Module 6: Shared Memory Multiprocessors: Consistency and Coherence

Lecture 11: Introduction to Snoopy Coherence

The Lecture Contains:

Previous Next

Snoopy Protocols

- Cache coherence protocols implemented in bus-based machines are called snoopy protocols
  - The processors snoop or monitor the bus and take appropriate protocol actions based on snoop results
  - Cache controller now receives requests both from processor and bus
  - Since cache state is maintained on a per line basis that also dictates the coherence granularity
  - Cannot normally take a coherence action on parts of a cache line
  - The coherence protocol is implemented as a finite state machine on a per cache line basis
  - The snoop logic in each processor grabs the address from the bus and decides if any action should be taken on the cache line containing that address (only if the line is in cache)

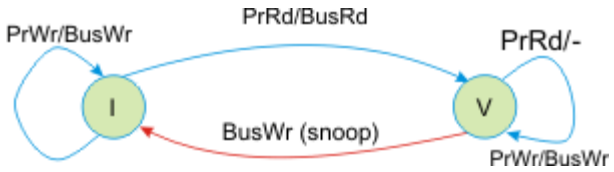
Write Through Caches

- There are only two cache line states
  - Invalid (I): not in cache
  - Valid (V): present in cache, may be present in other caches also
- Read access to a cache line in I state generates a BusRd request on the bus
  - Memory controller responds to the request and after reading from memory launches the line on the bus
  - Requester matches the address and picks up the line from the bus and fills the cache in V state
  - A store to a line always generates a BusWr transaction on the bus (since write through); other sharers either invalidate the line in their caches or update the line with new value

Previous Next

State Transition

- The finite state machine for each cache line:



- On a write miss no line is allocated
  - The state remains at I: called write through write no-allocated
- A/B means: A is generated by processor, B is the resulting bus transaction (if any)
- Changes for write through write allocate?

Ordering Memory op

- Assume that the bus is atomic
  - It takes up the next transaction only after finishing the previous one
- Read misses and writes appear on the bus and hence are visible to all processors
- What about read hits?
  - They take place transparently in the cache
  - But they are correct as long as they are correctly ordered with respect to writes
  - And all writes appear on the bus and hence are visible immediately in the presence of an atomic bus
- In general, in between writes reads can happen in any order without violating coherence
  - Writes establish a partial order

Previous Next

Write Through is Bad

- High bandwidth requirement
  - Every write appears on the bus
  - Assume a 3 GHz processor running application with 10% store instructions, assume CPI of 1
  - If the application runs for 100 cycles it generates 10 stores; assume each store is 4 bytes; 40 bytes are generated per 100/3 ns i.e. BW of 1.2 GB/s
  - A 1 GB/s bus cannot even support one processor
  - There are multiple processors and also there are read misses
- Writeback caches absorb most of the write traffic
  - Writes that hit in cache do not go on bus (not visible to others)
  - Complicated coherence protocol with many choices

Memory Consistency

- Need a more formal description of memory ordering
  - How to establish the order between reads and writes from different processors?
- 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

Previous Next

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

Previous Next

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

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? We will discuss it in detail later

Previous Next

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)

Previous Next

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)

Previous Next