Module 10: "Design of Shared Memory Multiprocessors"

Lecture 19: "Sequential Consistency and Cache Coherence Protocols"

- Memory consistency
- Consistency model
- Sequential consistency
- What is program order?
- OOO and SC
- SC example
- Implementing SC
- Write atomicity
- Summary of SC
- Back to shared bus
- Snoopy protocols
- Stores
- Invalidation vs. update
- Which one is better?
- MSI protocol
- State transition
- MSI protocol
- M to S, or M to I?
- MSI example
- MESI protocol
- State transition
- MESI protocol
- MESI example

Previous Next

Module 10: "Design of Shared Memory Multiprocessors"
Lecture 19: "Sequential Consistency and Cache Coherence Protocols"

Memory consistency

- Need a more formal description of memory ordering
  - How to establish the order between reads and writes from different processors to different variables?
- The most clear way is to use synchronization
  P0: A=1; flag=1
  P1: while (!flag); print A;
- Another example (assume A=0, B=0 initially)
  P0: A=1; print B;
  P1: B=1; print A;
  - What do you expect?
- Memory consistency model is a contract between programmer and hardware regarding memory ordering

Consistency model

- A multiprocessor normally advertises the supported memory consistency model
  - This essentially tells the programmer what the possible correct outcome of a program could be when run on that machine
  - Cache coherence deals with memory operations to the same location, but not different locations
  - Without a formally defined order across all memory operations it often becomes impossible to argue about what is correct and what is wrong in shared memory
- Various memory consistency models
  - Sequential consistency (SC) is the most intuitive one and we will focus on it now (more consistency models later)

Sequential consistency

- Total order achieved by interleaving accesses from different processors
- The accesses from the same processor are presented to the memory system in program order
- Essentially, behaves like a randomly moving switch connecting the processors to memory
  - Picks the next access from a randomly chosen processor
- Lamport's definition of SC
  - A multiprocessor 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

What is program order?

- Any legal re-ordering is allowed
- The program order is the order of instructions from a sequential piece of code where programmer's intuition is preserved
  - The order must produce the result a programmer expects
- Can out-of-order execution violate program order?
  - No. All microprocessors commit instructions in-order and that is where the state

becomes visible
    - For modern microprocessors the program order is really the commit order
- Can out-of-order (OOO) execution violate SC?
    - Yes. Need extra logic to support SC on top of OOO

Previous Next

OOO and SC

- Consider a simple example (all are zero initially)

  P0: x=w+1; r=y+1;

  P1: y=2; w=y+1;

  - Suppose the load that reads w takes a miss and so w is not ready for a long time; therefore, x=w+1 cannot complete immediately; eventually w returns with value 3
  - Inside the microprocessor r=y+1 completes (but does not commit) before x=w+1 and gets the old value of y (possibly from cache); eventually instructions commit in order with x=4, r=1, y=2, w=3
  - So we have the following partial orders

    P0: x=w+1 < r=y+1 and P1: y=2 < w=y+1

    Cross-thread: w=y+1 < x=w+1 and r=y+1 < y=2

  - Combine these to get a contradictory total order
- What went wrong?

SC example

- Consider the following example

  P0: A=1; print B;

  P1: B=1; print A;

- Possible outcomes for an SC machine
  - (A, B) = (0,1); interleaving: B=1; print A; A=1; print B
  - (A, B) = (1,0); interleaving: A=1; print B; B=1; print A
  - (A, B) = (1,1); interleaving: A=1; B=1; print A; print B
    A=1; B=1; print B; print A
  - (A, B) = (0,0) is impossible: read of A must occur before write of A and read of B must occur before write of B i.e. print A < A=1 and print B < B=1, but A=1 < print B and B=1 < print A; thus print B < B=1 < print A < A=1 < print B which implies print B < print B, a contradiction

Implementing SC

- Two basic requirements
  - Memory operations issued by a processor must become visible to others in program order
  - Need to make sure that all processors see the same total order of memory operations: in the previous example for the (0,1) case both P0 and P1 should see the same interleaving: B=1; print A; A=1; print B
- The tricky part is to make sure that writes become visible in the same order to all processors
  - Write atomicity: as if each write is an atomic operation
  - Otherwise, two processors may end up using different values (which may still be

correct from the viewpoint of cache coherence, but will violate SC)

Write atomicity

- Example (A=0, B=0 initially)

  P0: A=1;

  P1: while (!A); B=1;

  P2: while (!B); print A;

- A correct execution on an SC machine should print A=1
  - A=0 will be printed only if write to A is not visible to P2, but clearly it is visible to P1 since it came out of the loop
  - Thus A=0 is possible if P1 sees the order A=1 < B=1 and P2 sees the order B=1 < A=1 i.e. from the viewpoint of the whole system the write A=1 was not “atomic”
  - Without write atomicity P2 may proceed to print 0 with a stale value from its cache

Summary of SC

- Program order from each processor creates a partial order among memory operations
- Interleaving of these partial orders defines a total order
- Sequential consistency: one of many total orders
- A multiprocessor is said to be SC if any execution on this machine is SC compliant
- Sufficient but not necessary conditions for SC
  - Issue memory operation in program order
  - Every processor waits for write to complete before issuing the next operation
  - Every processor waits for read to complete and the write that affects the returned value to complete before issuing the next operation (important for write atomicity)

Previous Next

Back to shared bus

- Centralized shared bus makes it easy to support SC
  - Writes and reads are all serialized in a total order through the bus transaction ordering
  - If a read gets a value of a previous write, that write is guaranteed to be complete because that bus transaction is complete
  - The write order seen by all processors is the same in a write through system because every write causes a transaction and hence is visible to all in the same order
  - In a nutshell, every processor sees the same total bus order for all memory operations and therefore any bus-based SMP with write through caches is SC
- What about a multiprocessor with writeback cache?
  - No SMP uses write through protocol due to high BW

Snoopy protocols

- No change to processor or cache
  - Just extend the cache controller with snoop logic and exploit the bus
- We will focus on writeback caches only
  - Possible states of a cache line: Invalid (I), Shared (S), Modified or dirty (M), Clean exclusive (E), Owned (O); every processor does not support all five states
  - E state is equivalent to M in the sense that the line has permission to write, but in E state the line is not yet modified and the copy in memory is the same as in cache; if someone else requests the line the memory will provide the line
  - O state is exactly same as E state but in this case memory is not responsible for servicing requests to the line; the owner must supply the line (just as in M state)
  - Stores really read the memory (as opposed to write)

Stores

- Look at stores a little more closely
  - There are three situations at the time a store issues: the line is not in the cache, the line is in the cache in S state, the line is in the cache in one of M, E and O states
  - If the line is in I state, the store generates a read-exclusive request on the bus and gets the line in M state
  - If the line is in S or O state, that means the processor only has read permission for that line; the store generates an upgrade request on the bus and the upgrade acknowledgment gives it the write permission (this is a data-less transaction)
  - If the line is in M or E state, no bus transaction is generated; the cache already has write permission for the line (this is the case of a write hit; previous two are write misses)

Invalidation vs. update

- Two main classes of protocols:
  - Invalidation-based and update-based
  - Dictates what action should be taken on a write
  - Invalidation-based protocols invalidate sharers when a write miss (upgrade or readX) appears on the bus
  - Update-based protocols update the sharer caches with new value on a write: requires

write transactions (carrying just the modified bytes) on the bus even on write hits (not very attractive with writeback caches)
- Advantage of update-based protocols: sharers continue to hit in the cache while in invalidation-based protocols sharers will miss next time they try to access the line
- Advantage of invalidation-based protocols: only write misses go on bus (suited for writeback caches) and subsequent stores to the same line are cache hits

Which one is better?

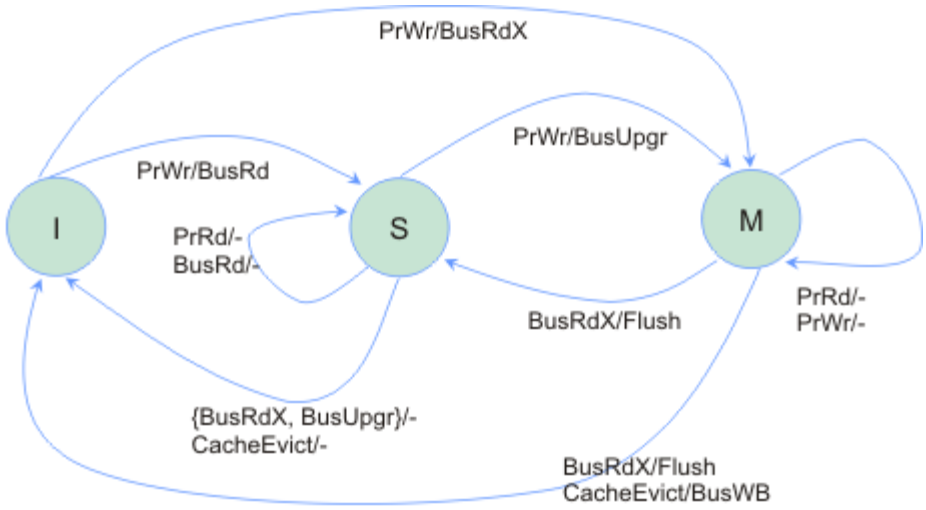
- Difficult to answer
  - Depends on program behavior and hardware cost
- When is update-based protocol good?
  - What sharing pattern? (large-scale producer/consumer)
  - Otherwise it would just waste bus bandwidth doing useless updates
- When is invalidation-protocol good?
  - Sequence of multiple writes to a cache line
  - Saves intermediate write transactions
- Also think about the overhead of initiating small updates for every write in update protocols
  - Invalidation-based protocols are much more popular
  - Some systems support both or maybe some hybrid based on dynamic sharing pattern of a cache line

Previous Next

MSI protocol

- Forms the foundation of invalidation-based writeback protocols
  - Assumes only three supported cache line states: I, S, and M
  - There may be multiple processors caching a line in S state
  - There must be exactly one processor caching a line in M state and it is the owner of the line
  - If none of the caches have the line, memory must have the most up-to-date copy of the line
- Processor requests to cache: PrRd, PrWr
- Bus transactions: BusRd, BusRdX, BusUpgr, BusWB

State transition



MSI protocol

- Few things to note
  - Flush operation essentially launches the line on the bus
  - Processor with the cache line in M state is responsible for flushing the line on bus whenever there is a BusRd or BusRdX transaction generated by some other processor
  - On BusRd the line transitions from M to S, but not M to I. Why? Also at this point both the requester and memory pick up the line from the bus; the requester puts the line in its cache in S state while memory writes the line back. Why does memory need to write back?
  - On BusRdX the line transitions from M to I and this time memory does not need to pick up the line from bus. Only the requester picks up the line and puts it in M state in its cache. Why?

M to S, or M to I?

- BusRd takes a cache line in M state to S state
  - The assumption here is that the processor will read it soon, so save a cache miss by going to S
  - May not be good if the sharing pattern is migratory: P0 reads and writes cache line A,

then P1 reads and writes cache line A, then P2…

- For migratory patterns it makes sense to go to I state so that a future invalidation is saved
- But for bus-based SMPs it does not matter much because an upgrade transaction will be launched anyway by the next writer, unless there is special hardware support to avoid that: how?
- The big problem is that the sharing pattern for a cache line may change dynamically: adaptive protocols are good and are supported by Sequent Symmetry and MIT Alewife

MSI example

- Take the following example
  - P0 reads x, P1 reads x, P1 writes x, P0 reads x, P2 reads x, P3 writes x
  - Assume the state of the cache line containing the address of x is I in all processors

    P0 generates BusRd, memory provides line, P0 puts line in S state

    P1 generates BusRd, memory provides line, P1 puts line in S state

    P1 generates BusUpgr, P0 snoops and invalidates line, memory does not respond, P1 sets state of line to M

    P0 generates BusRd, P1 flushes line and goes to S state, P0 puts line in S state, memory writes back

    P2 generates BusRd, memory provides line, P2 puts line in S state

    P3 generates BusRdX, P0, P1, P2 snoop and invalidate, memory provides line, P3 puts line in cache in M state

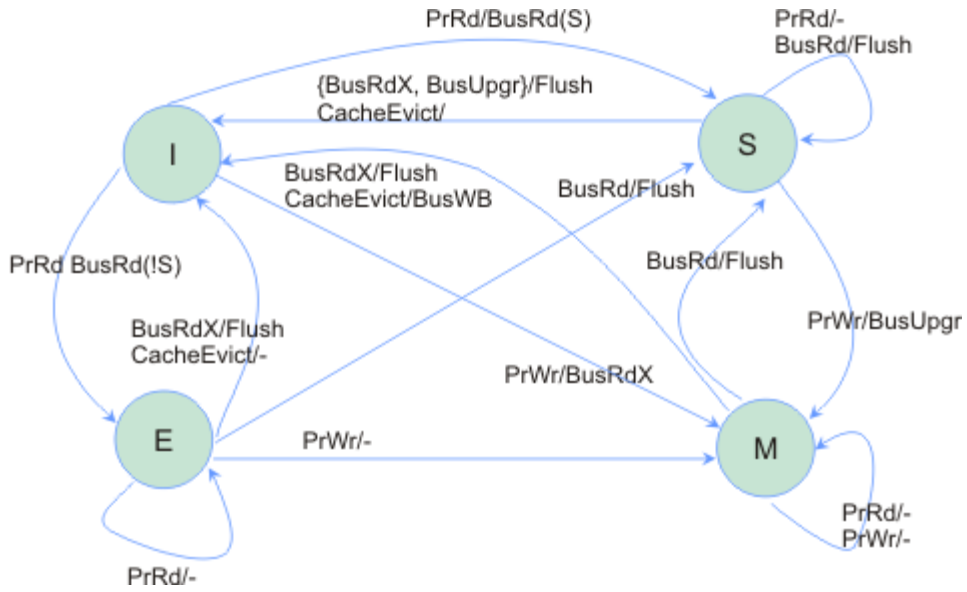
Previous Next

Module 10: "Design of Shared Memory Multiprocessors"
Lecture 19: "Sequential Consistency and Cache Coherence Protocols"

MESI protocol

- The most popular invalidation-based protocol e.g., appears in Intel Xeon MP
- Why need E state?
  - The MSI protocol requires two transactions to go from I to M even if there is no intervening requests for the line: BusRd followed by BusUpgr
  - We can save one transaction by having memory controller respond to the first BusRd with E state if there is no other sharer in the system
  - How to know if there is no other sharer? Needs a dedicated control wire that gets asserted by a sharer (wired OR)
  - Processor can write to a line in E state silently and take it to M state

State transition



MESI protocol

- If a cache line is in M state definitely the processor with the line is responsible for flushing it on the next BusRd or BusRdX transaction
- If a line is not in M state who is responsible?
  - Memory or other caches in S or E state?
  - Original Illinois MESI protocol assumed cache-to-cache transfer i.e. any processor in E or S state is responsible for flushing the line
  - However, it requires some expensive hardware, namely, if multiple processors are caching the line in S state who flushes it? Also, memory needs to wait to know if it should source the line
  - Without cache-to-cache sharing memory always sources the line unless it is in M state

MESI example

- Take the following example
  - P0 reads x, P0 writes x, P1 reads x, P1 writes x, …

    P0 generates BusRd, memory provides line, P0 puts line in cache in E state

P0 does write silently, goes to M state

P1 generates BusRd, P0 provides line, P1 puts line in cache in S state, P0 transitions to S state
Rest is identical to MSI

- Consider this example: P0 reads x, P1 reads x, …

  P0 generates BusRd, memory provides line, P0 puts line in cache in E state

  P1 generates BusRd, memory provides line, P1 puts line in cache in S state, P0 transitions to S state (no cache-to-cache sharing)
  Rest is same as MSI

Previous Next