Lecture 3: Coherence Protocols

- Topics: consistency models, coherence protocol examples

Sequential Consistency

- A multiprocessor is sequentially consistent if the result of the execution is achievable by maintaining program order within a processor and interleaving accesses by different processors in an arbitrary fashion

- For example, the code below should ensure mutual exclusion on a sequentially consistent machine

```
Initially A = B = 0
  P1                    P2
A ← 1                 B ← 1
  …                     …
if (B == 0)           if (A == 0)
  Crit.Section          Crit.Section
```

Relaxing Memory Ordering

```
Initially A = B = 0
  P1                    P2
A ← 1                 B ← 1
 …                     …
if (B == 0)           if (A == 0)
  Crit.Section          Crit.Section
```

- Executing memory accesses in order is extremely slow; we attempt optimizations → seq consistency is lost
- For example, each processor can be out-of-order; within P1, the write to A and the read of B are independent since they refer to different memory locations
- Ooo execution will allow each process to enter CS

Relaxed Consistency Models

- In order to write correct programs, the programmer must understand that memory accesses do not always happen in order
- The consistency model specifies how memory ordering differs from that of sequential consistency
- If the programmer demands sequential consistency in places, he/she can impose it with special fence instructions – a fence ensures that we make progress only after completing earlier memory accesses
- Fences are slow – a better understanding of the program and the consistency model can eliminate some fences

Cache Coherence

A multiprocessor system is cache coherent if

- a value written by a processor is eventually visible to reads by other processors – write propagation

- two writes to the same location by two processors are seen in the same order by all processors – write serialization

Cache Coherence Protocols

- Directory-based: A single location (directory) keeps track of the sharing status of a block of memory

- Snooping: Every cache block is accompanied by the sharing status of that block – all cache controllers monitor the shared bus so they can update the sharing status of the block, if necessary

  - Write-invalidate: a processor gains exclusive access of a block before writing by invalidating all other copies
  - Write-update: when a processor writes, it updates other shared copies of that block

Protocol-I MSI

- 3-state write-back invalidation bus-based snooping protocol
- Each block can be in one of three states – invalid, shared, modified (exclusive)
- A processor must acquire the block in exclusive state in order to write to it – this is done by placing an exclusive read request on the bus – every other cached copy is invalidated
- When some other processor tries to read an exclusive block, the block is demoted to shared

Design Issues, Optimizations

- When does memory get updated?
  - demotion from modified to shared?
  - move from modified in one cache to modified in another?

- Who responds with data? – memory or a cache that has the block in exclusive state – does it help if sharers respond?

- We can assume that bus, memory, and cache state transactions are atomic – if not, we will need more states

- A transition from shared to modified only requires an upgrade request and no transfer of data

- Is the protocol simpler for a write-through cache?

4-State Protocol

- Multiprocessors execute many single-threaded programs
- A read followed by a write will generate bus transactions to acquire the block in exclusive state even though there are no sharers
- Note that we can optimize protocols by adding more states – increases design/verification complexity

MESI Protocol

- The new state is exclusive-clean – the cache can service read requests and no other cache has the same block
- When the processor attempts a write, the block is upgraded to exclusive-modified without generating a bus transaction
- When a processor makes a read request, it must detect if it has the only cached copy – the interconnect must include an additional signal that is asserted by each cache if it has a valid copy of the block

Design Issues

- When caches evict blocks, they do not inform other caches – it is possible to have a block in shared state even though it is an exclusive-clean copy

- Cache-to-cache sharing: SRAM vs. DRAM latencies, contention in remote caches, protocol complexities (memory has to wait, which cache responds), can be especially useful in distributed memory systems

- The protocol can be improved by adding a fifth state (owner – MOESI) – the owner services reads (instead of memory)

Update Protocol (Dragon)

- 4-state write-back update protocol, first used in the Dragon multiprocessor (1984)
- Write-back update is not the same as write-through – on a write, only caches are updated, not memory
- Goal: writes may usually not be on the critical path, but subsequent reads may be

4 States

- No invalid state
- Modified and Exclusive-clean as before: used when there is a sole cached copy
- Shared-clean: potentially multiple caches have this block and main memory may or may not be up-to-date
- Shared-modified: potentially multiple caches have this block, main memory is not up-to-date, and this cache must update memory – only one block can be in Sm state
- In reality, one state would have sufficed – more states to reduce traffic

Design Issues

- If the update is also sent to main memory, the Sm state can be eliminated
- If all caches are informed when a block is evicted, the block can be moved from shared to M or E – this can help save future bus transactions
- Having an extra wire to determine exclusivity seems like a worthy trade-off in update systems

Examples

	P1			P2		
	MSI	MESI	Dragon	MSI	MESI	Dragon
• P1: Rd X						
• P1: Wr X						
• P2: Rd X						
• P1: Wr X						
• P1: Wr X						
• P2: Rd X						
• P2: Wr X						
Total transfers:						

Title

- Bullet