CS152 Discussion Section

Memory Consistency

April 23
Spring 2024

Agenda

- Memory consistency models
  - Much of this material is taken from: Adve and Gharachorloo.
  - Shared Memory Consistency Models: A Tutorial (1995)
- Lab 5 due Wednesday May 1
- PS5 due Monday Apr 29

Memory Consistency: Other half of the ISA

- The instruction set and architectural state do not completely define the behavior of the ISA
- Sequential ISA only specifies that each processor observes its own memory operations in program order
- Need a *memory consistency model*, which defines the legal set of values that loads can return across multiple hardware threads

Memory Consistency vs Cache Coherence

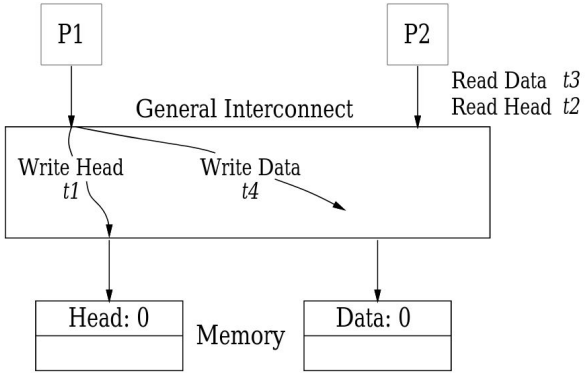
- *Coherence* describes the legal values that a *single* memory address could return
- *Consistency* describes properties across all memory address

Coherence alone does not imply any particular memory consistency model!

Memory Consistency and Caches

Q: Do memory consistency models apply only to systems with caches?

A: No — consider a cache-less system with multiple memory banks

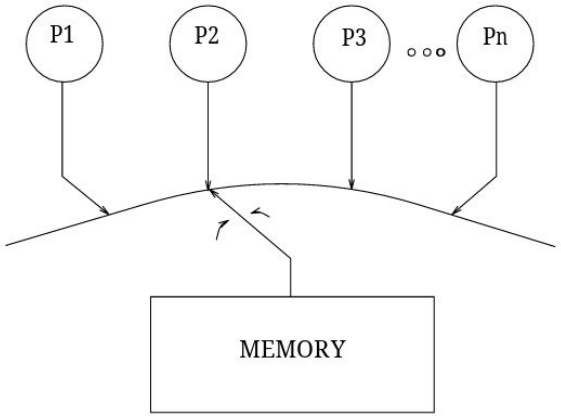


P1	P2
Data = 2000	while (Head == 0) {;}
Head = 1	... = Data

Sequential Consistency: Intuitive Starting Point

Definition: [A multiprocessor system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs (1979)



Worksheet Q1

Consider the following two threads executing on two different cores. 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`

We are interested in the final values of P1.x2, P1.x3, and P2.x2.

Sequential Consistency (Q1.1)

	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`

Give all possible sets of values of P1.x2, P1.x3, and P2.x2 under sequential consistency.

Operation Order	P1.x2	P1.x3	P2.x2

Sequential Consistency (Q1.1)

```
     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
```

Give all possible sets of values of P1.x2, P1.x3, and P2.x2 under sequential consistency.

Operation Order	P1.x2	P1.x3	P2.x2
I1 I2 I3 J1 J2 J3	0	0	1
J1 J2 J3 I1 I2 I3	2	2	0
J1 I1 I2 I3 J2 J3	0	2	1
I1 J1 J2 I2 I3 J3	0	2	0

Relaxing Memory Consistency Models

Strictness of SC suggests two major targets for relaxation:

1. *Program order requirement*: When can one thread's memory operations be reordered with respect to one another
2. *Write atomicity requirement*: When can a processor see the effect of a write relative to other processors in the system

Relaxing Program Order Requirement

Four types of ordering constraints:

1. Write → Read
2. Write → Write
3. Read → Write
4. Read → Read

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

Relaxing Program Order Requirement

Four types of ordering constraints:

1. Write → Read
2. Write → Write
3. Read → Write
4. Read → Read

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

A: Out-of-order execution of loads before stores in program order

Relaxing Write Atomicity

Two flavors:

1. Can a processor see its own write before others?
2. Can processor A see some other processor B's writes before they are made visible to the rest of the memory system?

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

Relaxing Write Atomicity

Two flavors:

1. Can a processor see its own write before others?
2. Can processor A see some other processor B's writes before they are made visible to the rest of the memory system?

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

A:

1. Load bypassing from a store queue or write buffer
2. Multithreading

Worksheet Q1.2

Weak Versus Strong Memory Consistency Models

In general, what is the difference between a weak and a strong memory consistency model? Give an example of each.

Worksheet Q1.2

Weak Versus Strong Memory Consistency Models

In general, what is the difference between a weak and a strong memory consistency model?

Weak memory models relax some combination of the program-order and/or write-atomicity constraints imposed by stronger memory models.

Advantage of a strong memory model:

- More intuitive memory semantics make it easier to write and debug concurrent code
- Similarly, easier to write correct compilers for high-level languages

Advantage of a weak memory model:

- Easier to build simple implementations (more design flexibility, many simple optimizations would violate a strong MCM without considerably more complexity)
- Easier to build high-performance implementations, as the implementation can aggressively reorder memory ops without needing to speculate on the MCM

Categorizing Relaxed Models

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

RISC-V Memory Models

RISC-V currently has two:

1. Default: Weak memory ordering (RVWMO)
2. Optional extension: Total store order (RVTSO)

Relaxing Program Order Requirement

Four types of ordering constraints:

1. Write → Read
2. Write → Write
3. Read → Write
4. Read → Read

W→R Relaxation (Q1.3)

	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`

Give all new possible sets of values if we relax Write → Read ordering and the instruction orderings that caused them.

Operation Order	P1.x2	P1.x3	P2.x2

W→R Relaxation (Q1.3)

	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`

Give all new possible sets of values if we relax Write → Read ordering and the instruction orderings that caused them.

Operation Order	P1.x2	P1.x3	P2.x2
I1 I3 J2 I2 J1 J3	0	0	0

W→W Relaxation (Q1.4)

	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`

Give all new possible sets of values if we relax Write → Write ordering and the instruction orderings that caused them.

Operation Order	P1.x2	P1.x3	P2.x2

W→W Relaxation (Q1.4)

	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`

Give all new possible sets of values if we relax Write → Write ordering and the instruction orderings that caused them.

Operation Order	P1.x2	P1.x3	P2.x2
J2 J3 I1 I2 I3 J1	2	0	0

R→R and R→W Relaxation (Q1.5)

	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`

Give all new possible sets of values if we relax Read → Read and Read → Write ordering constraints and the instruction orderings that caused them.

Operation Order	P1.x2	P1.x3	P2.x2

R→R and R→W Relaxation (Q1.5)

	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`

Give all new possible sets of values if we relax Read → Read and Read → Write ordering constraints and the instruction orderings that caused them.

Operation Order	P1.x2	P1.x3	P2.x2
J3 I1 I2 I3 J1 J2	2	0	1
I2 J1 J3 I1 I3 J2	2	2	1

Using Fences to Constrain Memory Ordering

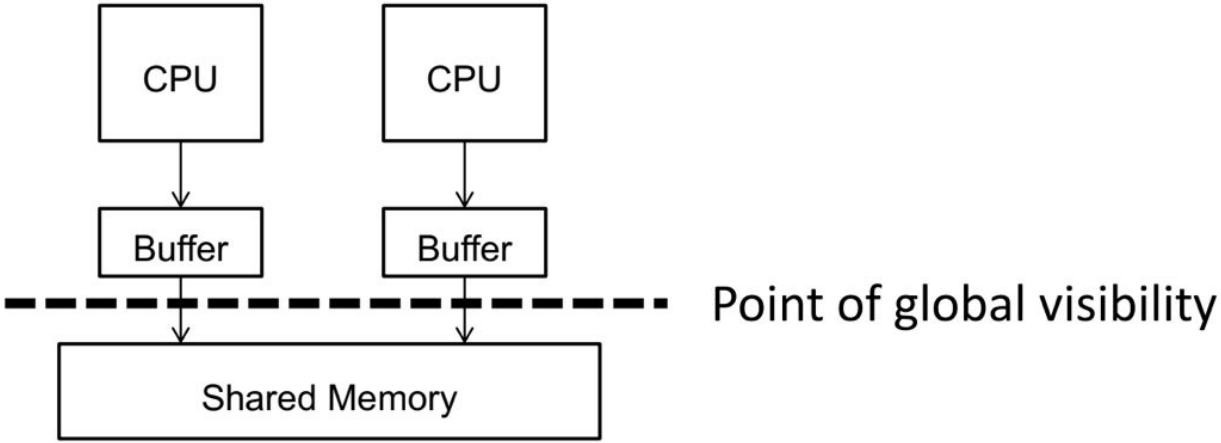
The RISC-V `FENCE` instruction comes in several fine-grained variants:

1. Write → Read `FENCE w,r`
2. Write → Write `FENCE w,w`
3. Read → Write `FENCE r,w`
4. Read → Read `FENCE r,r`

Can combine constraints: `FENCE r,rw == FENCE r,r; FENCE r,w`

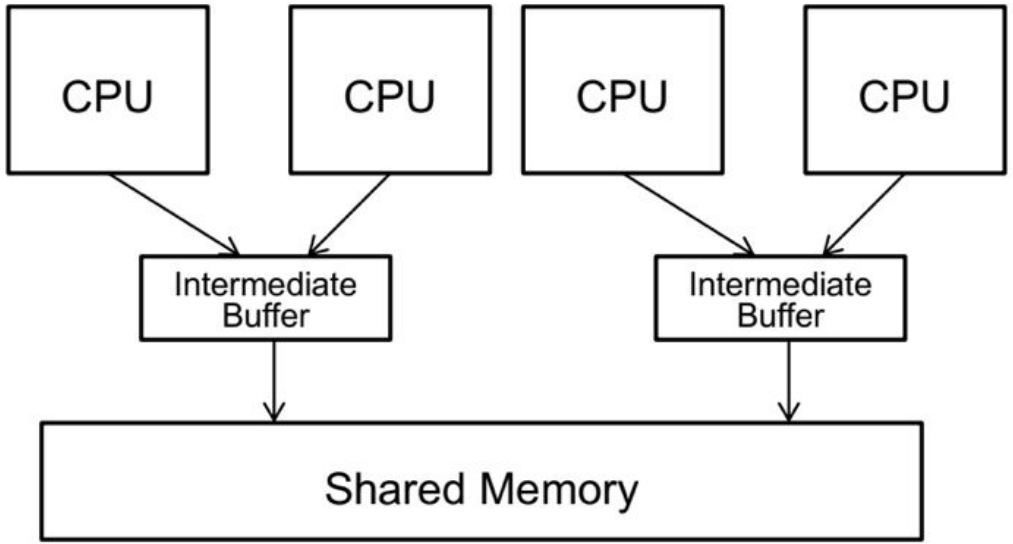
`FENCE` without parameters is a full barrier: `FENCE rw,rw`

Multi-Copy Atomicity



Multi-copy atomic -> once one processor’s write becomes visible to another processor, it becomes visible to all processors

Non-Multi-Copy Atomic



A system where a write by one processor becomes visible to a subset of other processors before it becomes visible to all processors

Example: Hierarchical shared buffers

Worksheet Q2

Q2: True/False

Indicate whether the following statements are true or false:

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

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

Worksheet Q2

Q2: True/False

Indicate whether the following statements are true or false:

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

False – The memory system is also responsible for enforcing sequential consistency, not only the core pipelines. Even with in-order issue and cache coherence, reordering of memory operations can arise from write buffers with bypassing, write-through and non-blocking caches, and a non-shared-bus interconnect fabric to separate memory banks.

Worksheet Q2

Q2: True/False

Indicate whether the following statements are true or false:

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

True – In the extreme case, the compiler can conservatively insert full fences between all memory operations to enforce strict program order within each thread. A more practical approach is to emit fences only as necessary or create locks implicitly around accesses to variables that the programmer identifies as being shared, typically with a keyword or another language feature. This guarantees SC executions for certain programs classified as data-race-free (all accesses to shared locations are properly synchronized).

Alternatively, false – It could be argued that local fences are *not* sufficient without assuming multi-copy atomicity. Whether SC can be implemented on top of a non-multi-copy-atomic memory model depends on the ISA providing global barriers to enforce ordering with respect to accesses in threads other than the thread issuing the barrier.

Worksheet Q2

Q2: True/False

Indicate whether the following statements are true or false:

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?

Worksheet Q2

Q2: True/False

Indicate whether the following statements are true or false:

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?

   Yes/True – As sequential consistency mandates multi-copy atomicity and is therefore a stronger form of memory consistency, SC executions are a proper subset of the execution outcomes permitted by the less restrictive non-multi-copy-atomic model. Software that is correctly written with the proper fences necessary to avoid data races under the weaker model would continue to execute correctly on this hardware implementation. The hardware implicitly enforces the same ordering constraints that fences would; the fence instructions would be treated by this implementation as NOPs.