- *Sequential consistency maintains all four orderings.*
- *Relaxed memory consistency* models allow ***certain*** orderings to be violated.

Total Store Order Model

Total Store Order (TSO) Model

- Processor P can read B before its write to A is seen by all processors.
  - Each processor can move its own reads in front of its own writes.
  - *Remember LSQs?*



Total Store Order (TSO) Model

- Processor P can read B before its write to A is seen by all processors.
  - Each processor can move its own reads in front of its own writes.

- NOTE: Read by other processors *cannot* return new value of A until the write to A is observed by all processors.

How TSO helps?

`a = 0, flag = 0`

thread 1	**thread 2**

`store 1 → a`
`store 1 → flag`
	`loop: if (flag == 0) goto loop`
	`      load a`

➔ *Can `load` read 0?*

TSO vs. SC

- if initially `M[Z] = M[Y] = 0`

```
Core1: lw x1, Z          Core2: lw x2, Y
...                      ...
Core1: sw Y, 100         Core2: sw Z, 200
```

→ What are the potential values for `x1` and `x2`?

TSO Summary

- Improving the per-core performance by allowing the reordering within the core!

- Still not observable by other cores (timing problem)

Weak Consistency

- Different assumptions
    - What and when a memory access is visible to the other cores.

- Performance vs. Complexity

- Different architectures use different models.

- More about it soon!

RISC-V Memory Model

- RISC-V OoO (BOOM) follows the RVWMO memory consistency model (RISC-V weak memory ordering).

RISC-V Memory Model

- Loads are optimistically fired to memory on arrival to the LSU (for performance).
- Simultaneously, the load instruction compares its address with all of the store addresses that it depends on.
  - If there is a match, the memory request is killed.
  - If the corresponding store data is present, then the store data is forwarded to the load and the load marks itself as having succeeded.
  - If the store data is not present, then the load goes to sleep.
    - Loads that have been put to sleep are retried at a later time.

RISC-V Memory Model

- RISC-V OoO (BOOM) follows the RVWMO memory consistency model (RISC-V weak memory ordering).

- BOOM currently exhibits the following behavior:
    - Write → Read constraint is relaxed (newer loads may execute before older stores).
    - Read → Read constraint is maintained (loads to the same address appear in order).
    - A thread can read its own writes early.

How to write a correct program?

- The memory model is given.

Simple Solution

- Let it be!
  - Race-free programming.

Not very simple solution

- Have the hardware to track potential violations!
  - Each core reorders but if another core needs that data, it should recover.
  - Very costly!

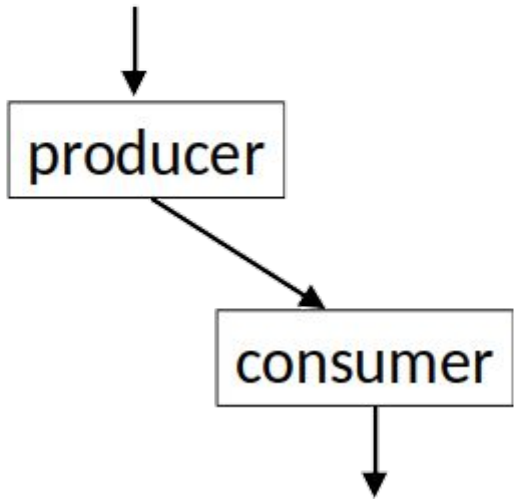
More realistic solution

- Use synchronization!
  - The programmer (and/or hardware) use some primitives to synchronize different programs.
  - This solves both timing and ordering problems.

Synchronization

- Two major solutions:

  1- Producer-consumer model

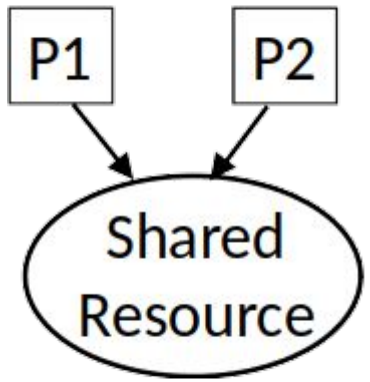


Synchronization

- Two major solutions:

  1- Producer-consumer model

  2- Mutual exclusiveness

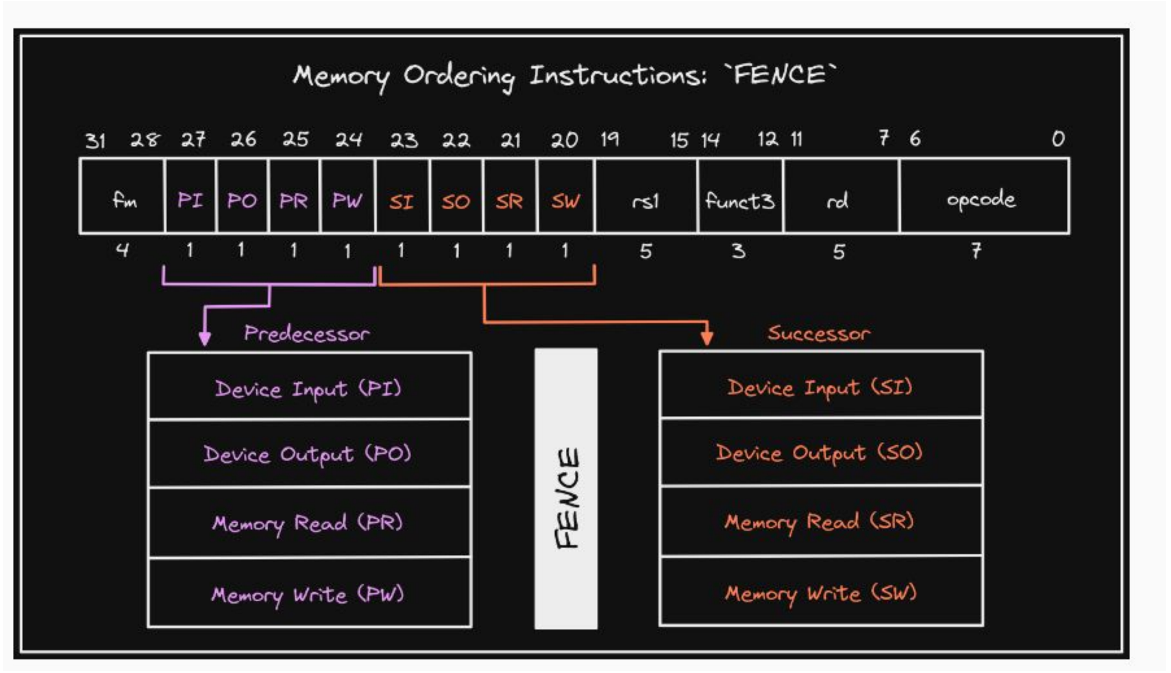


Solution

- Use ***fences!***
  - FENCE is an instruction that enforces all memory operations *before* it to complete before fence is executed.
  - All memory operations *after* FENCE must wait for FENCE to be finished.
  - This happens *within* each core!

FENCE in RISC-V

- FENCE (rwio)
  - fence r,r



What about between cores?

Semaphore

- Create some barriers!
- This will synchronize all the processors.
- Typically implemented in software.

Synchronization Using Semaphores

```
Core1: lw x1, Z          Core2: lw x3, Y
…                        …
Core1: sw Y, 100         Core2: sw Z, 200
```

How to implement barrier/semaphore in hardware?

- Hardware-or with one-hot encoding → set a flag to one when a core reaches to the barrier. Wait until all flags are one!

What about mutual exclusiveness?

Mutual Exclusiveness

- Making sure only one process has access to the shared region at any given time.

Mutual Exclusiveness

- Making sure only one process has access to the shared region at any given time.

→ How?

How to fix interleaving?

- Use Mutex/Locks!

```
      Acquire (lock)          Acquire (lock)

Core1: lw x1, Z          Core2: lw x3, Y
…                        …
Core1: sw Y, 100         Core2: sw Z, 200

      Release (lock)
```

Release Consistency

- It is guaranteed that all reads and writes are executed and visible to all cores when the lock is released.

- Other than this guarantee, any ordering outside the acquire/release blocks is possible.

How to implement `acquire/release`?

```
acquire(lock) {
   while (lock != 0)
   {} // busy wait
   // lock is acquired at this point
   lock = 1; // making sure no one else can get it
   }
```

Assembly Version

```
loop:
   lw ra, 0(x1)
   bnez ra, loop
   addi ra, x0, 1
   sw ra 0(x1)
```

- Assume that Mem[x1] is where lock is stored.

How to implement `acquire/release`?

```
release(lock) {
  lock = 0;
  }
```

- Assembly Version:

```
  st r0 -> 0(&lock)
```

Problem?

```
(Core 1)
loop:
   lw ra, 0(x1)
   bnez ra, loop
   addi ra, x0, 1
   sw ra 0(x1)
```

```
(Core 2)
loop:
   lw ra, 0(x1)
   bnez ra, loop
   addi ra, x0, 1
   sw ra 0(x1)
```

Lock and Release

- Makes sense but we need to implement it such that locks are exclusives (i.e., only one lock is active at any given time).

Atomic Instructions

- A hardware primitive that guarantees instructions are executed in-order and without interruption.

How to implement Mutex?

- Hardware-enforce instruction (called test and set).

```
TS(int x) {
   oldval = SWAP(x,1); // atomic swap
   return oldval;
}
```

  - Other variants: *compare-and-swap, fetch-and-add*

How SWAP is implemented?

- Load-reserved, store-conditional (in RISC-V)

```
Label:
    load-link x2, 0(x1)
    addi x2, x0,1
    store-conditional x2,0(x0)
    branch-not-zero label // check for failure
```

LR and SC

- On load-link, processor remembers address.
- Then checks for writes by other processors.
    - If write is detected, store-conditional will fail.

- For MESI:
    - Load-Reserved ensures line in cache in Exclusive/Modified state.
    - Store-Conditional succeeds if line still in Exclusive/Modified state

Recap

- Locks
  - Test-and-set
  - Atomic
  - LR and SC

➔ This can be combined with a very weak consistency model (called release consistency).

Optimizations?

- Too many `SW`s when waiting/spinning.
  → Use test-test-and-set

Test-test-and-set

```
TTS(int x) {
oldval = x; // the read of x
if (oldval == 0) oldval = SWAP(x,1);
return oldval;
}
```

➔ `while (TTS(lock) == 1);`

Optimizations?

- Too many `SW`s when waiting/spinning.
  → Use test-test-and-set

- Unfairness?
  → Use queues

- Multiple/Parallel Locks

Bottom-Line

- Range of possibilities!
- The burden is (mostly) on the developer!

CA 3

- Designing a MOESIF cache coherence protocol.

End of Presentation

Acknowledgement

- This course is partly inspired by the following courses created by my colleagues:
  - CS152, Krste Asanovic (UCB)
  - 18-447, James C. Hoe (CMU)
  - CSE141, Steven Swanson (UCSD)
  - CIS 501, Joe Devietti (Upenn)
  - CS4290, Tom Conte (Georgia Tech)
  - 252-0028-00L, Onur Mutlu (ETH)