ECE 454
Computer Systems Programming

Memory Consistency

Ashvin Goel, Ding Yuan
ECE Dept, University of Toronto

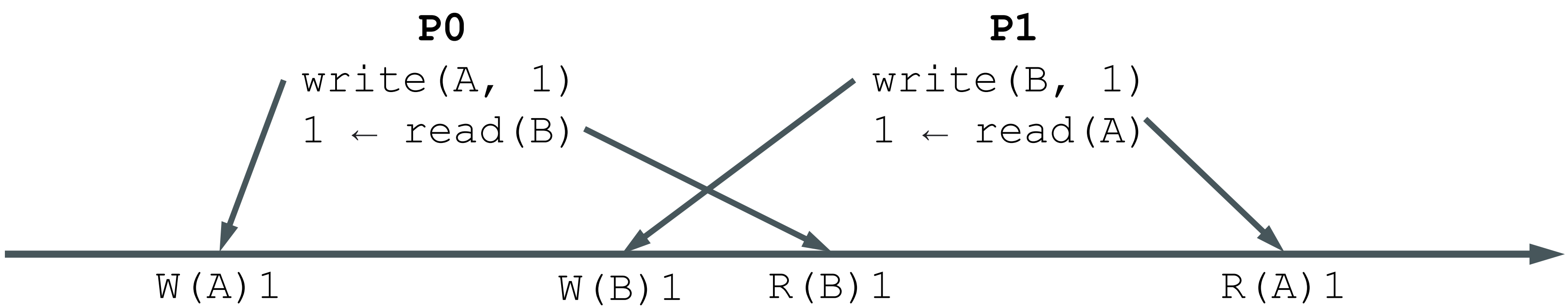
With thanks to Anton Burtsy, Paul E. McKenney

Coherence versus Consistency

- Recall cache coherence ensures that all processors have a consistent view of a single memory location (e.g., X)
  - All loads and stores to X can be put on a timeline (total order) that respects the program order of loads and stores of each processor
  - Defines memory behavior in the presence of processor caches
- Memory consistency defines the behavior of reads and writes by a processor to different locations (as observed by other processors)
  - Defines when writes propagate to other processors, what values reads can return (or cannot return), whether caches exist or not
  - Intuitively, reads should return value of last write
    - But how should last be defined?

Sequential Consistency

- A system is sequentially consistent if the result of any execution is the same as if all the memory operations were executed in some sequential order, and the memory operations of each processor are executed in program order



This model is intuitive to programmers, but not implemented by real processors, as we see next

Memory Ordering With Sequential Consistency

P0

```
void T0(void) {
    A = 1;
    read(B);
}
```

P1

```
void T1(void) {
    B = 1;
    read(A);
}
```

- With sequential consistency, can both reads return 0?
- Suppose this is possible (proof by contradiction):
  - Add edge between ops X and Y to indicate X happens before Y
    - 2 edges for program order
    - 2 edges for memory ordering dependency, why?
    - Happens-before edges form a cycle!
  - Would need time warp for both reads to return 0 ☺

Pros and Cons of Sequential Consistency

```
        P0                                  P1
void T0(void) {                     void T1(void) {
    A = 1;                              B = 1;
    read(B);                            read(A);
}                                   }
```

- Pros: an intuitive model of parallelism ☺
  - Each processor executes memory instructions in order
  - Memory ops from all processors appear sequentially ordered
- Cons: programs run terribly slowly ☹
  - Requires each memory operation to complete (results are visible) before proceeding with next memory operation in program order
  - Requires writes be visible in the same order at other processors

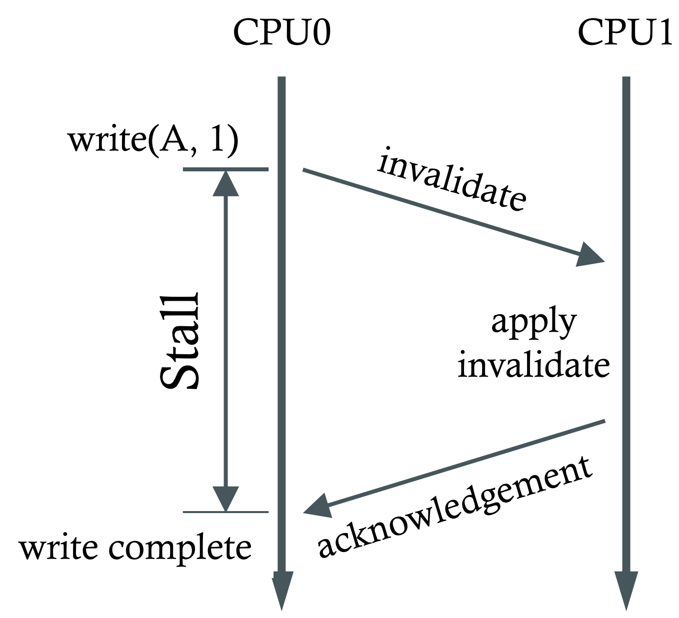
Memory Ordering on Real Processors

```
       P0                                 P1
void T0(void) {                    void T1(void) {
    A = 1;                             B = 1;
    read(B);                           read(A);
}                                  }
```

- With sequential consistency, can both reads return 0?
- But what happens on real processors?

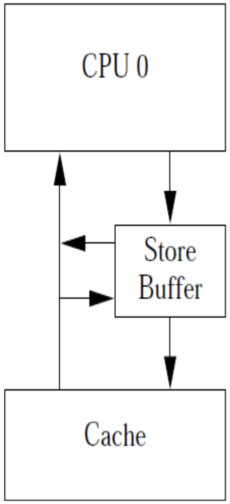
Understanding Write Completion

- Say write(A, 1) on CPU0 is a write miss
  - Cache coherence protocol sends an invalidate message to other CPUs to invalidate their cached copies of A
- Write completes only after CPU0 receives acknowledgment from CPU1
- Otherwise, another CPU could receive writes out-of-order, perform stale reads, etc.
- Problem: writes become slow



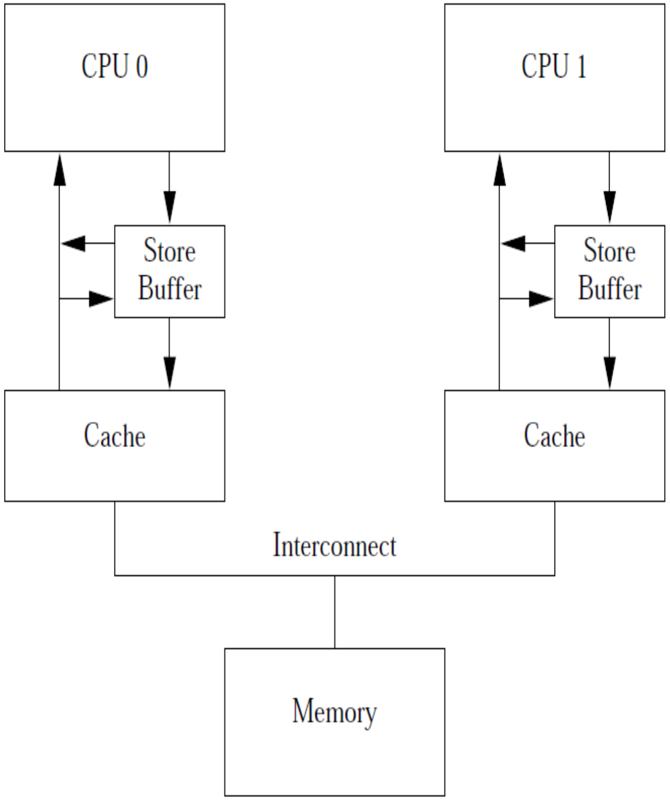
Processor Optimization: Store Buffers

- So, let's not wait for the write completion…
- Record a store in a CPU buffer
- Let CPU proceed immediately
- Causes no issues on uniprocessors



Processor Optimization: Store Buffers

- So, let's not wait for the write completion…
- Record a store in a CPU buffer
- Let CPU proceed immediately
- What about multiprocessors?
  - Send invalidate message, complete the store when invalidate message is acked, i.e., flush the store from the store buffer to the cache



Memory Ordering With Store Buffer

P0

```
void writer(void) {
    A = 1;
    B = 1;
}
```

P1

```
void reader(void) {
    while (B == 0)
        continue;
    assert(A == 1);
}
```

- Can the assert fail?
- Assert can fail on some processors ☹, let's look at why

Memory Ordering With Store Buffer

P0

```
A=[invalid], B=[excl, 0]
A = 1;
// save A in store buffer
// send invalidate(A)

B = 1;
// B in [excl],
// so update B in cache

// receive read(B)
// B in [shared,1]
// send read_reply(B, 1)
```

P1

```
A=[shared, 0], B=[invalid]
while (B == 0)
    // read(B)
    continue;

// receive read_reply(B, 1)
// exit while loop
assert(A == 1); // fails
// receive invalidate(A)
```

How can we fix this problem?

Memory Ordering With Store Buffer

P0

```
A=[invalid], B=[excl, 0]
A = 1;
// save A in store buffer
// send invalidate(A)

B = 1;
// B in [excl],
// so update B in cache
// DO NOT UPDATE CACHE UNTIL
// STORE BUFFER IS DRAINED
// receive read(B)
// B in [shared,0]
// send read_reply(B, 1)
```

P1

```
A=[shared, 0], B=[invalid]
while (B == 0)
    // read(B)
    continue;

// receive read_reply(B, 1)
// exit while loop
assert(A == 1);
```

Write Memory Barrier (wmb)

- smp_wmb()
  - Causes the CPU to flush its store buffer before applying subsequent stores to its cache lines
  - The CPU can either
    - Stall until the store buffer is empty before proceeding, or
    - It can use the store buffer to hold subsequent stores until all the prior entries in the buffer had been applied

Memory Ordering With Write Barrier

P0

```
void writer(void) {
    A = 1;
    smp_wmb();
    B = 1;
}
```

P1

```
void reader(void) {
    while (B == 0)
        continue;
    assert(A == 1);
}
```

- Assert will not fail ☺, let's look at why

Memory Ordering With Write Barrier

P0

```
A=[invalid], B=[excl, 0]
A = 1;
// save A in store buffer
// send invalidate(A)
smp_wmb();
// mark store buffer
B = 1;
// store buffer has marked entries,
// so save B in store buffer
// receive read(B)
// B in [shared,0]
// send read_reply(B, 0)
// receive invalidate_ack(A)
// flush store buffer
```

P1

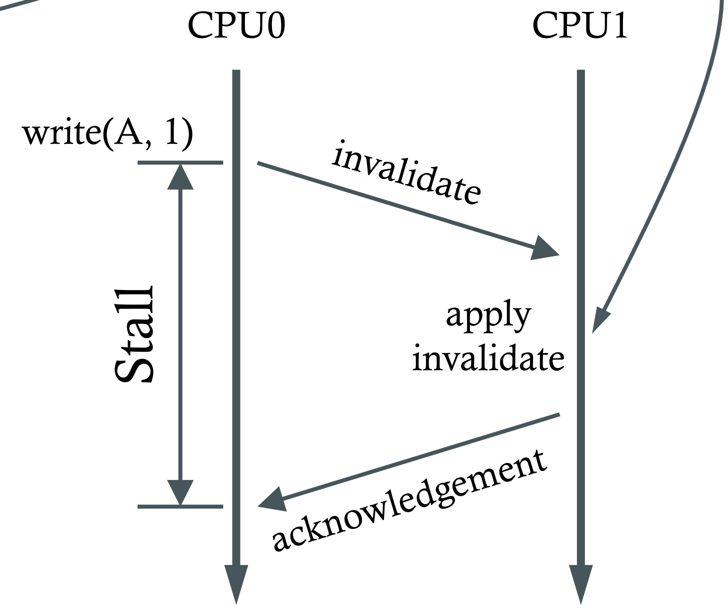
```
A=[shared, 0], B=[invalid]
while (B == 0)
    // read(B)
    continue;

// receive invalidate(A)
// send invalidate_ack(A)

// receive read_reply(B, 1)
// exit while loop
assert(A == 1); // succeeds
// read(A), reads 1
```

Understanding Invalidate Messages

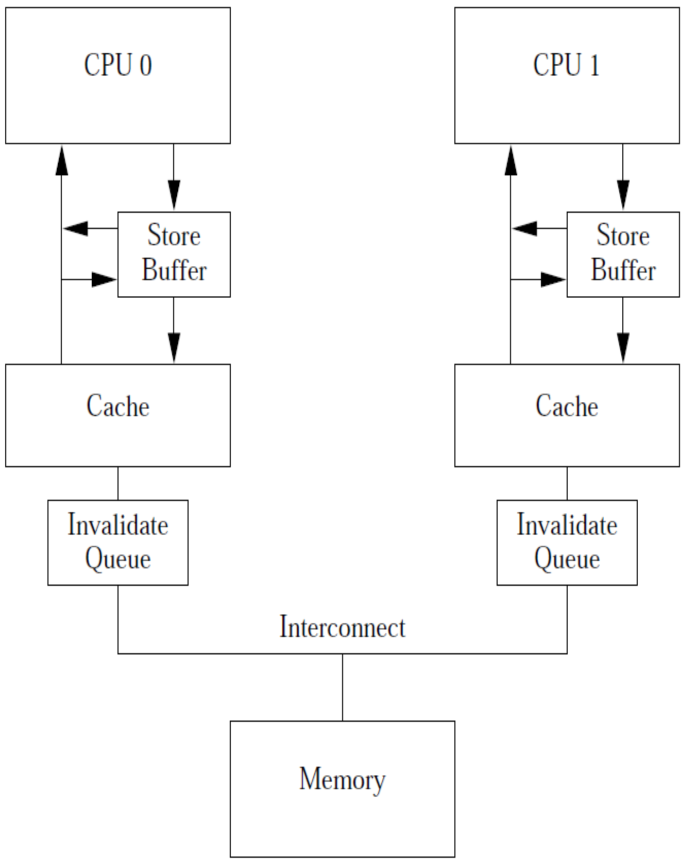
- Invalidate messages (and their response) can be slow
  - CPU1 cache could be overloaded, so it could respond slowly
- While waiting for invalidate acknowledgements, CPU0 can run out of space in store buffer, stalling execution



Processor Optimization: Invalidate Queues

- So, let's not wait to invalidate the cache…

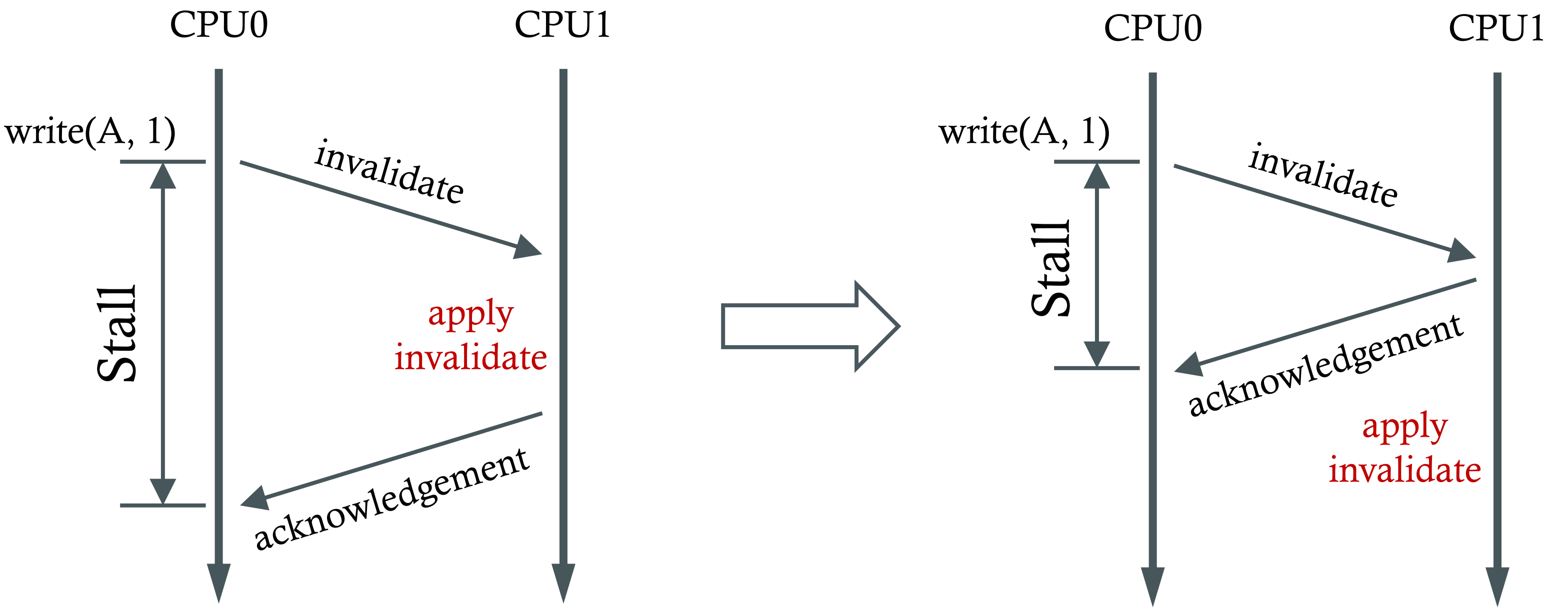
- Receive side
  - Stores invalidate request in a queue
  - Acknowledges invalidate right away
  - Applies invalidate later



Invalidate Processing

Write invalidation

Write invalidation
with invalidate queue



Memory Ordering With Invalidate Queue

P0

```
A=[invalid], B=[excl, 0]
A = 1;
// save A in store buffer
// send invalidate(A)
smp_wmb();
// mark store buffer

// receive invalidate_ack(A)
// flush store buffer
B = 1;
// b in [excl], update B in cache
// receive read(B)
// B in [shared,1]
// send read_reply(B, 1)
```

P1

```
A=[shared, 0], B=[invalid]
while (B == 0)
    // read(B)
    continue;
// receive invalidate(A)
// queue invalidate(A)
// send invalidate_ack(A)

// receive read_reply(B, 1)
// exit while loop
assert(A == 1); // fails
```

How can we fix this problem?

Memory Ordering With Invalidate Queue

P0

```
A=[invalid], B=[excl, 0]
A = 1;
// save A in store buffer
// send invalidate(A)
smp_wmb();
// mark store buffer

// receive invalidate_ack(A)
// flush store buffer
B = 1;
// b in [excl], update B in cache
// receive read(B)
// B in [shared]
// send read_reply(B, 1)
```

P1

```
A=[shared, 0], B=[invalid]
while (B == 0)
    // read(B)
    continue;
// receive invalidate(A)
// queue invalidate(A)
// send invalidate_ack(A)

// receive read_reply(B, 1)
// exit while loop
// DRAIN INVALIDATE QUEUE
assert(A == 1);
```

Read Memory Barrier (rmb)

- smp_rmb()
- Marks all the entries currently in the processor's invalidate queue
- Forces any subsequent load to wait until all marked entries have been applied to the CPU's cache

Memory Ordering With Read & Write Barriers

P0

```
void writer(void) {
    A = 1;
    smp_wmb();
    B = 1;
}
```

P1

```
void reader(void) {
    while (B == 0)
        continue;
    smp_rmb();
    assert(A == 1);
}
```

- Assert will not fail ☺
- The two barriers ensure sequential consistency!

Memory Ordering Conclusions

- Sequential consistency model makes it easier to write parallel programs since it matches the programmer's mental model of parallel program execution
  - However, sequential consistency is expensive to implement
- Processors play games by buffering stores and delaying cache invalidations to get good performance
  - Reads and writes may appear to be performed out of order, and reads may return stale data
  - Programmers need to use memory barriers to ensure correct order of memory operations across CPUs
  - Only programming wizards need apply (as we will see next)!

Memory Consistency and Related Resources

- For an introduction to memory consistency models, see:
  https://www.cs.utexas.edu/~bornholt/post/memory-models.html

- For an excellent tutorial, see:
  Shared Memory Consistency Models: A Tutorial
  Sarita V. Adve, Kourosh Gharachorloo

- For an excellent (online) book, see:
  A Primer on Memory Consistency and Cache Coherence
  V. Nagarajan, et al

- Gory details about Linux memory barriers:
  https://bruceblinn.com/linuxinfo/MemoryBarriers.html
  https://www.kernel.org/doc/Documentation/memory-barriers.txt