Lect. 5: Snooping Coherence Protocol

- Snooping coherence on simple shared bus
  - "Easy" as all processors and memory controller can observe all transactions
  - Bus-side cache controller monitors the tags of the lines involved and reacts if necessary by checking the contents and state of the local cache
  - Bus provides a serialization point (i.e., any transaction A is either before or after another transaction B)
    - More complex with split transaction buses

Cache states:
- 00 = invalid
- 01 = shared
- 10 = modified
Lect. 5: Snooping Coherence Protocol

- Read/Write miss
  - When should memory provide data?
    - Wait until inhibit is deasserted
    - If Wired OR (sharers, modified) is false, then provides data.

- Write-backs?
  - Don’t want to wait for writes → Write-back buffer
Problem: conflict when processor and bus-side controller must check the cache

Solutions:
- Use dual-ported modules for the tag and state array
- Or, duplicate tag and state array
  - Both must be kept consistent when one is changed, which introduces some amount of conflicts
Snooping on Simple Bus

Fig 6.4
Culler et al.
Problem: even if bus is atomic, state transitions are not instantaneous and may require several steps → transitions are not atomic
  – E.g., read- miss transaction = wait for bus + wait for bus-side controllers to check cache + data response (or memory response)
  – E.g. write-upgrade transactions = wait for bus + wait for bus-side controllers to invalidate

What to do if there are conflicting requests (i.e to same cache line) from the same processor or from the bus?
  – E.g., an upgrade request may lose bus arbitration to another processor’s and may have to be re-issued as a full write miss (due to the intervening invalidation)

Solution:
  – Introduce transient states to cache lines and the protocol (the I, S, M, etc states seen in Lecture 4 are then called the stable states)
Example: Extended MESI Protocol

Transactions originating at this CPU:

- CPU read
- CPU write
- Bus granted & shr.
- Bus granted & no shr.
- CPU read hit
- CPU write hit
- CPU write miss
- Conflict
Questions:

- Processor interacts with L1 while bus snooping device interacts with L2, and propagating such operations up or down is not instantaneous.
- Note: L2 lines could be bigger than L1 lines. 

Cache states:
- 00 = invalid
- 01 = shared
- 10 = modified
Snooping with Multi-Level Hierarchies

Solution:

1. Maintain inclusion property
   - Lines in L1 must also be in L2 → no data is found solely in L1, so no risk of missing a relevant transaction when snooping at L2
   - Lines M state in L1 must also be in M state in L2 → snooping controller at L2 can identify all data that is modified locally

2. Propagate coherence transactions to L1 as well.
   - Propagate all transactions from to L1, whether relevant or not
   - Keep extra state in the L2 lines to tell whether the line is also present in L1 or not (inclusion bits). If it is present in L2, but inclusion bits say it is not present in L1, no need to propagate transaction to L1.
Maintaining inclusion property

Assume: L1: associativity $a_1$, number of sets $n_1$, block size $b_1$

L2: associativity $a_2$, number of sets $n_2$, block size $b_2$

- Difficulty: Replacement policy (e.g., LRU)
  Assume: $a_1=a_2=2$; $b_1=b_2$; $n_2=k \times n_1$; lines $m_1$, $m_2$, and $m_3$ map to same set in L1 and the same set in L2; initially $m_1$ is present in L1 and L2
Snooping with Multi-Level Hierarchies

- Maintaining inclusion property

  Assume: L1: associativity $a_1$, number of sets $n_1$, block size $b_1$
  
  L2: associativity $a_2$, number of sets $n_2$, block size $b_2$

  - Difficulty: Different line sizes

    Assume: $a_1=a_2=1$; $b_1=1$, $b_2=2$; $n_1=4$, $n_2=8$

    Thus, words $w_0$ and $w_{17}$ can coexist in L1, but not in L2
Snooping with Multi-Level Hierarchies

- **Maintaining inclusion property**
  - Most combinations of L1/L2 size, associativity, and line size do not automatically lead to inclusion
  - Static solution: One solution is to have $a_1 = 1$, $a_2 \geq 1$, $b_1 = b_2$, and $n_1 \leq n_2$
  - Dynamic solution: More common solution is to invalidate the L1 line (or lines, if $b_1 < b_2$) upon replacing a line in L2. Must also invalidate L1 line(s) when L2 line is invalidated due to coherence
    - Propagate all invalidations from L2 to L1, whether relevant or not
    - Keep extra state in the L2 lines to tell whether the line is also present in L1 or not (inclusion bits)
  - Finally, add a new state to L2 (modified-but-stale) to keep track of lines that are in state M in L1 (or make L1 write-through)
Non-split-transaction buses are idle from when the address request is finished until the data returns from memory or another cache.

In split-transaction buses transactions are split into a request transaction and a response transaction, which can be separated.

Sometimes implemented as two buses: one for requests and one for responses.
Snooping with Split-Transaction Buses

- Problems
  - Multiple requests can clash (e.g., a read and a write, or two writes, to the same data) (Note that this is more complicated than the case in Slide 4, as now different transactions may be at different stages of service)
  - Buffers used to hold pending transactions may fill up (flow control is required)
  - Responses from multiple requests may appear in a different order than their respective requests
    - Responses and requests must then be matched using tags for each transaction
Snooping with Split-Transaction Buses

- Clashing requests
  - snoop controllers keep track themselves of what transactions are pending, in case there is conflict
  - Allow only one request at a time for each line (e.g., SGI Challenge)

- Flow control
  - Use **negative acknowledgement (NACK)** when buffers are full (requests must be retried later; a bit more tricky with responses, due to danger of deadlock) (e.g., SGI Challenge)
  - Or, design the size of all queues for the worst case scenario

- Ordering of transactions
  - Responses can appear in any order → the interleaving of the requests fully determine the order of transactions (e.g., SGI Challenge)
  - Or, enforce a FIFO order of transactions across the whole system (caches + memory) (e.g., Sun Enterprise)
Snooping with Ring

- Like a bus, rings easily support broadcasts
- Snooping implemented by all controllers checking the message as it passes by and re-injecting it into the ring
- Potentially multiple transactions can be simultaneously on different stretches of the ring (harder to enforce proper ordering)
- Large latency for long rings and growing linearly with number of processors
- Used to provide coherence across multiple chips in recent CMP systems (e.g., IBM Power 5)
References and Further Reading

- **Original (hardware) cache coherence works:**
  


- **An early survey of cache coherence protocols:**
  

- **Discussion on the difficulties of maintaining inclusion**
  
Modern bus-based coherent multiprocessors:

Some software cache coherence schemes: