Lecture 8: Snooping and Directory Protocols

• Topics: 4/5-state snooping protocols, split-transaction implementation details, directory implementations (memory- and cache-based)
Reporting Snoop Results

• In a multiprocessor, memory has to wait for the snoop result before it chooses to respond – need 3 wired-OR signals: (i) indicates that a cache has a copy, (ii) indicates that a cache has a modified copy, (iii) indicates that the snoop has not completed

• Ensuring timely snoops: the time to respond could be fixed or variable (with the third wired-OR signal)

• Tags are usually duplicated if they are frequently accessed by the processor (regular ld/sts) and the bus (snoops)
4 and 5 State Protocols

• Multiprocessors execute many single-threaded programs

• A read followed by a write will generate bus transactions to acquire the block in exclusive state even though there are no sharers (leads to MESI protocol)

• Also, to promote cache-to-cache sharing, a cache must be designated as the responder (leads to MOESI protocol)

• Note that we can optimize protocols by adding more states – increases design/verification complexity
MESI Protocol

• The new state is exclusive-clean – the cache can service read requests and no other cache has the same block

• When the processor attempts a write, the block is upgraded to exclusive-modified without generating a bus transaction

• When a processor makes a read request, it must detect if it has the only cached copy – the interconnect must include an additional signal that is asserted by each cache if it has a valid copy of the block

• When a block is evicted, a block may be exclusive-clean, but will not realize it
MOESI Protocol

- The first reader or the last writer are usually designated as the owner of a block.
- The owner is responsible for responding to requests from other caches.
- There is no need to update memory when a block transitions from M → S state.
- The block in O state is responsible for writing back a dirty block when it is evicted.
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)

![Diagram showing cache nodes and pointers]

- Cache 7
- Cache 3
- Cache 26

Main memory
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

Main memory

Cache 1  Cache 2  ...  Cache 64
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

Cache 7

Cache 3

Cache 26

Main memory
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

Cache 7  Cache 3  Cache 26
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
Title

- Bullet