Lecture 3: Snooping Protocols

- Topics: snooping-based cache coherence implementations

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

State Transitions

To / From	NP	I	E	S	M
NP	0	0	1.25	0.96	1.68
I	0.64	0	0	1.87	0.002
E	0.20	0	14.0	0.02	1.00
S	0.42	2.5	0	134.7	2.24
M	2.63	0.002	0	2.3	843.6

NP – Not Present

State transitions per 1000 data memory references for Ocean

To / From	NP	I	E	S	M
NP	--	--	BusRd	BusRd	BusRdX
I	--	--	BusRd	BusRd	BusRdX
E	--	--	--	--	--
S	--	--	Not possible	--	BusUpgr
M	BusWB	BusWB	Not possible	BusWB	--

Bus actions for each state transition

Basic Implementation

- Assume single level of cache, atomic bus transactions

- It is simpler to implement a processor-side cache controller that monitors requests from the processor and a bus-side cache controller that services the bus

- Both controllers are constantly trying to read tags
  - tags can be duplicated (moderate area overhead)
  - unlike data, tags are rarely updated
  - tag updates stall the other controller

Reporting Snoop Results

- Uniprocessor system: initiator places address on bus, all devices monitor address, one device acks by raising a wired-OR signal, data is transferred

- In a multiprocessor, memory has to wait for the snoop result before it chooses to respond – need 3 wired-OR signals: (i) indicates that a cache has a copy, (ii) indicates that a cache has a modified copy, (iii) indicates that the snoop has not completed

- Ensuring timely snoops: the time to respond could be fixed or variable (with the third wired-OR signal), or the memory could track if a cache has a block in M state

Non-Atomic State Transitions

- Note that a cache controller's actions are not all atomic: tag look-up, bus arbitration, bus transaction, data/tag update

- Consider this: block A in shared state in P1 and P2; both issue a write; the bus controllers are ready to issue an upgrade request and try to acquire the bus; is there a problem?

- The controller can keep track of additional intermediate states so it can react to bus traffic (e.g. S→M, I→M, I→S,E)

- Alternatively, eliminate upgrade request; use the shared wire to suppress memory's response to an exclusive-rd

Livelock

- Livelock can happen if the processor-cache handshake is not designed correctly
- Before the processor can attempt the write, it must acquire the block in exclusive state
- If all processors are writing to the same block, one of them acquires the block first – if another exclusive request is seen on the bus, the cache controller must wait for the processor to complete the write before releasing the block -- else, the processor’s write will fail again because the block would be in invalid state

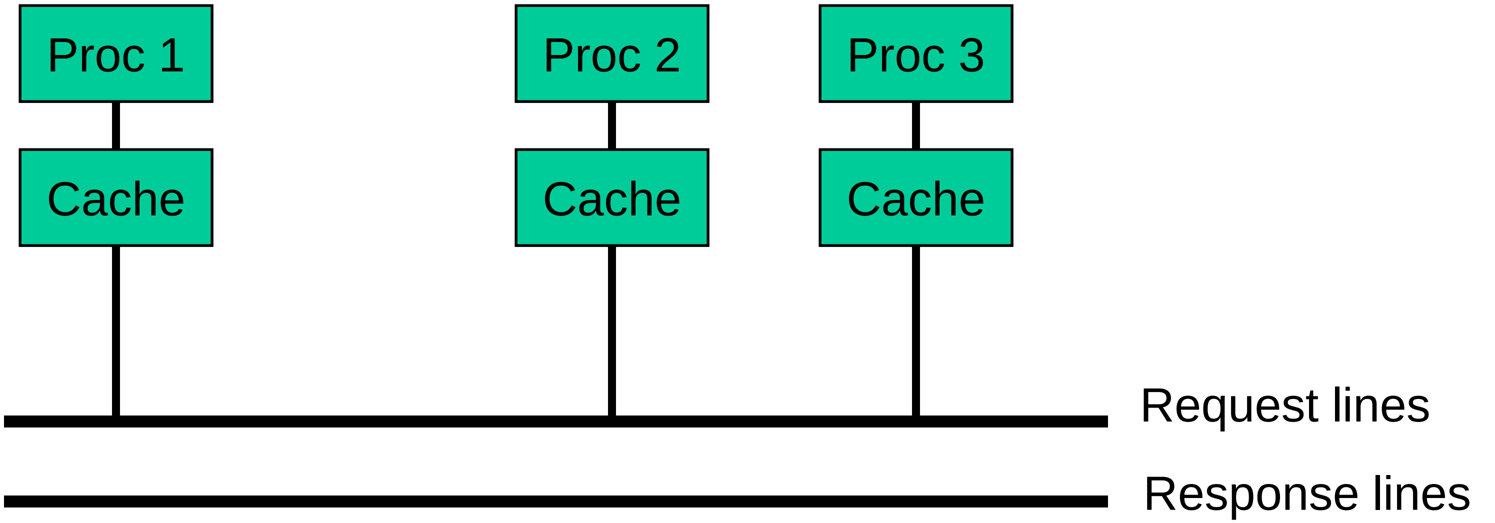
Split Transaction Bus

- What would it take to implement the protocol correctly while assuming a split transaction bus?

- Split transaction bus: a cache puts out a request, releases the bus (so others can use the bus), receives its response much later

- Assumptions:
  - only one request per block can be outstanding
  - separate lines for addr (request) and data (response)

Split Transaction Bus



Design Issues

- When does the snoop complete? What if the snoop takes a long time?
- What if the buffer in a processor/memory is full? When does the buffer release an entry? Are the buffers identical?
- How does each processor ensure that a block does not have multiple outstanding requests?
- What determines the write order – requests or responses?

Design Issues II

- What happens if a processor is arbitrating for the bus and witnesses another bus transaction for the same address?
- If the processor issues a read miss and there is already a matching read in the request table, can we reduce bus traffic?

Title

- Bullet