The Lecture Contains:

Previous Next

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

Previous Next

Which One is Better?

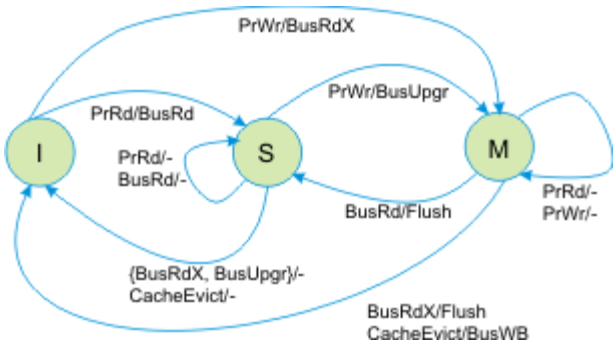
- Difficult to answer
  - Depends on program behavior and hardware cost
- When is update-based protocol good?
  - What sharing pattern? (large-scale producer/consumer)
  - Otherwise it would just waste bus bandwidth doing useless updates
- When is invalidation-protocol good?
  - Sequence of multiple 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

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

Previous Next

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?

Previous Next

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

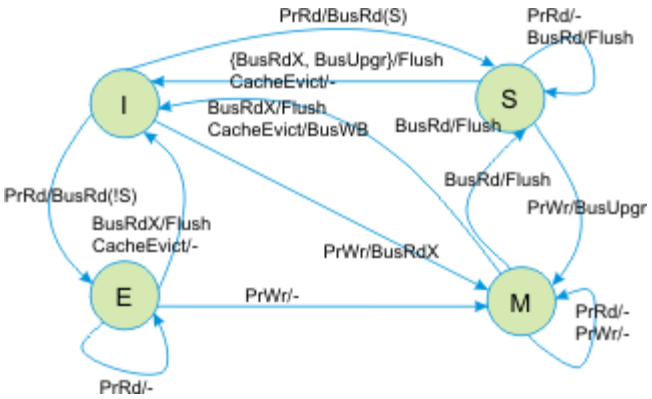
- Take the following example
  - P0 reads x, P1 reads x, P1 writes x, P0 reads x, P2 reads x, P3 writes x
  - Assume the state of the cache line containing the address of x is I in all processors

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

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



Previous Next

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

MOESI Protocol

- Some SMPs implement MOESI today e.g., AMD Athlon MP and the IBM servers
- Why is the O state needed?
  - O state is very similar to E state with four differences: 1. If a cache line is in O state in some cache, that cache is responsible for sourcing the line to the next requester; 2. The memory may not have the most up-to-date copy of the line (this implies 1); 3. Eviction of a line in O state generates a BusWB ; 4. Write to a line in O state must generate a bus transaction
  - When a line transitions from M to S it is necessary to write the line back to memory
  - For a migratory sharing pattern (frequent in database workloads) this leads to a series of writebacks to memory
  - These writebacks just keep the memory banks busy and consumes memory bandwidth
- Take the following example
  - P0 reads x, P0 writes x, P1 reads x, P1 writes x, P2 reads x, P2 writes x, …
  - Thus at the time of a BusRd response the memory will write the line back: one writeback per processor handover
  - O state aims at eliminating all these writebacks by transitioning from M to O instead of M to S on a BusRd /Flush
  - Subsequent BusRd requests are replied by the owner holding the line in O state
  - The line is written back only when the owner evicts it: one single writeback

Previous Next

MOESI Protocol

- State transitions pertaining to O state
  - I to O: not possible (or maybe; see below)
  - 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 (or maybe; next slide)
  - O to E: not possible (or maybe if silent eviction not allowed) '
  - 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 bus-based SMPs the two choices will yield roughly the same performance
  - For DSM multiprocessors we will revisit this issue if time permits
  - 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

MOSI Protocol

- 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

Previous Next