Path of a Cache Miss

(1) Handling a Read Miss
(2) Handling a Write Miss

Handling a Read Miss

- Processor wants to read a block which is not in the cache.
  - Need to issue BusRd
- Check if Request table has request for same block
  - Yes: see if you can get the block when the existing request is being served
  - No: issue request and watch-out for race conditions

Handling read miss

Prior request BusRd

Yes: prior request exists for same block and request is also BusRd

We can also grab data. For this add 2-bits to each entry in the table:

1st bit : "want to grab response"

2nd bit : "I am original requestor"

For this cache the bits are 1st, 2nd = <1,0>

Non-original grabber must assert sharing line so that others will load in 'S' state rather than 'E' state

Original requester will have them as? <1,1>

Prior request BusRdX

If incompatible prior request, then wait for the earlier request to complete and later retry

Handling read miss

No: there is no prior request for this block in the request table

Issue request and watch-out for race conditions

Possible race condition=

(i) controller checks req-table. No conflicting requests

(ii) before it gets the bus, another conflicting req is granted the bus

(iv) this cache granted bus immediately next

There is a possibility of 2 conflicting requests on bus

Therefore this cache must check req-table entry just before its own slot on the bus

If it finds an entry then

(i) issues null-request (no-action) on bus to occupy the given slot and

(ii) withdraw from further arbitration until the conflicting request is completed

Effect on other processors

- Processor finally is successful in issuing BusRd. What should other cache-controller and memory do?

- Request is entered in the req-table of all cache controllers
- Cache controllers start checking their caches for the block
- Memory does not know if the block is dirty in some cache. So memory also starts fetching the block

- 3 scenarios:
  - (i) Cache has dirty block: Cache wins bus
  - (ii) Cache has dirty block: memory wins bus
  - (iii) Caches do not have dirty block: memory supplies data

Effect on other processors

- **(i) Cache has dirty block: Cache wins bus**

- One cache has dirty block and wins bus arbitration to send the block
- Memory and other caches load this new updated block
- Seeing this block response memory aborts its fetch

- If some cache controller has not completed snooping by the time response goes on bus this cache raises the inhibit-line and hence response transaction is extended
  - (this can happen if the response comes immediately in next cycle to the end of request phase...remember that the snoops are completed by extending the ACK cycle in request phase)

- Memory also takes the new bock. In case buffer is full at memory, flow control NACK is sent, and the controller having dirty block will retry the response transaction later

Effect on other processors

- **(ii) Cache has dirty block: memory wins bus**

If memory wins bus before the cache with dirty block has completed snoop and acquired bus

This cache controller will (i) assert inhitbit-line, (ii) complete snoop (iii) assert dirty line (iv) release inhibit line

Memory sees dirty-line=1 and does not put data on bus

i.e. response is postponed by inhibit signal and then dirty=1 cancels this memory response

Cache with dirty block will later acquire bus and send data

Effect on other processors

- **(iii) Caches do not have dirty block: memory supplies data**

No cache has dirty block

Memory acquires bus to send response

If cache snoops not complete, inhibit=1

Memory does not send data as long as inhibit=1

Once inhibit=0, data sent by memory, taken by caches

In this system, for shared block, cache-to-cache transfer is NOT done. Memory gives data

Handling Write Miss

- Send BusRdX transaction
- All actions same as in Read-miss

- When another cache sends dirty block, memory does not take it, since it is again going to change
- Also, no other cache grabs it
- In case block is valid-shared and needs to write, then send BusUpgr transaction
  - This needs no response
- If another processor was just about to issue a BusUpgr for the same block, it will now need to convert its request to BusRdX as in the case of atomic bus

Proving Write Serialisation

- In split transaciton bus, the bus-order is determined by order of requests appearing on the bus
- At end of response phase, they are committed for visibility in that order
- **Read X -> Write X**
  - This write-X should not affect the value returned by the read transaction
  - Easy as conflicting requests not allowed on bus
  - Response for Read-X precedes the request write-X on bus
- **Write X -> Read X**
  - Similarly, this read-X that follows the write-X transaction will not return the old value. As conflicting transactions not allowed. Read-X can take place only after response of Write-X has completed
- BusRdX and BusUpgr : pending Inv in queue. Therefore before every Read from Processor check queue.

Alternative design choices

(1) In-order Response instead of out-of-order

(2) Easy handling of Conflicting requests
Req-1: P1 write-X and Req-2: P2 write-X

In-order response

- Here a FIFO request table suffices instead of a fully-associative table
- Put request in FIFO when it actually appears on the bus
- As dirty cache has to give data now, and that the response cannot be postpone
  - The dirty cache raises the inhibit line till the data is ready to be supplied
  - If others have not completed their snoops, they raise the inhibit line
- Performance issue in case of interleaved memory
  - Ex: access order A -> B -> C.
  - A,B : Bank-1, C: Bank-2
  - A and C early, B-late
  - But C has to wait for B to complete

Handling conflicting req

- As we have FIFO order of response, we can allow conflicting requests, because responses are in-order. Still some care is required:
- Two BusRdX appear on bus, one after another for same block
  - Req-1: P1 write-X and Req-2: P2 write-X
- Controller P2 invalidates its block after seeing req-1
- P1 sees req-2 before response of req-1
- But controller P1 does not invalidate its block on seeing req-2
  - As the block is in flight and its own write needs to be performed and the data has to be flushed/invalidated

- Out-of-order response, not known which response will appear first
- With In-order it is known which response comes first. Therefore optimise
- Seeing req-2, P1 does not inv block
- P1 makes note that req-2 is pending
- When P1 gets response for req-1, it updates the block, short-cuts block back on bus, inv its copy (this reduces ping-pong latency under write-write false sharing)

27_Sep_2021 - Goog

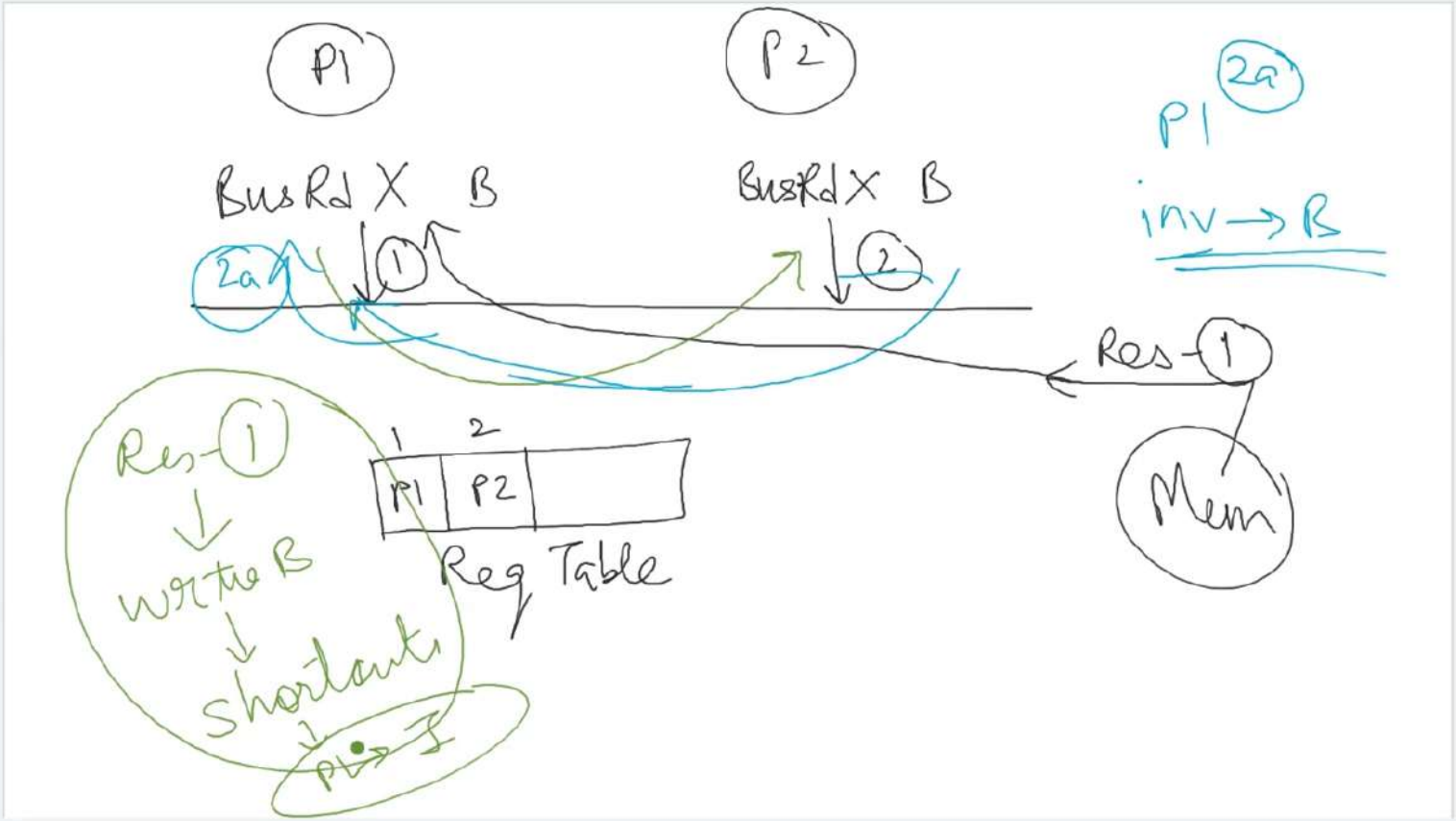
jamboard.google.com/d/1AIogDeB2COtifaU930L4zY8balv-UOA_HbR3-LKr-L4/viewer?f=0

27_Sep_2021

1/1

Share

Set background | Clear frame



100% Mon, September 27, 15:30

+5 | AM | AS | MN MUNINDRA NAIK | AS ASWATHY N S | Hemangee Kalpesh Kapoor | A AMIT | YK YOGESH KUMAR | NARESH BHARASAGAR | Hemangee Kalpesh Kapoor

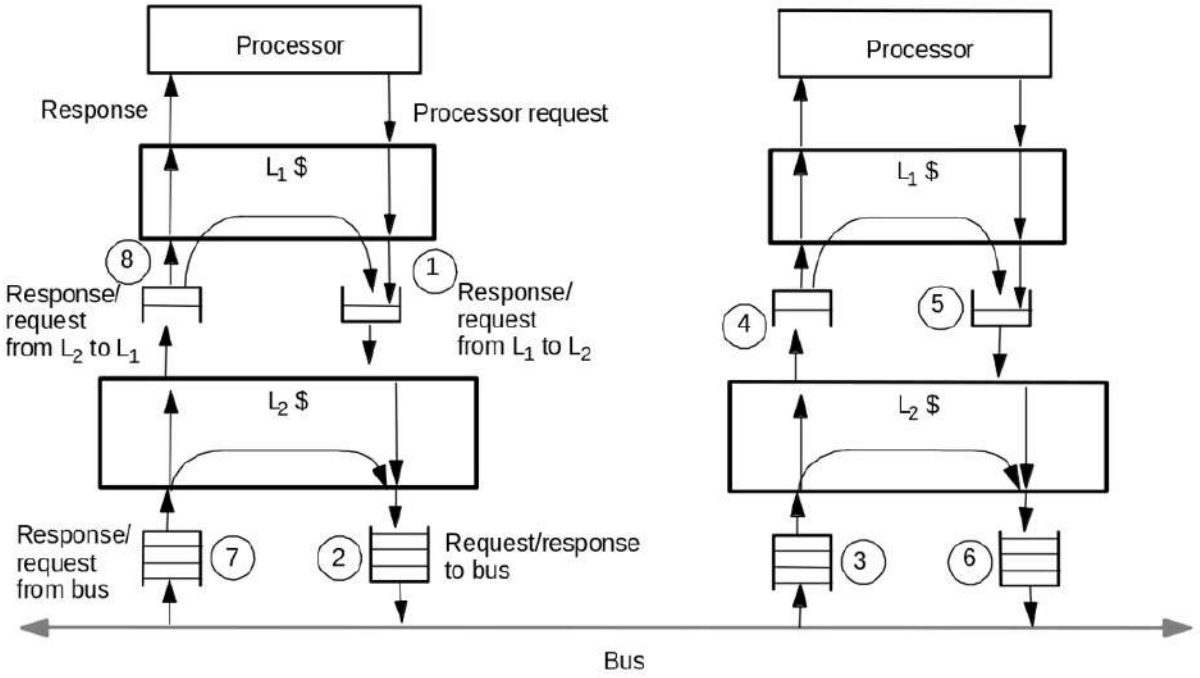
Split transaction bus
+ multi level cache

Multi-level Caches with Split Transaction bus

- Key new problem: many cycles to propagate through hierarchy
- Must let other transactions propagate up and down the hierarchy
  - To maintain high bandwidth and allow individual units to operate at their own rates, we keep queues between levels
  - However, these queues introduce deadlock and serialisation problems

Multi-level Caches with Split Transaction bus

Read by first processor satisfied by second processor



Deadlock considerations

- Fetch deadlock
  - In a two phase request-response protocol, an entity attempting to issue its request, also needs to service incoming transactions. Else deadlock
  - To avoid this
    - Must buffer incoming requests/responses while there are outstanding requests
    - Let total processors=P, Buffer space=?
    - One outstanding request per processor + one reply = buffer of (P+1) size.
      - As in worst case all processors send request to one cache
    - If buffer smaller than this then may need to NACK. Will need priority mechanism in bus arbiter to ensure progress
- Buffer deadlock

Deadlock considerations

- Fetch deadlock
  - In a two phase request-response protocol, an entity attempting to issue its request, also needs to service incoming transactions. Else deadlock
  - To avoid this
    - Must buffer incoming requests/responses while there are outstanding requests
    - Let total processors=P, Buffer space=?
    - One outstanding request per processor + one reply = buffer of (P+1) size.
      - As in worst case all processors send request to one cache
    - If buffer smaller than this then may need to NACK. Will need priority mechanism in bus arbiter to ensure progress
- Buffer deadlock

Deadlock considerations

- Buffer deadlock
- Scenario:
  - L1 to L2 queue filled with read requests, waiting for response from L2
  - L2 to L1 queue filled with read requests, waiting for response from L1
  - Causing deadlock
- Latter (second) condition occurs only when cache closer to processor (i.e. the highest level) is a write-back cache. Else incoming requests from bus/L2 will not require response from L1
  - Solution: provide enough buffering. Requires more hardware real-estate
  - Each request may need 2 outgoing buffer entries: 1 for the request and 1 for the write-back that it may generate
  - With large number of outstanding transactions allowed, the incoming buffers will also need more entries
  - Therefore keep limited entries and use standard deadlock avoidance methods
    - Optimisations exists, in Ch-7. Not discussed here

Buffer deadlock ..

- If outstanding bus transactions (allowed) are smaller than outstanding cache requests (i.e. cache misses)

  # bus transaction < # cache miss

- Then response from cache must get on the bus before new requests from it are allowed
- Queues may need to support by-passing
  - Support responses bypassing requests
- Ex:
  - Incoming requests to cache = 10, Caches needs to send responses = 10
  - Caches sends out requests = 'n' (n>10)
  - Buffer required = 10 + n + c
  - Buffer may become full and FIFO does not allow response to go out
  - Therefore, By-pass response out to the bus

Multiple outstanding Requests from the Processor

Multiple outstanding processor requests

- So far assumed only 1 request per processor
  - Not true of modern processors
- Danger: operations from same processor can complete out-of-order
- Ex. of outstanding references: write-buffer(s) holds many outstanding write to next level
- Until a write is serialised on the bus it must not be made visible to others, else it will violate coherence and write serialisation
  - One possibility is to write into the local cache but do not make it available until exclusive ownership is obtained (i.e. this cache will not respond to requests to this block)
  - Other option is to keep all the writes in the write-buffer and move them to the cache only after getting exclusive ownership

Multiple outstanding processor requests

- Processor sends several writes in rapid succession to wr-buffer without stalling
- Uniprocessor is ok: as long as reads check wr-buffer
- But in multiprocessor, a processor cannot be allowed to proceed until it has exclusive access to block, i.e. not until the BusRdX transaction is put on the bus and hence serialised
- However, there are special cases where this can be allowed
  - Sequence of writes to block in 'M' state is allowed. Here the cache must process all these writes from the wr-buffer before servicing requests from the bus side for this block
  - Multiple write to same block allowed if no other memory operation for the block on this processor is there in program order
    - Allow the write until the bus access is gained
    - Entire sequence of writes visible at once

Multiple outstanding processor requests

- Question: who should wait for serialisation and consistency when we allow processor to proceed past outstanding writes
  - i.e. not allow next operation until previous completes
  - Processor wait is in-efficient (as it will nullifiy the benefits of write-buffers and out-of-order processing)
  - Instead make the wr-buffer or re-order buffers (in OoO processor) + controllers manage this
    - i.e. the buffers take charge of making sure that the write operation is not visible to the memory and interconnect systems (i.e. they are not made visible to external system)
    - Also (local) read operations are not allowed to complete out of program order wrt the writes pending in these buffers
- Other requirements: to have multiple outstanding requests, cache must allow multiple misses outstanding
  - We need lockup-free cache instead of blocking cache
    - Blocking cache allows only 1 miss

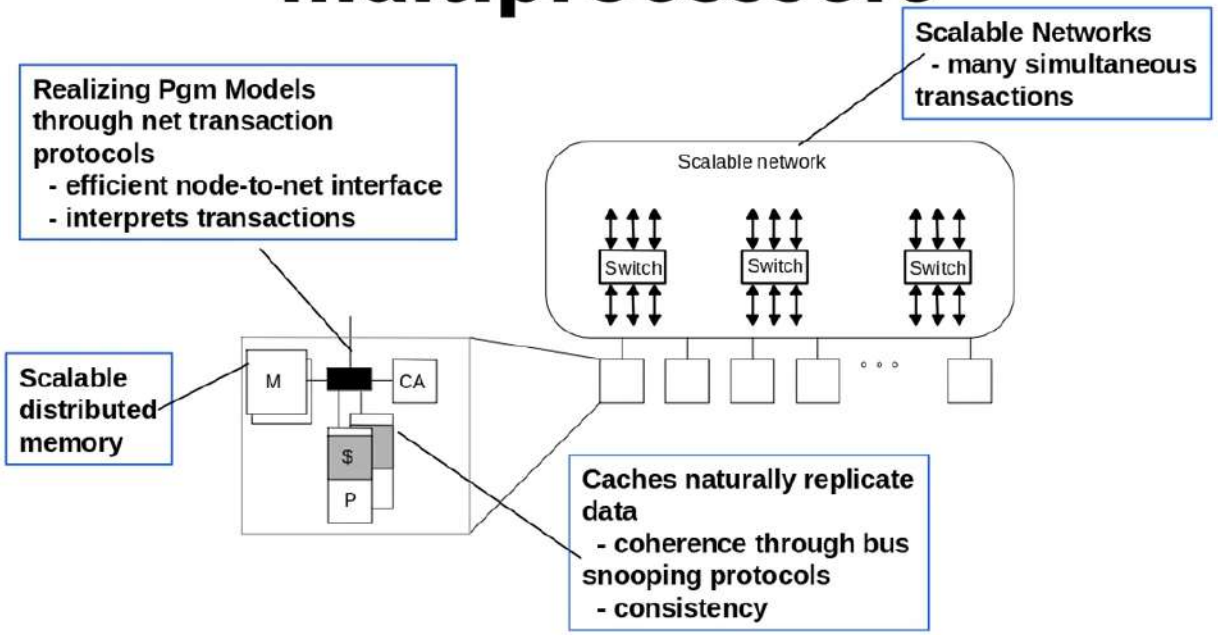
Directory based Cache Coherence

Topic-4
Chapter-8

Why snooping does not scale?

- Limitations of snooping
  - Broadcasting uses lot of bus bandwidth
  - Snooping every transaction uses lots of controller bandwidth
- Snooping is great for small to medium size machines
  - Largest current snooping system has 128 processors
- Snooping hits bottleneck for large machine
- How to overcome bottleneck?
  - Get rid of protocol that requires: Broadcasting / Logical bus

Large scale shared memory multiprocessors



- 100s to 1000s of nodes with single shared physical address space
- General purpose interconnection network
  - Still have cache coherence protocol
  - Use message instead of bus transaction
  - no broadcast or single point of order

Cache coherent system must:

- Provide set of state, transition diagram and actions
- Manage coherence protocol
  - (0) when to invoke
  - (a) Find state in all cache blocks
    - To determine whether to communicate with other cached copies
  - (b) Locate other copies
  - (c) Communicate with these copies (inv/upd)
- (0) is done same for all systems
  - State of line kept in cache
  - Invoke protocol on access fault (i.e. miss) :: I -> S, M or S -> M
- Different approaches are distinguished by (a), (b), (c)

Bus based coherence

- All (a), (b), (c) done through broadcast on bus
  - Faulting processor sends out a search
  - Others respond to search and take necessary action
- Could do it in scalable network too
  - Broadcast to all processors and let them respond
- Conceptually simple, but broadcast does not scale with p
  - On bus, bandwidth does not scale
  - On scalable network, every fault leads to atleast p network transactions
- Scalable coherence:
  - Can have same cache state and state transition diagram
  - Different mechanism to manage protocol

One Approach: Hierarchical Snooping

- Extend snooping approach. Have a hierarchy of broadcast media
  - Tree of bus or rings
  - Processors are in bus or ring based multiprocessors at the leaves
  - Snoop both busses and propagate relevant transactions
  - Main memory can be centralised at root or distributed among leaves
- Issues (a)-(c) handled similar to bus, but not full broadcast
  - Faulting processor sends out search on the bus
  - Propagates up and down the hierarchy based on snoop results
- Hope is that most of the time request is not propagated very far
- Problems:
  - High latency: multiple levels and snoop/lookup at every level
  - Bandwidth bottleneck at root
- Not popular today

Scalable solution: Directories

- Avoid broadcast requests to all nodes on a miss
- Every memory block has associated directory information
  - Keeps track of copies of cached block and their states
  - On a miss, find directory entry, send message to cached copies if necessary
  - In scalable network, communication with directory is through network transactions
- Maintain directory of which nodes have cached copies of the block (directory controller + directory state)
  - On a cache miss, cache controller sends message to directory
  - Directory determines what (if any) protocol action is required – e.g. inv of shared nodes
  - Directory waits for protocol actions to finish and then responds to the original request