Module 12: "Multiprocessors on a Snoopy Bus"

Lecture 26: "Case Studies"

- Conflict resolution
- Path of a cache miss
- Write serialization
- Write atomicity and SC
- Another example
- In-order response
- Multi-level caches
- Dependence graph
- Multiple outstanding requests
- SGI Challenge
- Sun Enterprise
- Sun Gigaplane bus

[From Chapter 6 of Culler, Singh, Gupta]

Previous Next

Conflict resolution

- Use the pending request table to resolve conflicts
  - Every processor has a copy of the table
  - Before arbitrating for the address bus every processor looks up the table to see if there is a match
  - In case of a match the request is not issued and is held in a pending buffer
- Flow control is needed at different levels
  - Essentially need to detect if any buffer is full
  - SGI Challenge uses a separate NACK line for each of address and data phases
  - Before the phases reach the “ack” cycle any cache controller can assert the NACK line if it runs out of some critical buffer; this invalidates the transaction and the requester must retry (may use back-off and/or priority)
  - Sun Enterprise requires the receiver to generate the retry when it has buffer space (thus only one retry)

Path of a cache miss

- Assume a read miss
  - Look up request table; in case of a match with BusRd just mark the entry indicating that this processor will snoop the response from the bus and that it will also assert the shared line
  - In case of a request table hit with BusRdX the cache controller must hold on to the request until the conflict resolves
  - In case of a request table miss the requester arbitrates for address bus; while arbitrating if a conflicting request arrives, the controller must put a NOP transaction within the slot it is granted and hold on to the request until the conflict resolves
- Suppose the requester succeeds in putting the request on address/command bus
  - Other cache controllers snoop the request, register it in request table (the requester also does this), take appropriate coherence action within own cache hierarchy, main memory also starts fetching the cache line
  - If a cache holds the line in M state it should source it on bus during response phase; it keeps the inhibit line asserted until it gets the data bus; then it lowers inhibit line and asserts the modified line; at this point the memory controller aborts the data fetch/response and instead fields the line from the data bus for writing back
- If the memory fetches the line even before the snoop is complete, the inhibit line will not allow the memory controller to launch the data on bus
  - After the inhibit line is lowered depending on the state of the modified line memory cancels the data response
  - If no one has the line in M state, the requester grabs the response from memory
- A store miss is similar
  - Only difference is that even if a cache has the line in M state, the memory controller does not write the response back
  - Also any pending BusUpgr to the same cache line must be converted to BusReadX

Write serialization

- In a split-transaction bus setting, the request table provides sufficient support for write

serialization

- Requests to the same cache line are not allowed to proceed at the same time
- A read to a line after a write to the same line can be launched only after the write response phase has completed; this guarantees that the read will see the new value
- A write after a read to the same line can be started only after the read response has completed; this guarantees that the value of the read cannot be altered by the value written

Previous Next

Write atomicity and SC

- Sequential consistency (SC) requires write atomicity i.e. total order of all writes seen by all processors should be identical
  - Since a BusRdX or BusUpgr does not wait until the invalidations are actually applied to the caches, you have to be careful

    P0: A=1; B=1;
    P1: print B; print A

  - Under SC (A, B) = (0, 1) is not allowed
  - Suppose to start with P1 has the line containing A in cache, but not the line containing B
  - The stores of P0 queue the invalidation of A in P1's cache controller
  - P1 takes read miss for B, but the response of B is re-ordered by P1's cache controller so that it overtakes the invalidaton (thought it may be better to prioritize reads)

Another example

P0: A=1; print B;

P1: B=1; print A;

- Under SC (A, B) = (0, 0) is not allowed
- Same problem if P0 executes both instructions first, then P1 executes the write of B (which let's assume generates an upgrade so that it is marked complete as soon as the address arbitration phase finishes), then the upgrade completion is re-ordered with the pending invalidation of A
- So, the reason these two cases fail is that the new values are made visible before older invalidations are applied
- One solution is to have a strict FIFO queue between the bus controller and the cache hierarchy
- But it is sufficient as long as replies do not overtake invalidations; otherwise the bus responses can be re-ordered without violating write atomicity and hence SC (e.g., if there are only read and write responses in the queue, it sometimes may make sense to prioritize read responses)

In-order response

- In-order response can simplify quite a few things in the design
  - The fully associative request table can be replaced by a FIFO queue
  - Conflicting requests where one is a write can actually be allowed now (multiple reads were allowed even before although only the first one actually appears on the bus)
  - Consider a BusRdX followed by a BusRd from two different processors
  - With in-order response it is guaranteed that the BusRdX response will be granted the data bus before the BusRd response (which may not be true for ooo response and hence such a conflict is disallowed)
  - So when the cache controller generating the BusRdX sees the BusRd it only notes that

it should source the line for this request after its own write is completed

- The performance penalty may be huge
  - Essentially because of the memory
  - Consider a situation where three requests are pending to cache lines A, B, C in that order
  - A and B map to the same memory bank while C is in a different bank
  - Although the response for C may be ready long before that of B, it cannot get the bus

Previous Next

Multi-level caches

- Split-transaction bus makes the design of multi-level caches a little more difficult
  - The usual design is to have queues between levels of caches in each direction
  - How do you size the queues? Between processor and L1 one buffer is sufficient (assume one outstanding processor access), L1-to-L2 needs P+1 buffers (why?), L2-to-L1 needs P buffers (why?), L1 to processor needs one buffer
  - With smaller buffers there is a possibility of deadlock: suppose the L1-to-L2 and L2-to-L1 have one queue entry each, there is a request in L1-to-L2 queue and there is also an intervention in L2-to-L1 queue; clearly L1 cannot pick up the intervention because it does not have space to put the reply in L1-to-L2 queue while L2 cannot pick up the request because it might need space in L2-to-L1 queue in case of an L2 hit
- Formalizing the deadlock with dependence graph
  - There are four types of transactions in the cache hierarchy: 1. Processor requests (outbound requests), 2. Responses to processor requests (inbound responses), 3. Interventions (inbound requests), 4. Intervention responses (outbound responses)
  - Processor requests need space in L1-to-L2 queue; responses to processors need space in L2-to-L1 queue; interventions need space in L2-to-L1 queue; intervention responses need space in L1-to-L2 queue
  - Thus a message in L1-to-L2 queue may need space in L2-to-L1 queue (e.g. a processor request generating a response due to L2 hit); also a message in L2-to-L1 queue may need space in L1-to-L2 queue (e.g. an intervention response)
  - This creates a cycle in queue space dependence graph

Dependence graph

- Represent a queue by a vertex in the graph
  - Number of vertices = number of queues
- A directed edge from vertex u to vertex v is present if a message at the head of queue u may generate another message which requires space in queue v
- In our case we have two queues
  - L2-L1 and L1-L2; the graph is not a DAG, hence deadlock

Multi-level caches

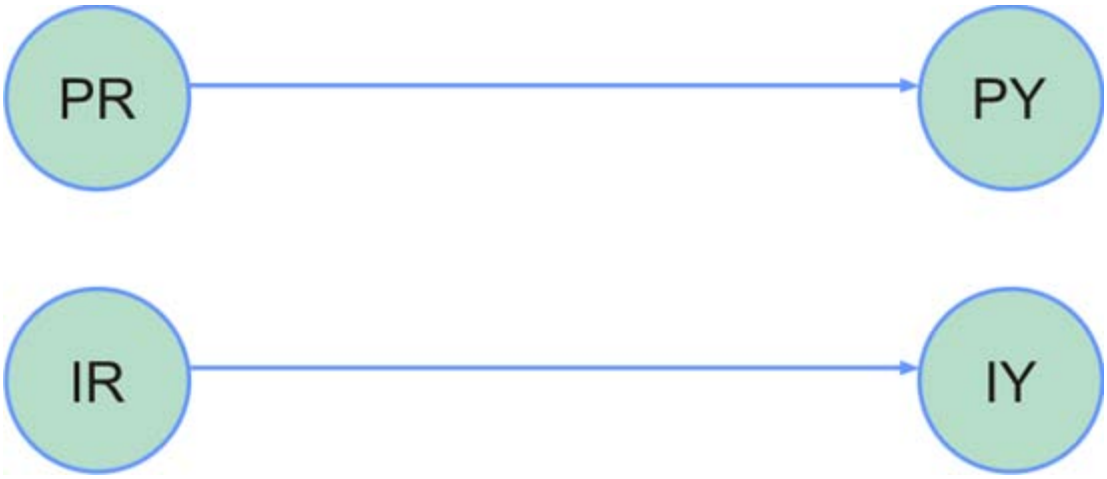
- In summary
  - L2 cache controller refuses to drain L1-to-L2 queue if there is no space in L2-to-L1 queue; this is rather conservative because the message at the head of L1-to-L2 queue may not need space in L2-to-L1 queue e.g., in case of L2 miss or if it is an intervention

reply; but after popping the head of L1-to-L2 queue it is impossible to backtrack if the message does need space in L2-to-L1 queue
    - Similarly, L1 cache controller refuses to drain L2-to-L1 queue if there is no space in L1-to-L2 queue
    - How do we break this cycle?
    - Observe that responses for processor requests are guaranteed not to generate any more messages and intervention requests do not generate new requests, but can only generate replies
- Solving the queue deadlock
    - Introduce one more queue in each direction i.e. have a pair of queues in each direction
    - L1-to-L2 processor request queue and L1-to-L2 intervention response queue
    - Similarly, L2-to-L1 intervention request queue and L2-to-L1 processor response queue
    - Now L2 cache controller can serve L1-to-L2 processor request queue as long as there is space in L2-to-L1 processor response queue, but there is no constraint on L1 cache controller for draining L2-to-L1 processor response queue
    - Similarly, L1 cache controller can serve L2-to-L1 intervention request queue as long as there is space in L1-to-L2 intervention response queue, but L1-to-L2 intervention response queue will drain as soon as bus is granted

Previous Next

Dependence graph

- Now we have four queues
  - Processor request (PR) and intervention reply (IY) are L1 to L2
  - Processor reply (PY) and intervention request (IR) are L2 to L1



- Possible to combine PR and IY into a supernode of the graph and still be cycle-free
  - Leads to one L1 to L2 queue
- Similarly, possible to combine IR and PY into a supernode
  - Leads to one L2 to L1 queue
- Cannot do both
  - Leads to cycle as already discussed
- Bottomline: need at least three queues for two-level cache hierarchy

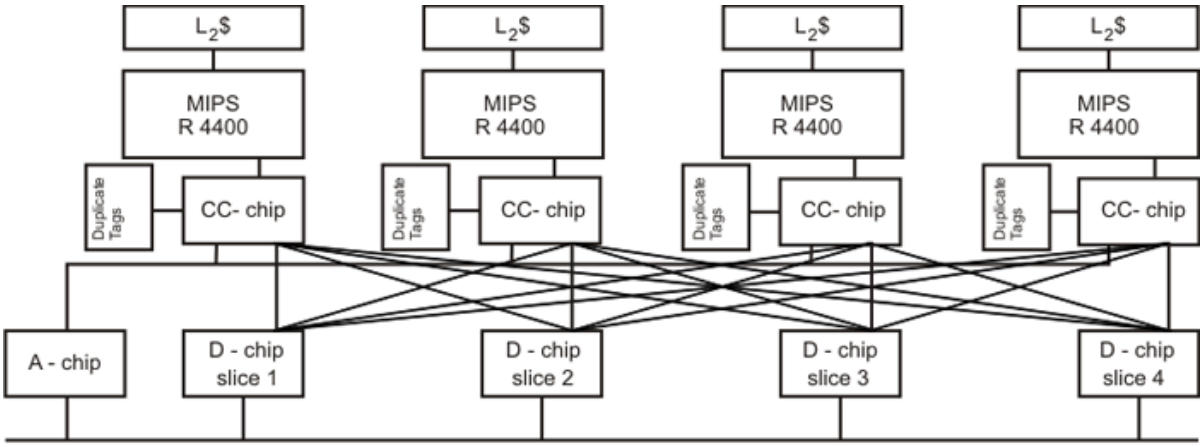
Multiple outstanding requests

- Today all processors allow multiple outstanding cache misses
  - We have already discussed issues related to ooo execution
  - Not much needs to be added on top of that to support multiple outstanding misses
  - For multi-level cache hierarchy the queue depths may be made bigger for performance reasons
  - Various other buffers such as writeback buffer need to be made bigger

Previous Next

Module 12: "Multiprocessors on a Snoopy Bus"
Lecture 26: "Case Studies"

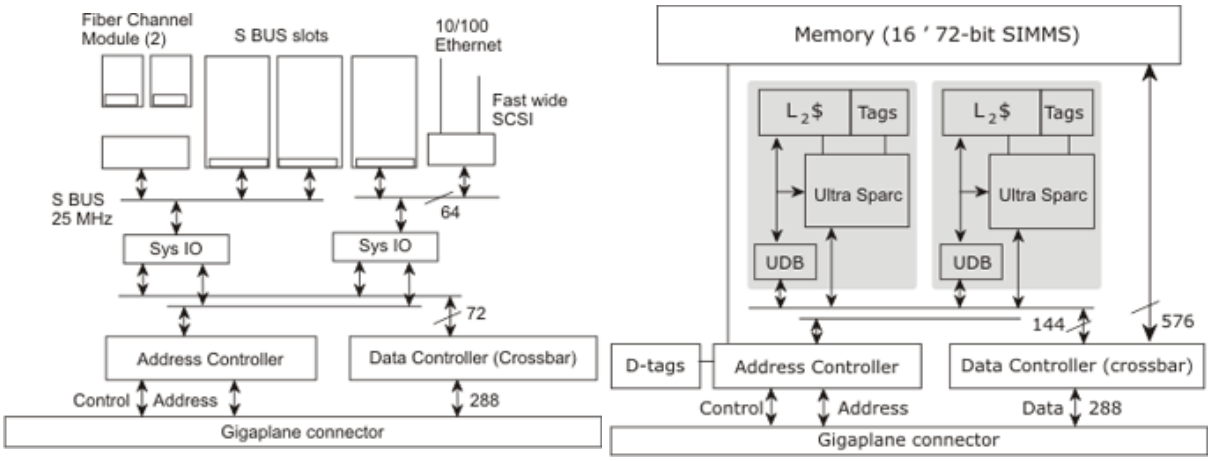
SGI Challenge



Powerpath - 2 bus

- Supports 36 MIPS R4400 (4 per board) or 18 MIPS R8000 (2 per board)
- A-chip has the address bus interface, request table
- CC-chip handles coherence through the duplicate set of tags
- Each D-chip handles 64 bits of data and as a whole 4 D-chips interface to a 256-bit wide data bus

Sun Enterprise



- Supports up to 30 UltraSPARC processors
- 2 processors and 1 GB memory per board
- Wide 64-byte memory bus and hence two memory cycles to transfer the entire cache line (128 bytes)

Sun Gigaplane bus

- Split-transaction, 256 bits data, 41 bits address, 83.5 MHz (compare to 47.6 MHz of SGI Powerpath-2)
- Supports 16 boards
- 112 outstanding transactions (up to 7 from each board)

- Snoop result is available 5 cycles after the request phase
- Memory fetches data speculatively
- MOESI protocol

Previous Next