Module 10: "Design of Shared Memory Multiprocessors"

Lecture 20: "Performance of Coherence Protocols"

- MOESI protocol
- Dragon protocol
- State transition
- Dragon example
- Design issues
- General issues
- Evaluating protocols
- Protocol optimizations
- Cache size
- Cache line size
- Impact on bus traffic
- Large cache line
- Performance of update protocol
- Hybrid inval+update
- Update-based protocol
- Shared cache

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
- 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; see below)
  - 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
- 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

Dragon protocol

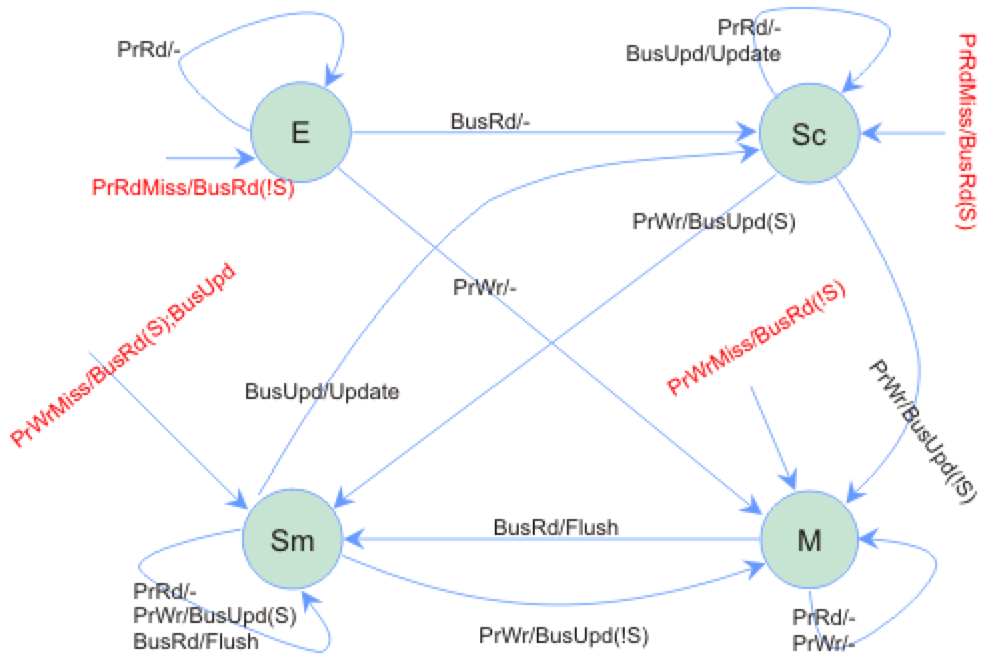
- An update-based protocol for writeback caches
- Four states: Two of them are standard E and M

- Shared clean (Sc): The standard S state
- Shared modified (Sm): This is really the O state

- In fact, five states because you always have I i.e. not in cache
- So really a MOESI update-based protocol
- New bus transaction: BusUpd
  - Used to update part of cache line
- Distinguish between cache hits and misses:
  - PrRd and PrWr are hits, PrRdMiss and PrWrMiss are misses

Previous Next

Design issues



Dragon example

- Take the following sequence
  - P0 reads x, P1 reads x, P1 writes x, P0 reads x, P2 reads x, P3 writes x
  - P0 generates BusRd, shared line remains low, it puts line in E state
  - P1 generates BusRd, shared line is asserted by P0, P1 puts line in Sc state, P0 also transitions to Sc state
  - P1 generates BusUpd, P0 asserts shared line, P1 takes the line to Sm state, P0 applies update but remains in Sc
  - P0 reads from cache, no state transition
  - P2 generates BusRd, P0 and P1 assert shared line, P1 sources the line on bus, P2 puts line in Sc state, P1 remains in Sm state, P0 remains in Sc state
  - P3 generates BusRd followed by BusUpd, P0, P1, P2 assert shared line, P1 sources the line on bus, P3 puts line in Sm state, line in P1 goes to Sc state, lines in P0 and P2 remain in Sc state, all processors update line

Design issues

- Can we eliminate the Sm state?
  - Yes. Provided on every BusUpd the memory is also updated; then the Sc state is sufficient (essentially boils down to a standard MSI update protocol)
  - However, update to cache may be faster than memory; but updating cache means occupying data banks during update thereby preventing the processor from accessing the cache, so not to degrade performance, extra cache ports may be needed
- Is it necessary to launch a bus transaction on an eviction of a line in Sc state?
  - May help if this was the last copy of line in Sc state
  - If there is a line in Sm state, it can go back to M and save subsequent unnecessary BusUpd transactions (the shared wire already solves this)

General issues

- Thus far we have assumed an atomic bus where transactions are not interleaved
    - In reality, high performance busses are pipelined and multiple transactions are in progress at the same time
    - How do you reason about coherence?
- Thus far we have assumed that the processor has only one level of cache
    - How to extend the coherence protocol to multiple levels of cache?
    - Normally, the cache coherence protocols we have discussed thus far executes only on the outermost level of cache hierarchy
    - A simpler but different protocol runs within the hierarchy to maintain coherence
- We will revisit these questions soon

Evaluating protocols

- In message passing machines the design of the message layer plays an important role
- Similarly, cache coherence protocols are central to the design of a shared memory multiprocessor
- The protocol performance depends on an array of parameters
- Experience and intuition help in determining good design points
- Otherwise designers use workload-driven simulations for cost/performance analysis
    - Goal is to decide where to spend money, time and energy
    - The simulators model the underlying multiprocessor in enough detail to capture correct performance trends as one explores the parameter space

Previous Next

Protocol optimizations

- MSI vs. MESI
  - Need to measure bus bandwidth consumption with and without E state because E state saves the otherwise S to M BusUpgr transactions
  - Turns out that the E state is not very helpful
  - The main reason is the E to M transition is rare; normally some other processor also reads the line before the write takes place (if at all)
- How important is BusUpgr?
  - Again need to look at bus bandwidth consumption with BusUpgr and with BusUpgr replaced by BusRdX
  - Turns out that BusUpgr helps
- Smaller caches demand more bus bandwidth
  - Especially when the primary working set does not fit in cache

Cache size

- With increasing problem size normally working set size also increases
  - More pressure on cache
- With increasing number of processors working set per processor goes down
  - Less pressure on cache
  - This effect sometimes leads to superlinear speedup i.e. on P processors you get speedup more than P
- Important to design the parallel program so that the critical working sets fit in cache
  - Otherwise bus bandwidth requirement may increase dramatically

Cache line size

- Uniprocessors have three C misses: cold, capacity, conflict
- Multiprocessors add two new types
  - True sharing miss: inherent in the algorithm e.g., P0 writes x and P1 uses x, so P1 will suffer from a true sharing miss when it reads x
  - False sharing miss: artifactual miss due to cache line size e.g. P0 writes x and P1 reads y, but x and y belong to the same cache line
- True and false sharing together form the communication or coherence misses in multiprocessors making it four C misses
- Technology is pushing for large cache line sizes, but…
- Increasing cache line size helps reduce
  - Cold misses if there is spatial locality
  - True sharing misses if the algorithm is properly structured to exploit spatial locality
- Increasing cache line size
  - Reduces the number of sets in a fixed-sized cache and may lead to more conflict misses
  - May increase the volume of false sharing
  - May increase miss penalty depending on the bus transfer algorithm (need to transfer more data per miss)
  - May fetch unnecessary data and waste bandwidth
- Note that true sharing misses will exist even with an infinite cache

- Impact of cache line size on true sharing heavily depends on application characteristics
    - Blocked matrix computations tend to have good spatial locality with shared data because they access data in small blocks thereby exploiting temporal as well as spatial locality
    - Nearest neighbor computations tend to have little spatial locality when accessing left and right border elements
- The exact proportion of various types of misses in an application normally changes with cache size, problem size and the number of processors
    - With small cache, capacity miss may dominate everything else
    - With large cache, true sharing misses may cause the major traffic

Impact on bus traffic

- When cache line size is increased it may seem that we bring in more data together and have better spatial locality and reuse
    - Should reduce bus traffic per unit computation
- However, bus traffic normally increases monotonically with cache line size
    - Unless we have enough spatial and temporal locality to exploit, bus traffic will increase
    - For most cases bus bandwidth requirement attains a minimum at a block size different from the minimum size; this is because at very small line sizes the overhead of communication becomes too high

Previous Next

Large cache line

- Large cache lines are intended to amortize the DRAM access and bus transfer latency over a large number of data points
- But false sharing becomes a problem
- Hardware solutions
    - Coherence at subblock level: divide the cache line into smaller blocks and maintain coherence for each of them; subblock invalidation on a write reduces chances of coherence misses even in the presence of false sharing
    - Delay invalidations: send invalidations only after the writer has completed several writes; but this directly impacts the write propagation model and hence leads to consistency models weaker than SC
    - Use update-based protocols instead of invalidation-based: probably not a good idea

Performance of update protocol

- Already discussed main trade-offs
- Consider running a sequential program on an SMP with update protocol
    - If the kernel decides to migrate the process to a different processor subsequent updates will go to caches that are never used: “pack-rat” phenomenon
- Possible designs that combine update and invalidation-based protocols
    - For each page, decide what type of protocol to run and make it part of the translation (i.e. hold it in TLB)
    - Otherwise dynamically detect for each cache line what protocol is good

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

Update-based protocol

- Update-based protocols tend to increase capacity misses slightly
    - Cache lines stay longer in the cache compared to an invalidation-based protocol; why?
- Update-based protocols can significantly reduce coherence misses
    - True sharing misses definitely go down
    - False sharing misses also decrease due to absence of invalidations
- But update-based protocols significantly increase bus bandwidth demand
    - This increases bus contention and delays other transactions
    - Possible to delay the updates by merging a number of them in a buffer

Shared cache

- Advantages
    - If there is only one level of cache no need for a coherence protocol

  - Very fine-grained sharing resulting in fast cross-thread communication
  - No false sharing
  - Smaller cache capacity requirement: overlapped working set
  - One processor’s fetched cache line can be used by others: prefetch effect
- Disadvantages
  - High cache bandwidth requirement and port contention
  - Destructive interference and conflict misses
- Will revisit this when discussing chip multiprocessing and hyper-threading

Previous Next