Lecture 8: Snooping and Directory Protocols

- Topics: split-transaction implementation details, directory implementations (memory- and cache-based)

Split Transaction Bus

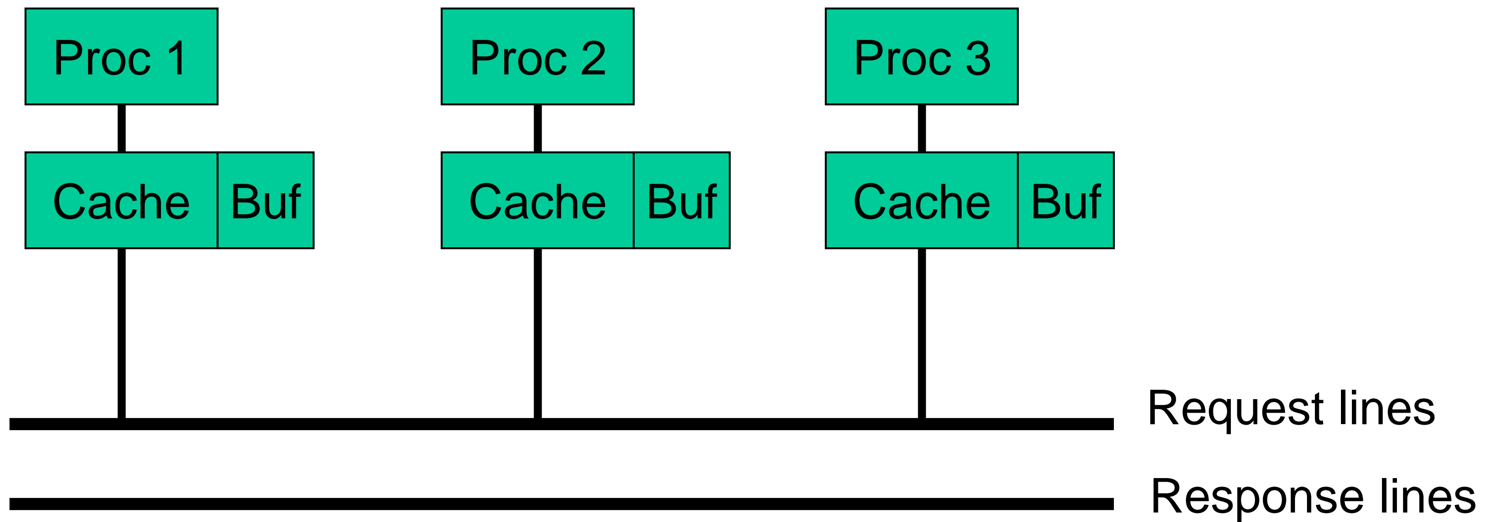
- So far, we have assumed that a coherence operation (request, snoops, responses, update) happens atomically

- 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

- Could be a 3-stage pipeline: request/snoop/response or (much simpler) 2-stage pipeline: request-snoop/response (note that the response is slowest and needs to be hidden)

- Buffers track the outstanding transactions; buffers are identical in each core; an entry is freed when the response is seen; the next operation uses any free entry; every bus operation carries the buffer entry number as a tag

- Must check the buffer before broadcasting a new operation; must ensure only one outstanding operation per block

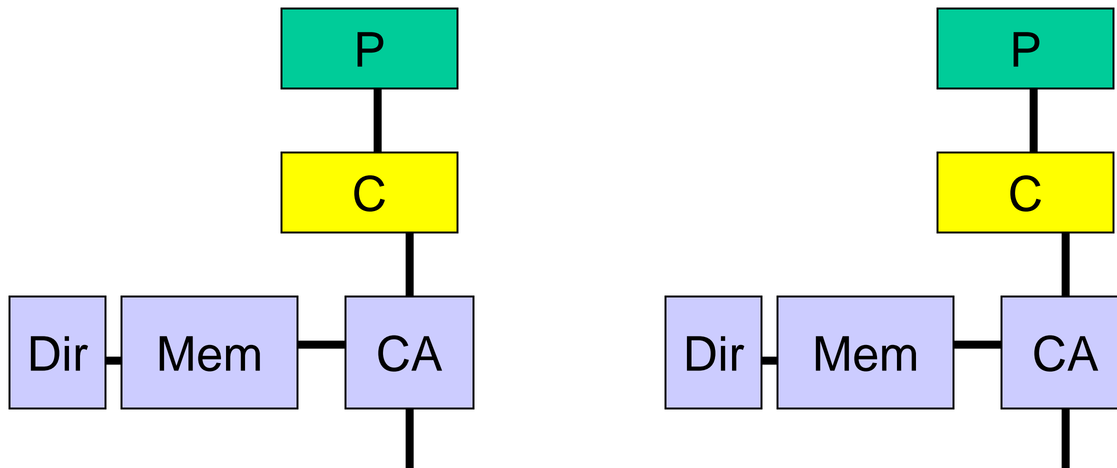
- What determines the write order – requests or responses?

Design Issues II

- What happens if processor-A is arbitrating for the bus and witnesses another bus transaction for the same address or same buffer entry?
- What if processor-A was trying to do an upgrade?
- What if processor-A was trying to do a read and there is already a matching read in the request table?
- Processor-cache handshake: after acquiring the block in excl state, the processor must complete the write before handing the block to other writers; else, there's a livelock

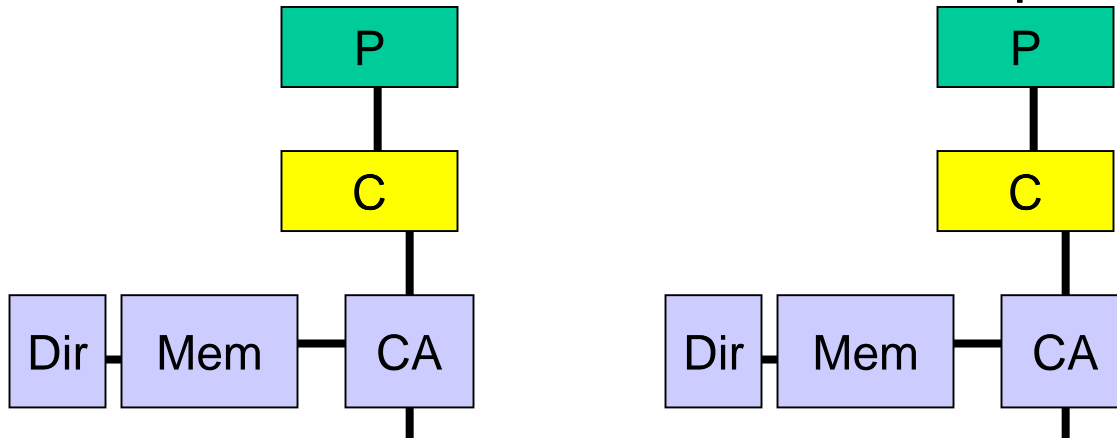
Directory-Based Protocol

- For each block, there is a centralized "directory" that maintains the state of the block in different caches
- The directory is co-located with the corresponding memory
- Requests and replies on the interconnect are no longer seen by everyone – the directory serializes writes



Definitions

- Home node: the node that stores memory and directory state for the cache block in question
- Dirty node: the node that has a cache copy in modified state
- Owner node: the node responsible for supplying data (usually either the home or dirty node)
- Also, exclusive node, local node, requesting node, etc.



Directory Organizations

- Centralized Directory: one fixed location – bottleneck!
- Flat Directories: directory info is in a fixed place, determined by examining the address – can be further categorized as memory-based or cache-based
- Hierarchical Directories: the processors are organized as a logical tree structure and each parent keeps track of which of its immediate children has a copy of the block – more searching, can exploit locality

Flat Memory-Based Directories

- Directory is associated with memory and stores info for all cached copies

- A presence vector stores a bit for every processor, for every memory block – the overhead is a function of memory/block size and #processors

- Reducing directory overhead:

Flat Memory-Based Directories

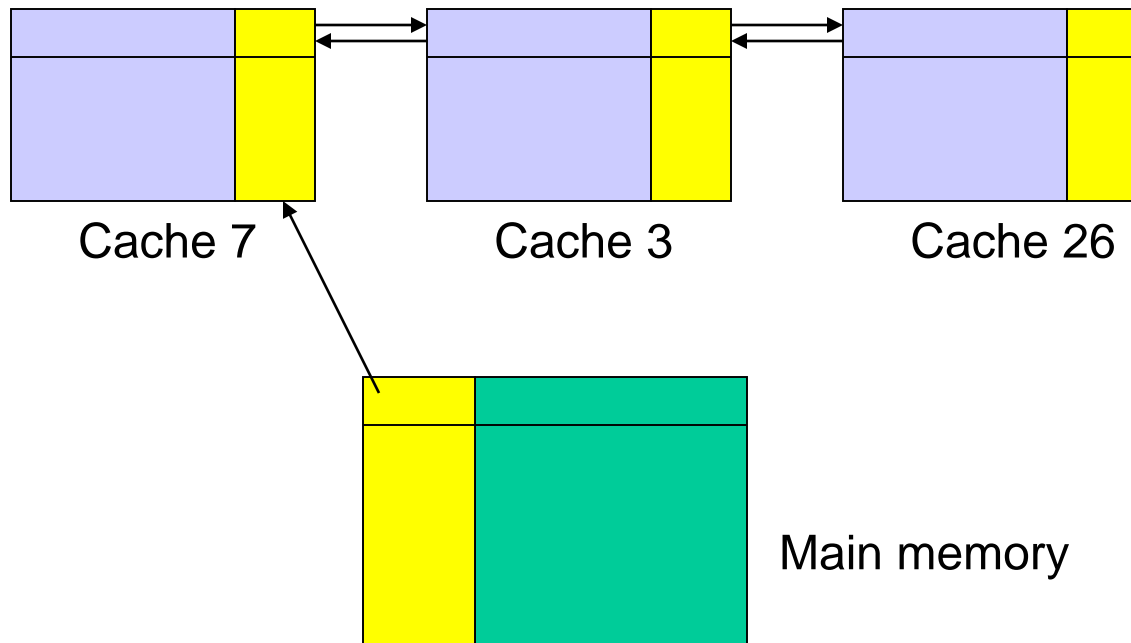
- Directory is associated with memory and stores info for all cache copies

- A presence vector stores a bit for every processor, for every memory block – the overhead is a function of memory/block size and #processors

- Reducing directory overhead:
  - Width: pointers (keep track of processor ids of sharers) (need overflow strategy), organize processors into clusters
  - Height: increase block size, track info only for blocks that are cached (note: cache size << memory size)

Flat Cache-Based Directories

- The directory at the memory home node only stores a pointer to the first cached copy – the caches store pointers to the next and previous sharers (a doubly linked list)



Flat Cache-Based Directories

- The directory at the memory home node only stores a pointer to the first cached copy – the caches store pointers to the next and previous sharers (a doubly linked list)

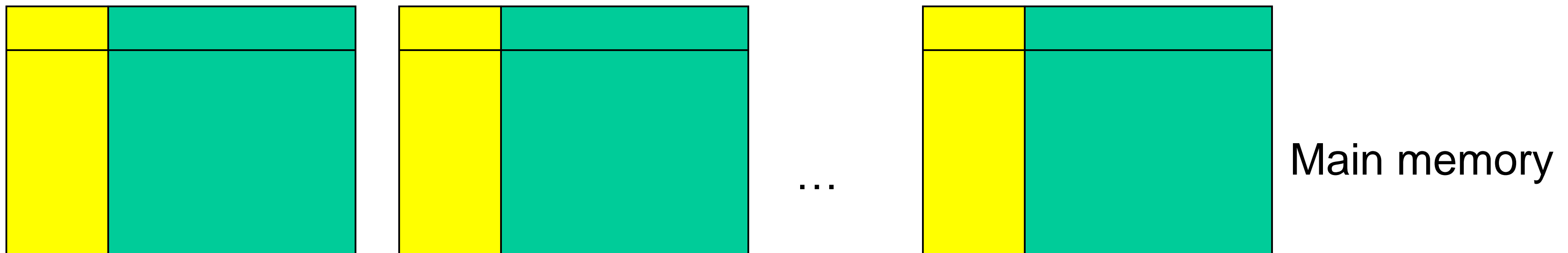
- Potentially lower storage, no bottleneck for network traffic

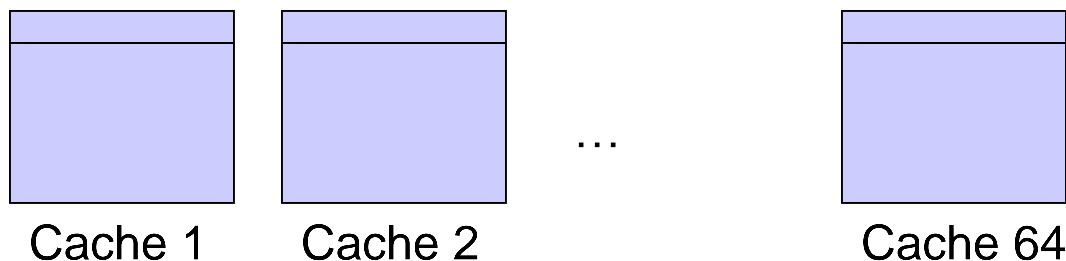
- Invalidates are now serialized (takes longer to acquire exclusive access), replacements must update linked list, must handle race conditions while updating list

Flat Memory-Based Directories

Block size = 128 B
Memory in each node = 1 GB
Cache in each node = 1 MB

For 64 nodes and 64-bit directory,
Directory size = 4 GB
For 64 nodes and 12-bit directory,
Directory size = 0.75 GB



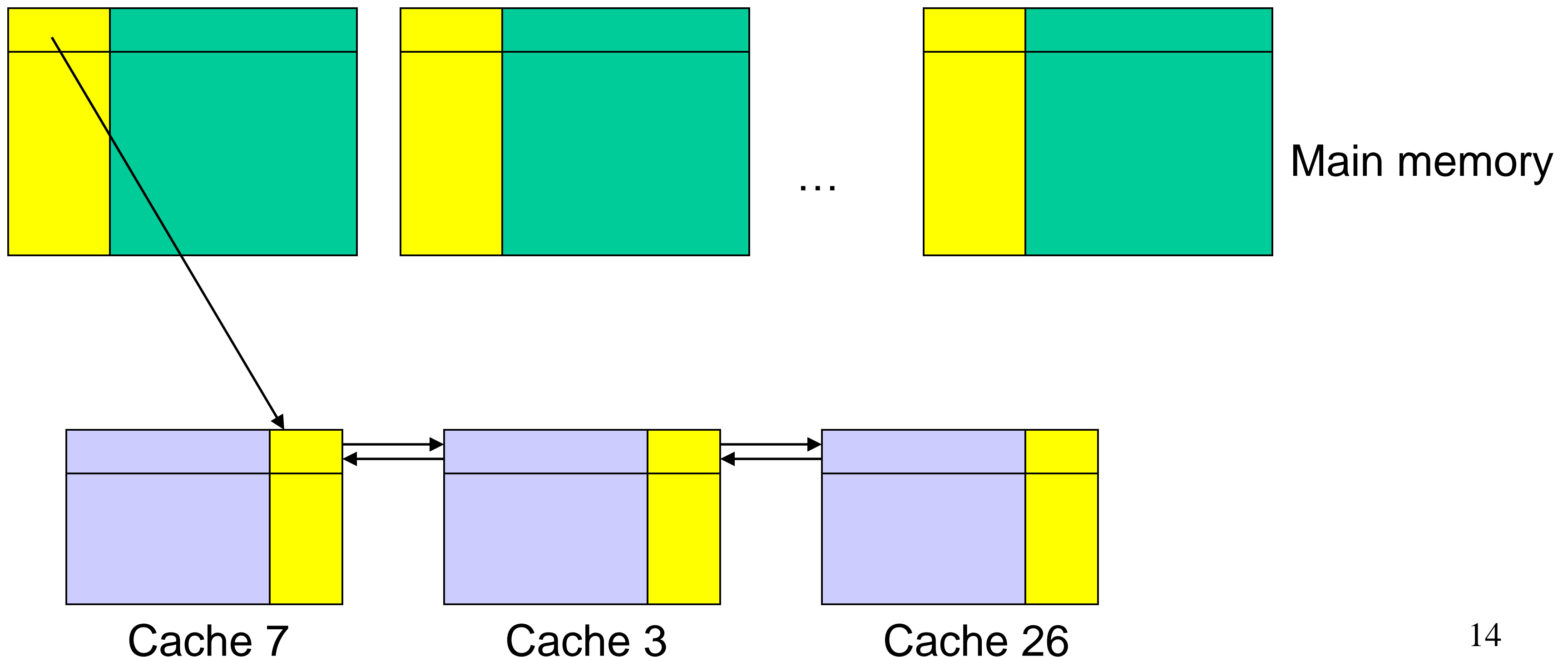


Flat Cache-Based Directories

Block size = 128 B
Memory in each node = 1 GB
Cache in each node = 1 MB

6-bit storage in DRAM for each block;
DRAM overhead = 0.375 GB

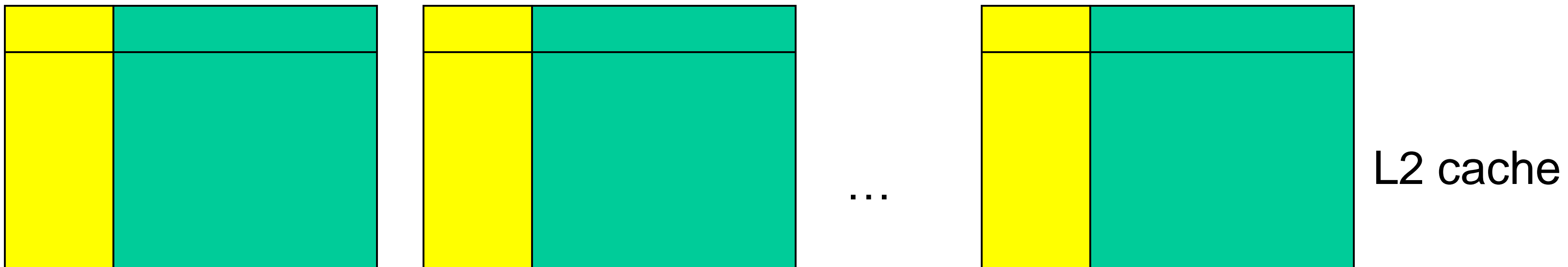
12-bit storage in SRAM for each block;
SRAM overhead = 0.75 MB

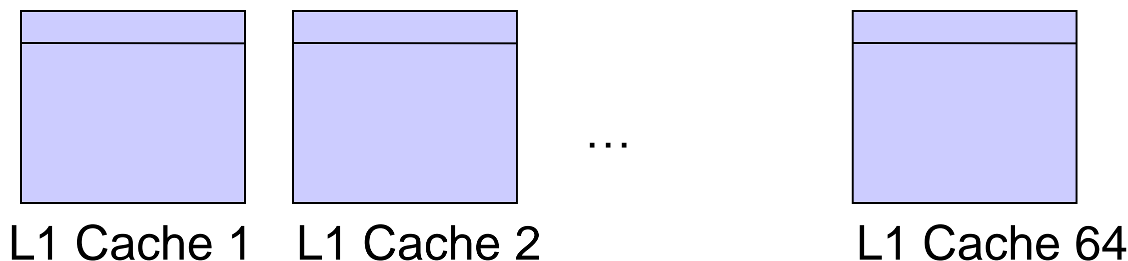


Flat Memory-Based Directories

Block size = 64 B
L3 cache in each node = 2 MB
L2 Cache in each node = 256 KB

For 64 nodes and 64-bit directory,
Directory size = 16 MB
For 64 nodes and 12-bit directory,
Directory size = 3 MB



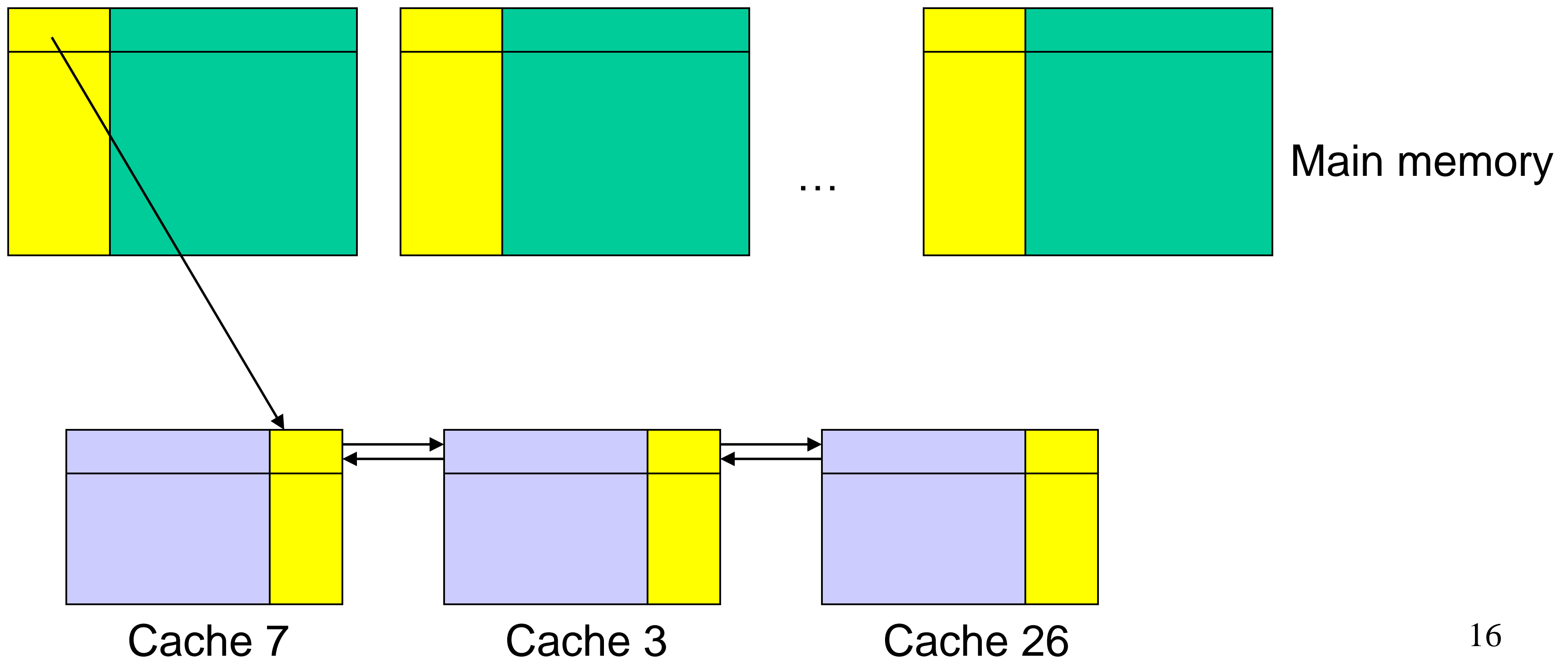


Flat Cache-Based Directories

Block size = 64 B
L3 cache in each node = 2 MB
L2 Cache in each node = 256 KB

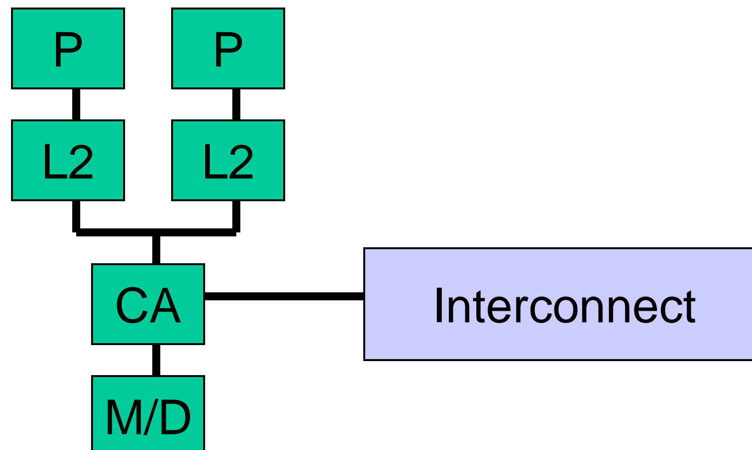
6-bit storage in L3 for each block;
L3 overhead = 1.5 MB

12-bit storage in L2 for each block;
L2 overhead = 384 KB



SGI Origin 2000

- Flat memory-based directory protocol
- Uses a bit vector directory representation
- Two processors per node – combining multiple processors in a node reduces cost



Directory Structure

- The system supports either a 16-bit or 64-bit directory (fixed cost); for small systems, the directory works as a full bit vector representation
- Seven states, of which 3 are stable
- For larger systems, a coarse vector is employed – each bit represents p/64 nodes
- State is maintained for each node, not each processor – the communication assist broadcasts requests to both processors

Handling Reads

- SGI Origin 2000 case study: directory states: 3 stable states, 3 busy states, and 1 poison state; cache states: invalid, shared, excl-clean, excl-modified

- When the home receives a read request, it looks up memory (speculative read) and directory in parallel

- Actions taken for each directory state:
  - shared or unowned: data is returned to requestor, state is changed to excl if there are no other sharers
  - busy: a NACK is sent to the requestor
  - exclusive: home is not the owner, request is fwded to owner, owner sends data to requestor and home

Inner Details of Handling the Read

- The block is in exclusive state – memory may or may not have a clean copy – it is speculatively read anyway
- The directory state is set to busy-exclusive and the presence vector is updated
- In addition to fwding the request to the owner, the memory copy is speculatively forwarded to the requestor
  - Case 1: excl-dirty: owner sends block to requestor and home, the speculatively sent data is over-written
  - Case 2: excl-clean: owner sends an ack (without data) to requestor and home, requestor waits for this ack before it moves on with speculatively sent data

Inner Details II

- Why did we send the block speculatively to the requestor if it does not save traffic or latency?
  - the R10K cache controller is programmed to not respond with data if it has a block in excl-clean state
  - when an excl-clean block is replaced from the cache, the directory need not be updated – hence, directory cannot rely on the owner to provide data and speculatively provides data on its own

Handling Write Requests

- The home node must invalidate all sharers and all invalidations must be acked (to the requestor), the requestor is informed of the number of invalidates to expect

- Actions taken for each state:
  - shared: invalidates are sent, state is changed to excl, data and num-sharers are sent to requestor, the requestor cannot continue until it receives all acks (Note: the directory does not maintain busy state, subsequent requests will be fwded to new owner and they must be buffered until the previous write has completed)

Handling Writes II

- Actions taken for each state:
  - unowned: if the request was an upgrade and not a read-exclusive, is there a problem?
  - exclusive: is there a problem if the request was an upgrade? In case of a read-exclusive: directory is set to busy, speculative reply is sent to requestor, invalidate is sent to owner, owner sends data to requestor (if dirty), and a “transfer of ownership” message (no data) to home to change out of busy
  - busy: the request is NACKed and the requestor must try again

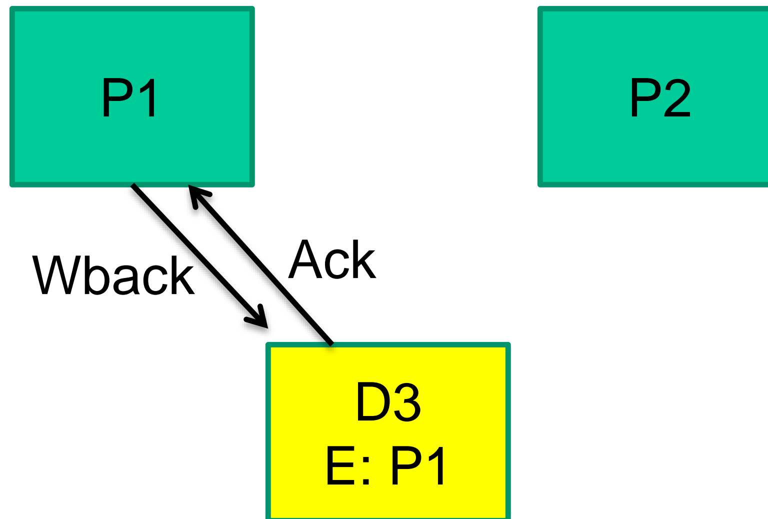
Handling Write-Back

- When a dirty block is replaced, a writeback is generated and the home sends back an ack

- Can the directory state be shared when a writeback is received by the directory?

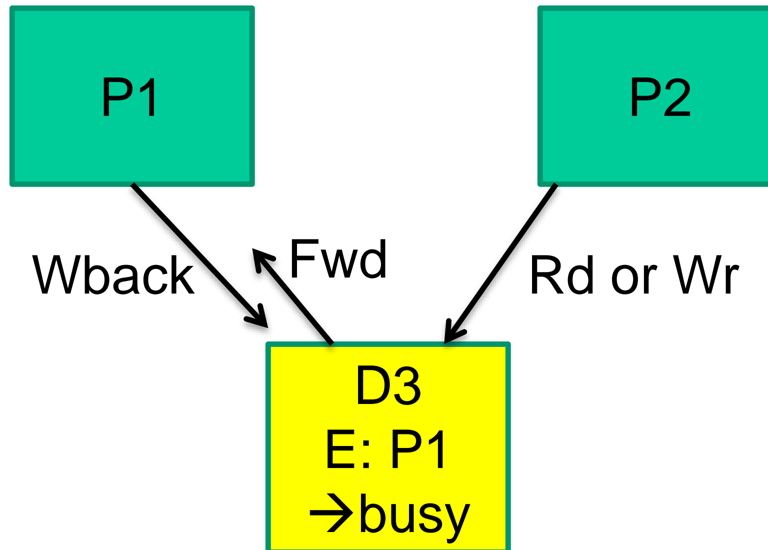
- Actions taken for each directory state:
  - exclusive: change directory state to unowned and send an ack
  - busy: a request and the writeback have crossed paths: the writeback changes directory state to shared or excl (depending on the busy state), memory is updated, and home sends data to requestor, the intervention request is dropped

Writeback Cases



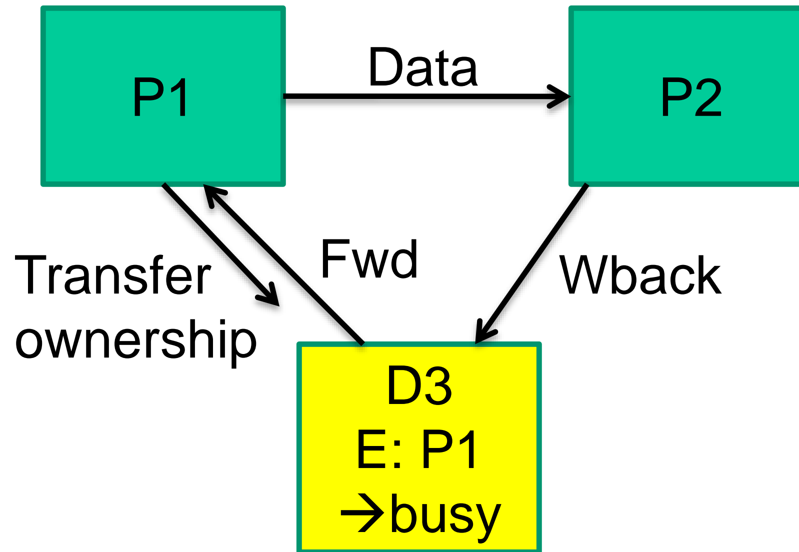
This is the “normal” case
D3 sends back an Ack

Writeback Cases



If someone else has the block in exclusive, D3 moves to busy
If Wback is received, D3 serves the requester
If we didn't use busy state when transitioning from E:P1 to E:P2,
    D3 may not have known who to service
    (since ownership may have been passed on to P3 and P4…)
    (although, this problem can be solved by NACKing the Wback
      and having P1 buffer its "strange" intervention requests…
      this could lead to other corner cases…)

Writeback Cases



If Wback is from new requester, D3 sends back a NACK
Floating unresolved messages are a problem
Alternatively, can accept the Wback and put D3 in some new busy state

Conclusion: could have got rid of busy state between E:P1 → E:P2, but with Wback ACK/NACK and other buffering
could have kept the busy state between E:P1 → E:P2, could have got rid of ACK/NACK, but need one new busy state

Title

- Bullet