Module 9: "Introduction to Shared Memory Multiprocessors"

Lecture 17: "Introduction to Cache Coherence Protocols"

- Invalidation vs. Update
- Sharing patterns
- Migratory hand-off
- States of a cache line
- Stores
- MSI protocol
- State transition
- MSI example
- MESI protocol
- State transition
- MESI example
- MOESI protocol
- Hybrid inval+update

Invalidation vs. Update

- Two main classes of protocols:
  - Dictates what action should be taken on a store
  - Invalidation-based protocols invalidate sharers when a store miss appears
  - Update-based protocols update the sharer caches with new value on a store
  - 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 store misses go on bus and subsequent stores to the same line are cache hits
- 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
- Overhead of initiating small updates
  - Invalidation-based protocols are much more popular
  - Some systems support both or maybe some hybrid based on dynamic sharing pattern of a cache line

Sharing patterns

- Producer-consumer (initially flag, done are zero)
  T0: while (!exit) {x=y; flag=1; while (done != k); flag=0; done=0;}
  T1 to Tk: while (!exit) {while (!flag); use x; done++; while (flag);}
  - Exit condition not shown
  - What if T1 to Tk do not have the outer loop?
- Migratory (initially flag is zero)
  T0: x = f0(x); flag++;
  T1 to Tk: while (flag != pid); x = f1(x); flag++;
  - Migratory hand-off?

Migratory hand-off

- Needs a memory writeback on every hand-off
  - r0, w0, r1, w1, r2, w2, r3, w3, r4, w4, …
  - How to avoid these unnecessary writebacks?
  - Saves memory bandwidth
  - Solution: add an owner state (different from M) in caches
  - Only owner can write a line back on eviction
  - Ownership shifts along the migratory chain

Previous Next

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

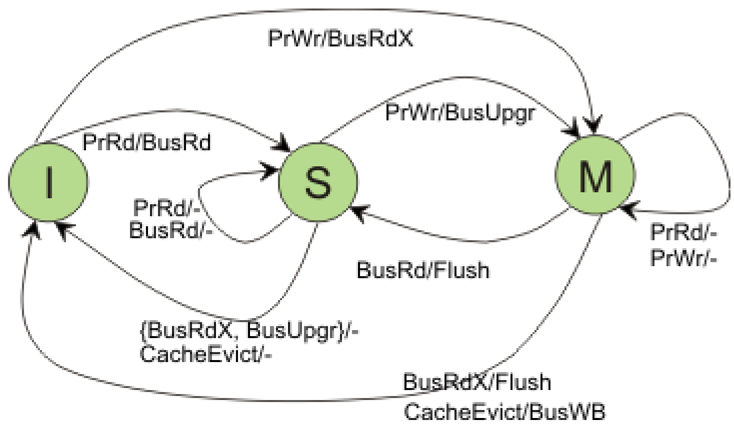
- Look at stores a little more closely
  - There are three situations at the time a store issues: the line is not in the cache, the line is in the cache in S state, the line is in the cache in one of M, E and O states
  - If the line is in I state, the store generates a read-exclusive request on the 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)

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

Previous Next

State transition



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

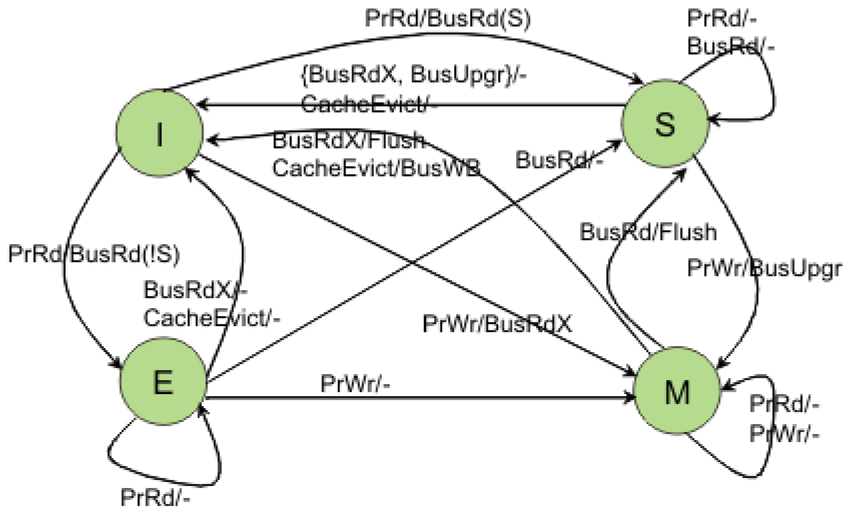
    P3 generates BusRdX, P0, P1, P2 snoop and invalidate, memory provides line, P3 puts line in cache in M state

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
  - Save one transaction by having memory controller respond to the first BusRd with E state if there is no other sharer in the system
  - Needs a dedicated control wire that gets asserted by a sharer (wired OR)
  - Processor can write to a line in E state silently

Previous Next

State transition



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

MOESI protocol

- State transitions pertaining to O state
  - I to O: not possible
  - E to O or S to O: not possible
  - M to O: on a BusRd/Flush (but no memory writeback)
  - O to I: on CacheEvict/BusWB or {BusRdX,BusUpgr}/Flush
  - O to S: not possible
  - O to E: not possible
  - O to M: on PrWr/BusUpgr
- At most one cache can have a line in O state at any point in time
- Two main design choices for MOESI
  - Consider the example: P0 reads x, P0 writes x, P1 reads x, P2 reads x, P3 reads x, …
  - When P1 launches BusRd, P0 sources the line and now the protocol has two options: 1. The line in P0 goes to O and the line in P1 is filled in state S; 2. The line in P0 goes to S and the line in P1 is filled in state O i.e. P1 inherits ownership from P0
  - For distributed shared memory, the second choice is better
  - According to the second choice, when P2 generates a BusRd request, P1 sources the line and transitions from O to S; P2 becomes the new owner
- Some SMPs do not support the E state
  - In many cases it is not helpful, only complicates the protocol
  - MOSI allows a compact state encoding in 2 bits
  - Sun WildFire uses MOSI protocol

Hybrid inval+update

- One possible hybrid protocol
  - Keep a counter per cache line and make Dragon update protocol the default
  - Every time the local processor accesses a cache line set its counter to some pre-defined threshold k
  - On each received update decrease the counter by one
  - When the counter reaches zero, the line is locally invalidated hoping that eventually the writer will switch to M state from Sm state when no sharers are left