Module 12: "Multiprocessors on a Snoopy Bus"

Lecture 25: "Protocols for Split-transaction Buses"

- Recap of inclusion
- Inclusion and snoop
- L2 to L1 interventions
- Invalidation acks?
- Intervention races
- Tag RAM design
- Exclusive cache levels
- Split-transaction bus
- New issues
- SGI Powerpath-2 bus
- Bus interface logic
- Snoop results

[From Chapter 6 of Culler, Singh, Gupta]

Previous Next

Recap of inclusion

- A processor read
  - Looks up L1 first and in case of miss goes to L2, and finally may need to launch a BusRd request if it misses in L2
  - Finally, the line is in S state in both L1 and L2
- A processor write
  - Looks up L1 first and if it is in I state sends a ReadX request to L2 which may have the line in M state
  - In case of L2 hit, the line is filled in M state in L1
  - In case of L2 miss, if the line is in S state in L2 it launches BusUpgr; otherwise it launches BusRdX; finally, the line is in state M in both L1 and L2
  - If the line is in S state in L1, it sends an upgrade request to L2 and either there is an L2 hit or L2 just conveys the upgrade to bus (Why can't it get changed to BusRdX?)
- L1 cache replacement
  - Replacement of a line in S state may or may not be conveyed to L2
  - Replacement of a line in M state must be sent to L2 so that it can hold the most up-to-date copy
  - The line is in I state in L1 after replacement, the state of line remains unchanged in L2
- L2 cache replacement
  - Replacement of a line in S state may or may not generate a bus transaction; it must send a notification to the L1 caches so that they can invalidate the line to maintain inclusion
  - Replacement of a line in M state first asks the L1 cache to send all the relevant L1 lines (these are the most up-to-date copies) and then launches a BusWB
  - The state of line in both L1 and L2 is I after replacement
- Replacement of a line in E state from L1?
- Replacement of a line in E state from L2?
- Replacement of a line in O state from L1?
- Replacement of a line in O state from L2?
- In summary
  - A line in S state in L2 may or may not be in L1 in S state
  - A line in M state in L2 may or may not be in L1 in M state; Why? Can it be in S state?
  - A line in I state in L2 must not be present in L

Inclusion and snoop

- BusRd snoop
  - Look up L2 cache tag; if in I state no action; if in S state no action; if in M state assert wired-OR M line, send read intervention to L1 data cache, L1 data cache sends lines back, L2 controller launches line on bus, both L1 and L2 lines go to S state
- BusRdX snoop
  - Look up L2 cache tag; if in I state no action; if in S state invalidate and also notify L1; if in M state assert wired-OR M line, send readX intervention to L1 data cache, L1 data cache sends lines back, L2 controller launches line on bus, both L1 and L2 lines go to I state
- BusUpgr snoop
  - Similar to BusRdX without the cache line flush

L2 to L1 interventions

- Two types of interventions
  - One is read/readX intervention that requires data reply
  - Other is plain invalidation that does not need data reply
- Data interventions can be eliminated by making L1 cache write-through
  - But introduces too much of write traffic to L2
  - One possible solution is to have a store buffer that can handle the stores in background obeying the available BW, so that the processor can proceed independently; this can easily violate sequential consistency unless store buffer also becomes a part of snoop logic
- Useless invalidations can be eliminated by introducing an inclusion bit in L2 cache state

Previous Next

Invalidation acks?

- On a BusRdX or BusUpgr in case of a snoop hit in S state L2 cache sends invalidation to L1 caches
  - Does the snoop logic wait for an invalidation acknowledgment from L1 cache before the transaction can be marked complete?
  - Do we need a two-phase mechanism?
  - What are the issues?

Intervention races

- Writebacks introduce new races in multi-level cache hierarchy
  - Suppose L2 sends a read intervention to L1 and in the meantime L1 decides to replace that line (due to some conflicting processor access)
  - The intervention will naturally miss the up-to-date copy
  - When the writeback arrives at L2, L2 realizes that the intervention race has occurred (need extra hardware to implement this logic; what hardware?)
  - When the intervention reply arrives from L1, L2 can apply the newly received writeback and launch the line on bus
  - Exactly same situation may arise even in uniprocessor if a dirty replacement from L2 misses the line in L1 because L1 just replaced that line too

Tag RAM design

- A multi-level cache hierarchy reduces tag contention
  - L1 tags are mostly accessed by the processor because L2 cache acts as a filter for external requests
  - L2 tags are mostly accessed by the system because hopefully L1 cache can absorb most of the processor traffic
  - Still some machines maintain duplicate tags at all or the outermost level only

Exclusive cache levels

- AMD K7 (Athlon XP) and K8 (Athlon64, Opteron) architectures chose to have exclusive levels of caches instead of inclusive
  - Definitely provides you much better utilization of on-chip caches since there is no duplication
  - But complicates many issues related to coherence
  - The uniprocessor protocol is to refill requested lines directly into L1 without placing a copy in L2; only on an L1 eviction put the line into L2; on an L1 miss look up L2 and in case of L2 hit replace line from L2 and put it in L1 (may have to replace multiple L1 lines to accommodate the full L2 line; not sure what K8 does: possible to maintain inclusion bit per L1 line sector in L2 cache)
  - For multiprocessors one solution could be to have one snoop engine per cache level and a tournament logic that selects the successful snoop result

Split-transaction bus

- Atomic bus leads to underutilization of bus resources
  - Between the address is taken off the bus and the snoop responses are available the bus stays idle
  - Even after the snoop result is available the bus may remain idle due to high memory access latency
- Split-transaction bus divides each transaction into two parts: request and response
  - Between the request and response of a particular transaction there may be other requests and/or responses from different transactions
  - Outstanding transactions that have not yet started or have completed only one phase are buffered in the requesting cache controllers

New issues

- Split-transaction bus introduces new protocol races
  - P0 and P1 have a line in S state and both issue BusUpgr, say, in consecutive cycles
  - Snoop response arrives later because it takes time
  - Now both P0 and P1 may think that they have ownership
- Flow control is important since buffer space is finite
- In-order or out-of-order response?
  - Out-of-order response may better tolerate variable memory latency by servicing other requests
  - Pentium Pro uses in-order response
  - SGI Challenge and Sun Enterprise use out-of-order response i.e. no ordering is enforced

SGI Powerpath-2 bus

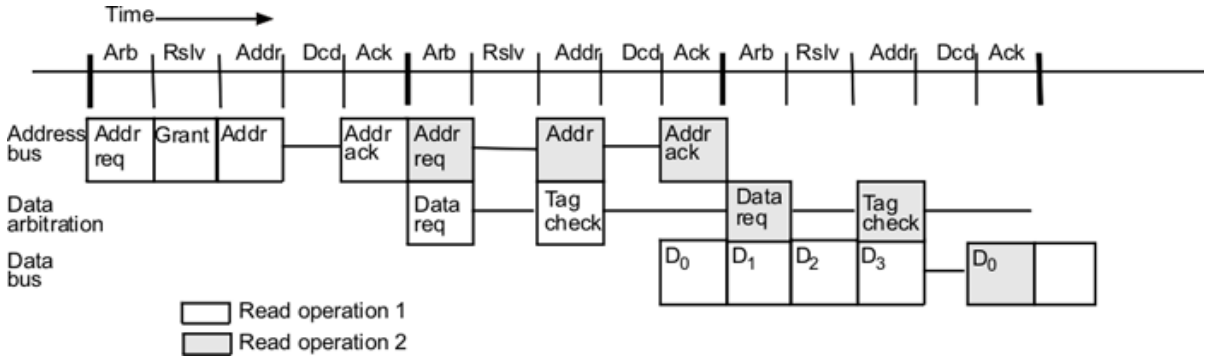
- Used in SGI Challenge
  - Conflicts are resolved by not allowing multiple bus transactions to the same cache line
  - Allows eight outstanding requests on the bus at any point in time
  - Flow control on buffers is provided by negative acknowledgments (NACKs): the bus has a dedicated NACK line which remains asserted if the buffer holding outstanding transactions is full; a NACKed transaction must be retried
  - The request order determines the total order of memory accesses, but the responses may be delivered in a different order depending on the completion time of them
  - In subsequent slides we call this design Powerpath-2 since it is loosely based on that
- Logically two separate buses
  - Request bus for launching the command type (BusRd, BusWB etc.) and the involved address
  - Response bus for providing the data response, if any
  - Since responses may arrive in an order different from the request order, a 3-bit tag is assigned to each request
  - Responses launch this tag on the tag bus along with the data reply so that the address bus may be left free for other requests
- The data bus is 256-bit wide while a cache line is 128 bytes
  - One data response phase needs four bus cycles along with one additional hardware

turnaround cycle

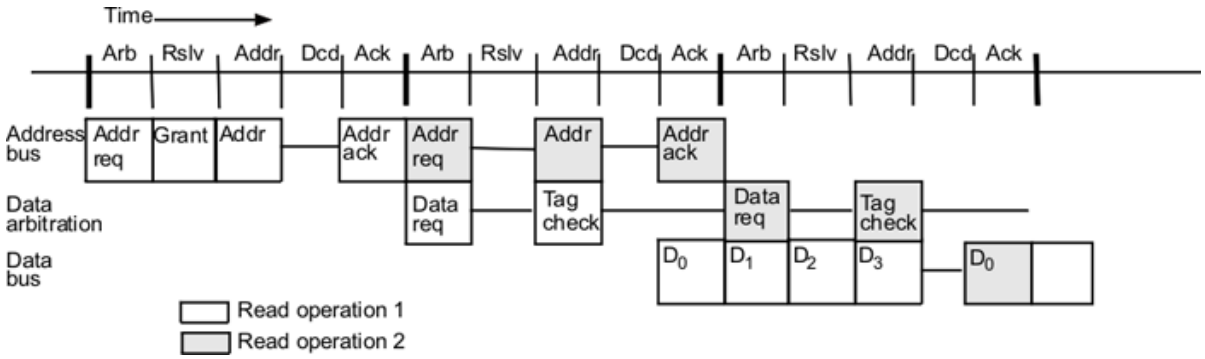
Previous Next

Module 12: "Multiprocessors on a Snoopy Bus"
Lecture 25: "Protocols for Split-transaction Buses"

SGI Powerpath-2 bus

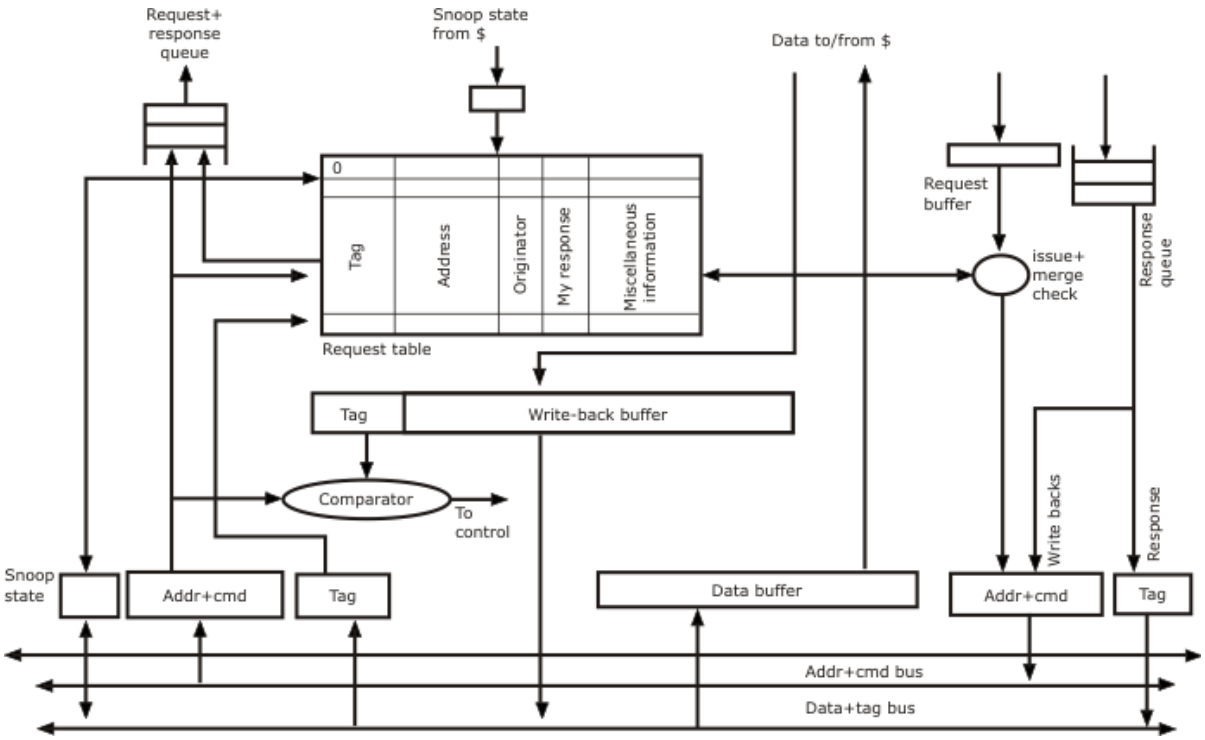


- Essentially two main buses and various control wires for snoop results, flow control etc.
  - Address bus: five cycle arbitration, used during request
  - Data bus: five cycle arbitration, five cycle transfer, used during response
  - Three different transactions may be in one of these three phases at any point in time



- Forming a total order
  - After the decode cycle during request phase every cache controller takes appropriate coherence actions i.e. BusRd downgrades M line to S, BusRdX invalidates line
  - If a cache controller does not get the tags due to contention with the processor; it simply lengthens the ack phase beyond one cycle
  - Thus the total order is formed during the request phase itself i.e. the position of each request in the total order is determined at that point
- BusWB case
  - BusWB only needs the request phase
  - However needs both address and data lines together
  - Must arbitrate for both together
- BusUpgr case
  - Consists only of the request phase
  - No response or acknowledgment
  - As soon as the “ack” phase of address arbitration is completed by the issuing node, the upgrade has sealed a position in the total order and hence is marked complete by sending a completion signal to the issuing processor by its local bus controller (each node has its own bus controller to handle bus requests)

Bus interface logic



A request table entry is freed when the response is observed on the bus

Snoop results

- Three snoop wires: shared, modified, inhibit (all wired-OR)
  - The inhibit wire helps in holding off snoop responses until the data response is launched on the bus
  - Although the request phase determines who will source the data i.e. some cache or memory, the memory controller does not know it
  - The cache with a modified copy keeps the inhibit line asserted until it gets the data bus and flushes the data; this prevents memory controller from sourcing the data
  - Otherwise memory controller arbitrates for the data bus
  - When the data appears all cache controllers appropriately assert the shared and modified line
  - Why not launch snoop results as soon as they are available?