Multiprocessors on A Snoopy Bus

- Agenda
- Correctness goals
- A simple design
- Cache controller
- Snoop logic
- Writebacks
- A simple design
- Inherently non-atomic
- Write serialization
- Fetch deadlock
- Livelock
- Starvation
- More on LL/SC
- Multi-level caches

[From Chapter 6 of Culler, Singh, Gupta]

Previous Next

Lecture 24: "Write Serialization in a Simple Design"

Agenda

- Goal is to understand what influences the performance, cost and scalability of SMPs
  - Details of physical design of SMPs
  - At least three goals of any design: correctness, performance, low hardware complexity
  - Performance gains are normally achieved by pipelining memory transactions and having multiple outstanding requests
  - These performance optimizations occasionally introduce new protocol races involving transient states leading to correctness issues in terms of coherence and consistency

Correctness goals

- Must enforce coherence and write serialization
  - Recall that write serialization guarantees all writes to a location to be seen in the same order by all processors
- Must obey the target memory consistency model
  - If sequential consistency is the goal, the system must provide write atomicity and detect write completion correctly (write atomicity extends the definition of write serialization for any location i.e. it guarantees that positions of writes within the total order seen by all processors be the same)
- Must be free of deadlock, livelock and starvation
  - Starvation confined to a part of the system is not as problematic as deadlock and livelock
  - However, system-wide starvation leads to livelock

A simple design

- Start with a rather naïve design
  - Each processor has a single level of data and instruction caches
  - The cache allows exactly one outstanding miss at a time i.e. a cache miss request is blocked if already another is outstanding (this serializes all bus requests from a particular processor)
  - The bus is atomic i.e. it handles one request at a time

Cache controller

- Must be able to respond to bus transactions as necessary 1
  - Handled by the snoop logic
- The snoop logic should have access to the cache tags
  - A single set of tags cannot allow concurrent accesses by the processor-side and the bus-side controllers
  - When the snoop logic accesses the tags the processor must remain locked out from accessing the tags
  - Possible enhancements: two read ports in the tag RAM allows concurrent reads; duplicate copies are also possible; multiple banks reduce the contention also
  - In all cases, updates to tags must still be atomic or must be applied to both copies in case of duplicate tags; however, tag updates are a lot less frequent compared to reads

Previous Next

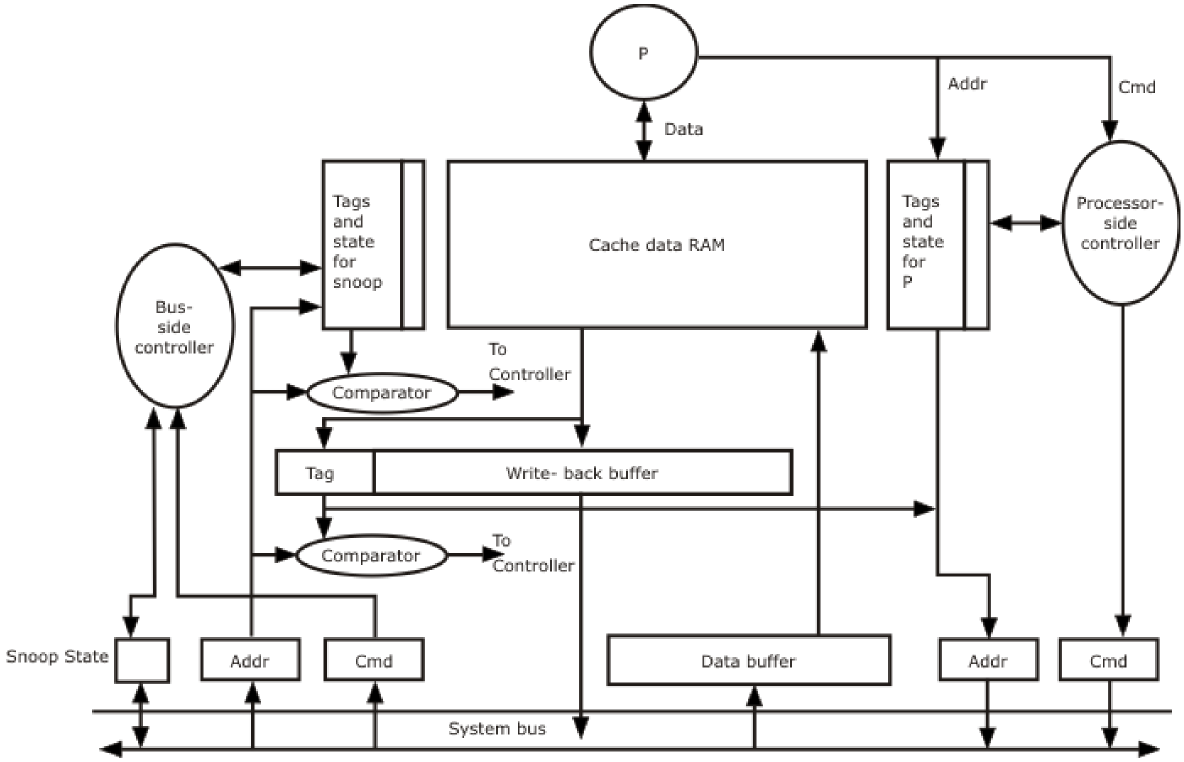
Snoop logic

- Couple of decisions need to be taken while designing the snoop logic
  - How long should the snoop decision take?
  - How should processors convey the snoop decision?
- Snoop latency (three design choices)
  - Possible to set an upper bound in terms of number of cycles; advantage: no change in memory controller hardware; disadvantage: potentially large snoop latency (Pentium Pro, Sun Enterprise servers)
  - The memory controller samples the snoop results every cycle until all caches have completed snoop (SGI Challenge uses this approach where the memory controller fetches the line from memory, but stalls if all caches haven't yet snooped)
  - Maintain a bit per memory line to indicate if it is in M state in some cache
- Conveying snoop result
  - For MESI the bus is augmented with three wired-OR snoop result lines (shared, modified, valid); the valid line is active low
  - The original Illinois MESI protocol requires cache-to-cache transfer even when the line is in S state; this may complicate the hardware enormously due to the involved priority mechanism
  - Commercial MESI protocols normally allow cache-to-cache sharing only for lines in M state
  - SGI Challenge and Sun Enterprise allow cache-to-cache transfers only in M state; Challenge updates memory when going from M to S while Enterprise exercises a MOESI protocol

Writebacks

- Writebacks are essentially eviction of modified lines
  - Caused by a miss mapping to the same cache index
  - Needs two bus transactions: one for the miss and one for the writeback
  - Definitely the miss should be given first priority since this directly impacts forward progress of the program
  - Need a writeback buffer (WBB) to hold the evicted line until the bus can be acquired for the second time by this cache
  - In the meantime a new request from another processor may be launched for the evicted line: the evicting cache must provide the line from the WBB and cancel the pending writeback (need an address comparator with WBB)

A simple design



Previous Next

Inherently non-atomic

- Even though the bus is atomic, a complete protocol transaction involves quite a few steps which together forms a non-atomic transaction
    - Issuing processor request
    - Looking up cache tags
    - Arbitrating for bus
    - Snoop action in other cache controller
    - Refill in requesting cache controller at the end
- Different requests from different processors may be in a different phase of a transaction
    - This makes a protocol transition inherently non-atomic
- Consider an example
    - P0 and P1 have cache line C in shared state
    - Both proceed to write the line
    - Both cache controllers look up the tags, put a BusUpgr into the bus request queue, and start arbitrating for the bus
    - P1 gets the bus first and launches its BusUpgr
    - P0 observes the BusUpgr and now it must invalidate C in its cache and change the request type to BusRdX
    - So every cache controller needs to do an associative lookup of the snoop address against its pending request queue and depending on the request type take appropriate actions
- One way to reason about the correctness is to introduce transient states
    - Possible to think of the last problem as the line C being in a transient S M state
    - On observing a BusUpgr or BusRdX, this state transitions to I M which is also transient
    - The line C goes to stable M state only after the transaction completes
    - These transient states are not really encoded in the state bits of a cache line because at any point in time there will be a small number of outstanding requests from a particular processor (today the maximum I know of is 16)
    - These states are really determined by the state of an outstanding line and the state of the cache controller

Write serialization

- Atomic bus makes it rather easy, but optimizations are possible
    - Consider a processor write to a shared cache line
    - Is it safe to continue with the write and change the state to M even before the bus transaction is complete?
    - After the bus transaction is launched it is totally safe because the bus is atomic and hence the position of the write is committed in the total order; therefore no need to wait any further (note that the exact point in time when the other caches invalidate the line is not important)
    - If the processor decides to proceed even before the bus transaction is launched (very much possible in ooo execution), the cache controller must take the responsibility of squashing and re-executing offending instructions so that the total order is consistent across the system

Fetch deadlock

- Just a fancy name for a pretty intuitive deadlock
  - Suppose P0's cache controller is waiting to get the bus for launching a BusRdX to cache line A
  - P1 has a modified copy of cache line A
  - P1 has launched a BusRd to cache line B and awaiting completion
  - P0 has a modified copy of cache line B
  - If both keep on waiting without responding to snoop requests, the deadlock cycle is pretty obvious
  - So every controller must continue to respond to snoop requests while waiting for the bus for its own requests
  - Normally the cache controller is designed as two separate independent logic units, namely, the inbound unit (handles snoop requests) and the outbound unit (handles own requests and arbitrates for bus)

Previous Next

Livelock

- Consider the following example
    - P0 and P1 try to write to the same cache line
    - P0 gets exclusive ownership, fills the line in cache and notifies the load/store unit (or retirement unit) to retry the store
    - While all these are happening P1's request appears on the bus and P0's cache controller modifies tag state to I before the store could retry
    - This can easily lead to a livelock
    - Normally this is avoided by giving the load/store unit higher priority for tag access (i.e. the snoop logic cannot modify the tag arrays when there is a processor access pending in the same clock cycle)
    - This is even rarer in multi-level cache hierarchy (more later)

Starvation

- Some amount of fairness is necessary in the bus arbiter
    - An FCFS policy is possible for granting bus, but that needs some buffering in the arbiter to hold already placed requests
    - Most machines implement an aging scheme which keeps track of the number of times a particular request is denied and when the count crosses a threshold that request becomes the highest priority (this too needs some storage)

More on LL/SC

- We have seen that both LL and SC may suffer from cache misses (a read followed by an upgrade miss)
- Is it possible to save one transaction?
    - What if I design my cache controller in such a way that it can recognize LL instructions and launch a BusRdX instead of BusRd?
    - This is called Read-for-Ownership (RFO); also used by Intel atomic xchg instruction
    - Nice idea, but you have to be careful
    - By doing this you have just enormously increased the probability of a livelock: before the SC executes there is a high probability that another LL will take away the line
    - Possible solution is to buffer incoming snoop requests until the SC completes (buffer space is proportional to P); may introduce new deadlock cycles (especially for modern non-atomic busses)

Multi-level caches

- We have talked about multi-level caches and the involved inclusion property
- Multiprocessors create new problems related to multi-level caches
    - A bus snoop result may be relevant to inner levels of cache e.g., bus transactions are not visible to the first level cache controller
    - Similarly, modifications made in the first level cache may not be visible to the second level cache controller which is responsible for handling bus requests
- Inclusion property makes it easier to maintain coherence
    - Since L1 cache is a subset of L2 cache a snoop miss in L2 cache need not be sent to L1 cache

Previous Next