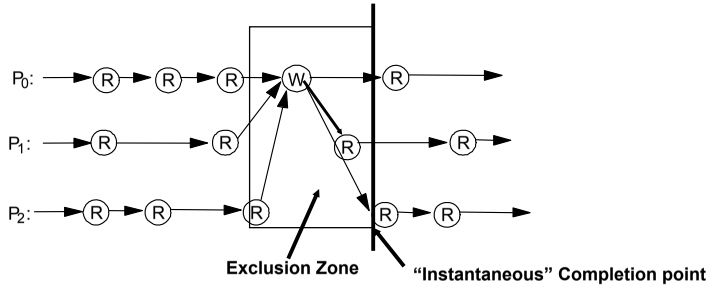
CS 258
Parallel Computer Architecture
Lecture 21

Directory Based Protocols

April 14, 2008
Prof John D. Kubiatowicz
http://www.cs.berkeley.edu/~kubitron/cs258

Recall Ordering: Scheurich and Dubois

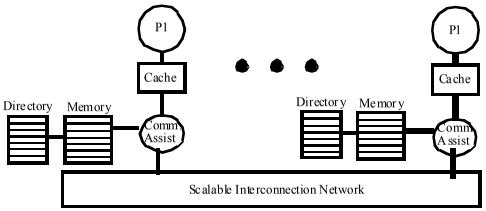


- Sufficient Conditions
  - every process issues mem operations in program order
  - after a write operation is issued, the issuing process waits for the write to complete before issuing next memory operation
  - after a read is issued, the issuing process waits for the read to complete and for the write whose value is being returned to complete (gloabaly) befor issuing its next operation

Terminology for Shared Memory

- UMA – Uniform Memory Access
  - Snoopy bus
  - Butterfly network
- NUMA – Non-uniform Memory Access
  - Directory Protocols
  - Hybrid Protocols
  - Etc.
- COMA – "Cache-Only" Memory Architecture
  - Hierarchy of buses
  - Directory-based (COMA – Flat)

Generic Distributed Mechanism: Directories



- Maintain state vector explicitly
  - associate with memory block
  - records state of block in each cache
- On miss, communicate with directory
  - determine location of cached copies
  - determine action to take
  - conduct protocol to maintain coherence

A Cache Coherent System Must:

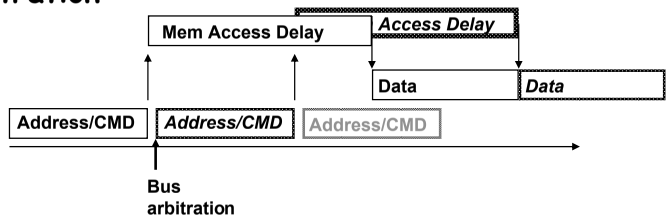
- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
  - (0) Determine when to invoke coherence protocol
  - (a) Find info about state of block in other caches to determine action
    - » whether need to communicate with other cached copies
  - (b) Locate the other copies
  - (c) Communicate with those copies (inval/update)
- (0) is done the same way on all systems
  - state of the line is maintained in the cache
  - protocol is invoked if an "access fault" occurs on the line
- Different approaches distinguished by (a) to (c)

Bus-based Coherence

- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a "search"
  - others respond to the search probe and take necessary action
- Could do it in scalable network too
  - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn't scale with p
  - on bus, bus bandwidth doesn't scale
  - on scalable network, every fault leads to at least p network transactions
- Scalable coherence:
  - can have same cache states and state transition diagram
  - different mechanisms to manage protocol

Split-Transaction Bus

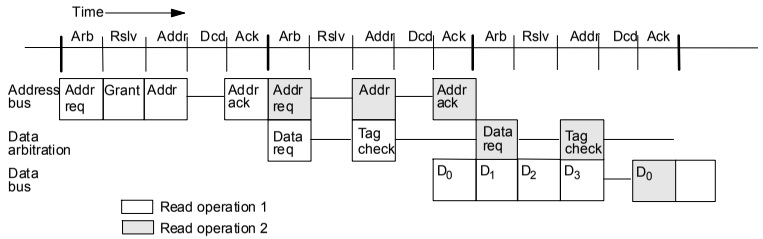
- Split bus transaction into request and response sub-transactions
  - Separate arbitration for each phase
- Other transactions may intervene
  - Improves bandwidth dramatically
  - Response is matched to request
  - Buffering between bus and cache controllers
- Reduce serialization down to the actual bus arbitration



Example (based on SGI Challenge)

- No conflicting requests for same block allowed on bus
  - 8 outstanding requests total, makes conflict detection tractable
- Flow-control through negative acknowledgement (NACK)
  - NACK as soon as request appears on bus, requestor retries
  - Separate command (incl. NACK) + address and tag + data buses
- Responses may be in different order than requests
  - Order of transactions determined by requests
  - Snoop results presented on bus with response
- Look at
  - Bus design, and how requests and responses are matched
  - Snoop results and handling conflicting requests
  - Flow control
  - Path of a request through the system

Bus Design (continued)

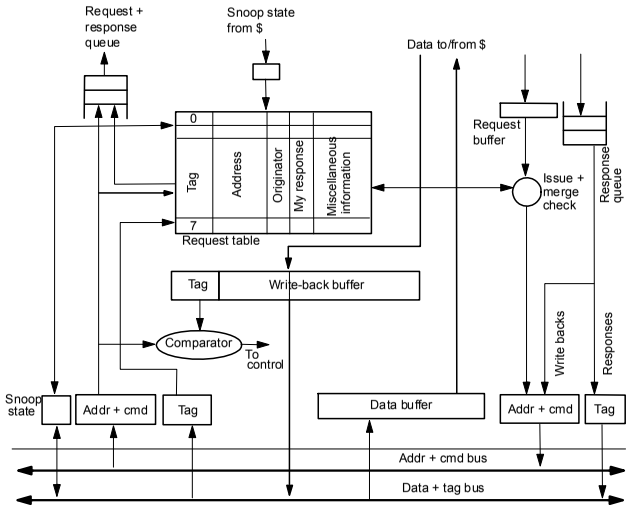


- Each of request and response phase is 5 bus cycles
  - Response: 4 cycles for data (128 bytes, 256-bit bus), 1 turnaround
  - Request phase: arbitration, resolution, address, decode, ack
  - Request-response transaction takes 3 or more of these
- Cache tags looked up in decode; extend ack cycle if not possible
  - Determine who will respond, if any
  - Actual response comes later, with re-arbitration
- Write-backs only request phase : arbitrate both data+addr buses
- Upgrades have only request part; ack'ed by bus on grant (commit)

Bus Design (continued)

- Tracking outstanding requests and matching responses
  - Eight-entry "request table" in each cache controller
  - New request on bus added to all at same index, determined by tag
  - Entry holds address, request type, state in that cache (if determined already), ...
  - All entries checked on bus or processor accesses for match, so fully associative
  - Entry freed when response appears, so tag can be reassigned by bus

Bus Interface with Request Table



Handling a Read Miss

- Need to issue BusRd
- First check request table. If hit:
  - If prior request exists for same block, want to grab data too!
    - » "want to grab response" bit
    - » "original requestor" bit
      - non-original grabber must assert sharing line so others will load in S rather than E state
  - If prior request incompatible with BusRd (e.g. BusRdX)
    - » wait for it to complete and retry (processor-side controller)
  - If no prior request, issue request and watch out for race conditions
    - » conflicting request may win arbitration before this one, but this one receives bus grant before conflict is apparent
      - watch for conflicting request in slot before own, degrade request to "no action" and withdraw till conflicting request satisfied

Upon Issuing the BusRd Request

- All processors enter request into table, snoop for request in cache
- Memory starts fetching block
- 1. Cache with dirty block responds before memory ready
  - Memory aborts on seeing response
  - Waiters grab data
    - » some may assert inhibit to extend response phase till done snooping
    - » memory must accept response as WB (might even have to NACK)
- 2. Memory responds before cache with dirty block
  - Cache with dirty block asserts inhibit line till done with snoop
  - When done, asserts dirty, causing memory to cancel response
  - Cache with dirty issues response, arbitrating for bus
- 3. No dirty block: memory responds when inhibit line released
  - Assume cache-to-cache sharing not used (for non-modified data)

Handling a Write Miss

- Similar to read miss, except:
  - Generate BusRdX
  - Main memory does not sink response since will be modified again
  - No other processor can grab the data
- If block present in shared state, issue BusUpgr instead
  - No response needed
  - If another processor was going to issue BusUpgr, changes to BusRdX as with atomic bus

Write Serialization

- With split-transaction buses, usually bus order is determined by order of *requests* appearing on bus
  - actually, the ack phase, since requests may be NACKed
  - by end of this phase, they are committed for visibility in order
- A write that follows a read transaction to the same location should not be able to affect the value returned by that read
  - Easy in this case, since conflicting requests not allowed
  - Read response precedes write request on bus
- Similarly, a read that follows a write transaction won't return old value

Administrivia

- Class this Wednesday is a guest lecture and *is in 3108 Etcheverry Hall from 2:30-4pm*
  - Anant Agarwal will talk about Tilera
- 3 ½ weeks left with the project!
  - Hopefully you are all well on your way
  - See me immediately if you are having trouble

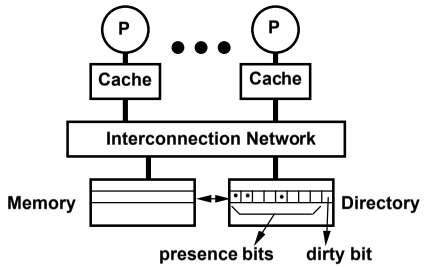
Scalable Approach: Hierarchical Snooping

- Extend snooping approach: hierarchy of broadcast media
  - tree of buses or rings (DDM,KSR-1)
  - processors are in the bus- or ring-based multiprocessors at the leaves
  - parents and children connected by two-way snoopy interfaces
    - » snoop both buses and propagate relevant transactions
  - main memory may be centralized at root or distributed among leaves
- Issues (a) - (c) handled similarly to bus, but not full broadcast
  - faulting processor sends out "search" bus transaction on its bus
  - propagates up and down hierarchy based on snoop results
- Problems:
  - high latency: multiple levels, and snoop/lookup at every level
  - bandwidth bottleneck at root
- Not popular today

Scalable Approach: Directories

- Every memory block has associated directory information
  - keeps track of copies of cached blocks and their states
  - on a miss, find directory entry, look it up, and communicate only with the nodes that have copies if necessary
  - in scalable networks, communication with directory and copies is through network transactions
- Many alternatives for organizing directory information

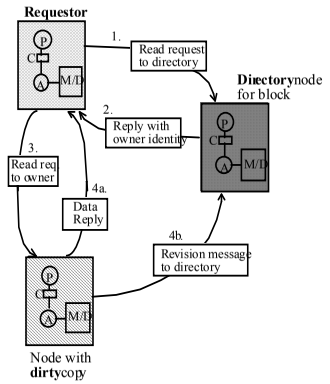
Basic Operation of Directory



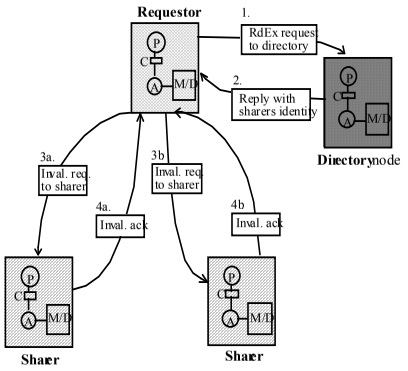
- k processors.
- With each cache-block in memory: k presence-bits, 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

- Read from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON; }
  - if dirty-bit ON then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i;}
- Write to main memory by processor i:
  - If dirty-bit OFF then { supply data to i; send invalidations to all caches that have the block; turn dirty-bit ON; turn p[i] ON; ... }
  - ...

Basic Directory Transactions



(a) Read miss to a block in dirty state

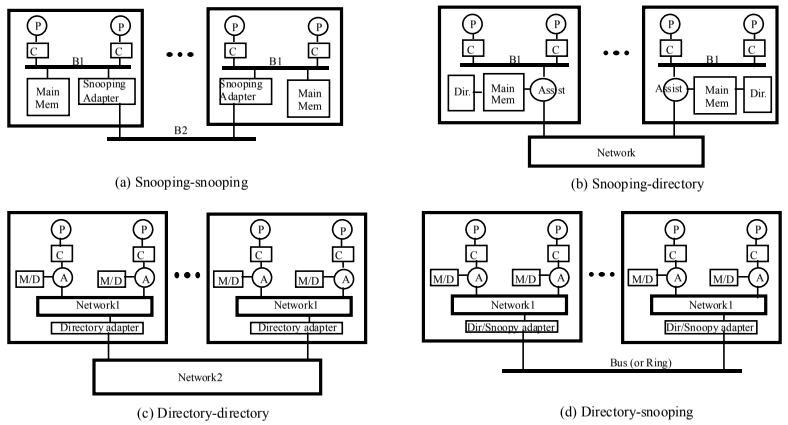


(b) Write miss to a block with two sharers

A Popular Middle Ground

- Two-level "hierarchy"
- Individual nodes are multiprocessors, connected non-hiearchically
  - e.g. mesh of SMPs
- Coherence across nodes is directory-based
  - directory keeps track of nodes, not individual processors
- Coherence within nodes is snooping or directory
  - orthogonal, but needs a good interface of functionality
- Examples:
  - Convex Exemplar: directory-directory
  - Sequent, Data General, HAL: directory-snoopy
- SMP on a chip?

Example Two-level Hierarchies



(a) Snooping-snooping (b) Snooping-directory

(c) Directory-directory (d) Directory-snooping

Scaling Issues

- memory and directory bandwidth
  - Centralized directory is bandwidth bottleneck, just like centralized memory
  - How to maintain directory information in distributed way?
- performance characteristics
  - traffic: no. of network transactions each time protocol is invoked
  - latency = no. of network transactions in critical path
- directory storage requirements
  - Number of presence bits grows as the number of processors
- How directory is organized affects all these, performance at a target scale, as well as coherence management issues

Insight into Directory Requirements

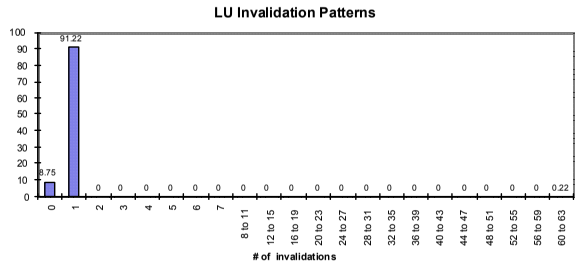
- If most misses involve O(P) transactions, might as well broadcast!

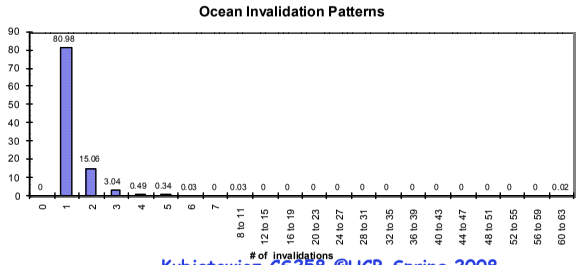
=> Study Inherent program characteristics:

  - frequency of write misses?
  - how many sharers on a write miss
  - how these scale

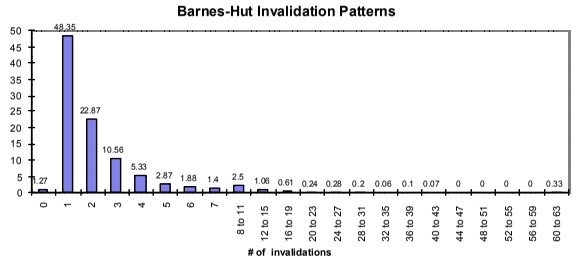
- Also provides insight into how to organize and store directory information

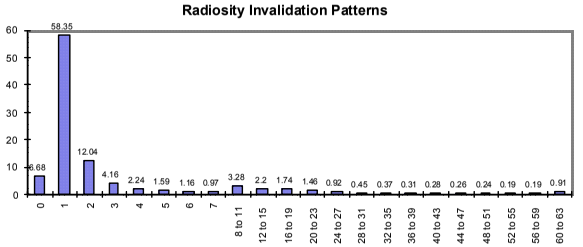
Cache Invalidation Patterns





Cache Invalidation Patterns

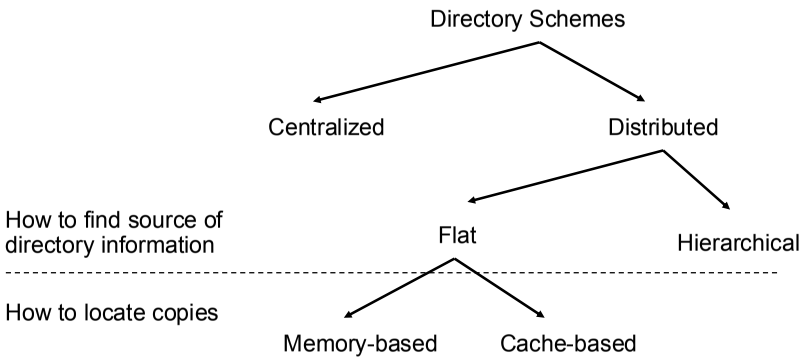




Sharing Patterns Summary

- Generally, few sharers at a write, scales slowly with P
  - Code and read-only objects (e.g, scene data in Raytrace)
    - » no problems as rarely written
  - Migratory objects (e.g., cost array cells in LocusRoute)
    - » even as # of PEs scale, only 1-2 invalidations
  - Mostly-read objects (e.g., root of tree in Barnes)
    - » invalidations are large but infrequent, so little impact on performance
  - Frequently read/written objects (e.g., task queues)
    - » invalidations usually remain small, though frequent
  - Synchronization objects
    - » low-contention locks result in small invalidations
    - » high-contention locks need special support (SW trees, queueing locks)
- Implies directories very useful in containing traffic
  - if organized properly, traffic and latency shouldn't scale too badly
- Suggests techniques to reduce storage overhead

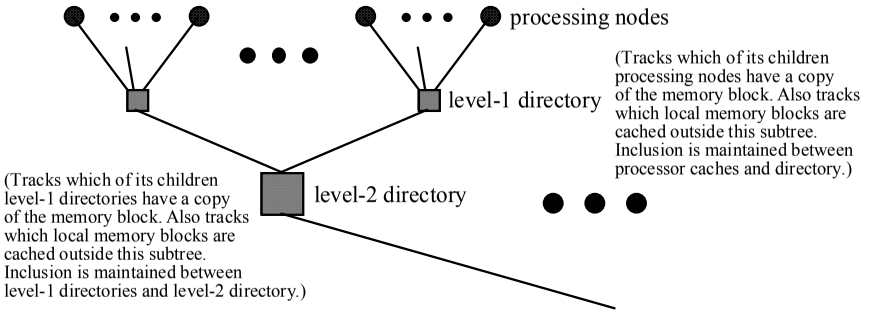
Organizing Directories



How to Find Directory Information

- centralized memory and directory - easy: go to it
  - but not scalable
- distributed memory and directory
  - flat schemes
    - » directory distributed with memory: at the home
    - » location based on address (hashing): network xaction sent directly to home
  - hierarchical schemes
    - » ??

How Hierarchical Directories Work



- Directory is a hierarchical data structure
  - leaves are processing nodes, internal nodes just directory
  - logical hierarchy, not necessarily phyiscal
    - » (can be embedded in general network)

Find Directory Info (cont)

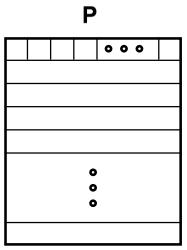
- distributed memory and directory
  - flat schemes
    - » hash
  - hierarchical schemes
    - » node's directory entry for a block says whether each subtree caches the block
    - » to find directory info, send "search" message up to parent
      - routes itself through directory lookups
    - » like hiearchical snooping, but point-to-point messages between children and parents

How Is Location of Copies Stored?

- Hierarchical Schemes
  - through the hierarchy
  - each directory has presence bits child subtrees and dirty bit
- Flat Schemes
  - vary a lot
  - different storage overheads and performance characteristics
  - Memory-based schemes
    - » info about copies stored all at the home with the memory block
    - » Dash, Alewife , SGI Origin, Flash
  - Cache-based schemes
    - » info about copies distributed among copies themselves
      - each copy points to next
    - » Scalable Coherent Interface (SCI: IEEE standard)

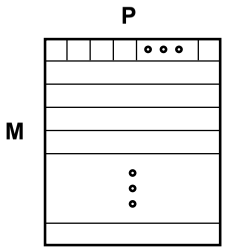
Flat, Memory-based Schemes

- info about copies colocated with block at the home
  - just like centralized scheme, except distributed
- Performance Scaling
  - traffic on a write: proportional to number of sharers
  - latency on write: can issue invalidations to sharers in parallel
- Storage overhead
  - simplest representation: *full bit vector*, i.e. one presence bit per node
  - storage overhead doesn't scale well with P; 64-byte line implies
    - » 64 nodes: 12.7% ovhd.
    - » 256 nodes: 50% ovhd.; 1024 nodes: 200% ovhd.
  - for M memory blocks in memory, storage overhead is proportional to P*M



Reducing Storage Overhead

- Optimizations for full bit vector schemes
  - increase cache block size (reduces storage overhead proportionally)
  - use multiprocessor nodes (bit per mp node, not per processor)
  - still scales as P*M, but reasonable for all but very large machines
    - » 256-procs, 4 per cluster, 128B line: 6.25% ovhd.
- Reducing "width"
  - addressing the P term?
- Reducing "height"
  - addressing the M term?

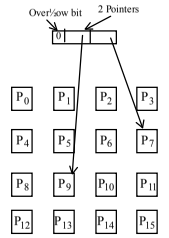


Storage Reductions

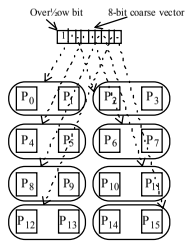
- Width observation:
  - most blocks cached by only few nodes
  - don't have a bit per node, but entry contains a few pointers to sharing nodes
  - P=1024 => 10 bit ptrs, can use 100 pointers and still save space
  - sharing patterns indicate a few pointers should suffice (five or so)
  - need an overflow strategy when there are more sharers
- Height observation:
  - number of memory blocks >> number of cache blocks
  - most directory entries are useless at any given time
  - organize directory as a cache, rather than having one entry per memory block

Overflow Schemes for Limited Pointers

- Broadcast (Dir$_i$B)
  - broadcast bit turned on upon overflow
  - bad for widely-shared frequently read data
- No-broadcast (Dir$_i$NB)
  - on overflow, new sharer replaces one of the old ones (invalidated)
  - bad for widely read data
- Coarse vector (Dir$_i$CV)
  - change representation to a coarse vector, 1 bit per k nodes
  - on a write, invalidate all nodes that a bit corresponds to



(a) No overflow

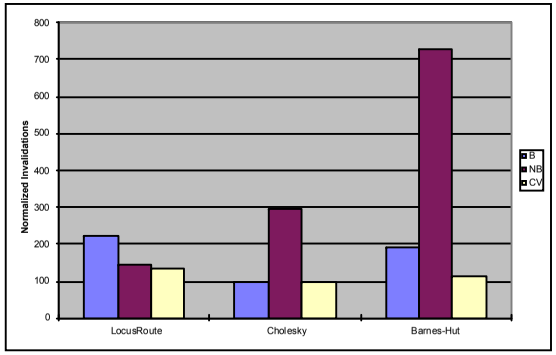


(a) Overflow

Overflow Schemes (contd.)

- Software (Dir_iSW)
  - trap to software, use any number of pointers (no precision loss)
    - » MIT Alewife: 5 ptrs, plus one bit for local node
  - but extra cost of interrupt processing on software
    - » processor overhead and occupancy
    - » latency
      - 40 to 425 cycles for remote read in Alewife
        - Actually, read insertion pipelined, so usually get fast response
      - 84 cycles for 5 inval, 707 for 6.
- Dynamic pointers (Dir_iDP)
  - use pointers from a hardware free list in portion of memory
  - manipulation done by hw assist, not sw
  - e.g. Stanford FLASH

Some Data



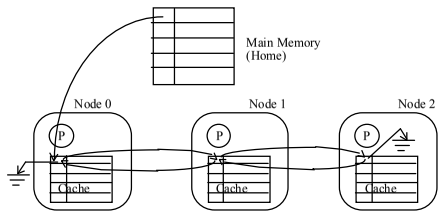
- 64 procs, 4 pointers, normalized to full-bit-vector
- Coarse vector quite robust
- General conclusions:
  - full bit vector simple and good for moderate-scale
  - several schemes should be fine for large-scale

Reducing Height: Sparse Directories

- Reduce M term in P*M
- Observation: total number of cache entries << total amount of memory.
  - most directory entries are idle most of the time
  - 1MB cache and 64MB per node => 98.5% of entries are idle
- Organize directory as a cache
  - but no need for backup store
    - » send invalidations to all sharers when entry replaced
  - one entry per "line"; no spatial locality
  - different access patterns (from many procs, but filtered)
  - allows use of SRAM, can be in critical path
  - needs high associativity, and should be large enough
- Can trade off width and height

Flat, Cache-based Schemes

- How they work:
  - home only holds pointer to rest of directory info
  - distributed linked list of copies, weaves through caches
    - » cache tag has pointer, points to next cache with a copy
  - on read, add yourself to head of the list (comm. needed)
  - on write, propagate chain of invals down the list
- Scalable Coherent Interface (SCI) IEEE Standard
  - doubly linked list



Scaling Properties (Cache-based)

- Traffic on write: proportional to number of sharers
- Latency on write: proportional to number of sharers!
  - don't know identity of next sharer until reach current one
  - also assist processing at each node along the way
  - (even reads involve more than one other assist: home and first sharer on list)
- Storage overhead: quite good scaling along both axes
  - Only one head ptr per memory block
    - » rest is all prop to cache size
- Very complex!!!
- Great example of why standards should not happen before research!!!!

Summary of Directory Organizations

- Flat Schemes:
- Issue (a): finding source of directory data
  - go to home, based on address
- Issue (b): finding out where the copies are
  - memory-based: all info is in directory at home
  - cache-based: home has pointer to first element of distributed linked list
- Issue (c): communicating with those copies
  - memory-based: point-to-point messages (perhaps coarser on overflow)
    - » can be multicast or overlapped
  - cache-based: part of point-to-point linked list traversal to find them
    - » serialized
- Hierarchical Schemes:
  - all three issues through sending messages up and down tree
  - no single explict list of sharers
  - only direct communication is between parents and children

Summary of Directory Approaches

- Directories offer scalable coherence on general networks
  - no need for broadcast media
- Many possibilities for organizing directory and managing protocols
- Hierarchical directories not used much
  - high latency, many network transactions, and bandwidth bottleneck at root
- Both memory-based and cache-based flat schemes are alive
  - for memory-based, full bit vector suffices for moderate scale
    - » measured in nodes visible to directory protocol, not processors
  - will examine case studies of each

Issues for Directory Protocols

- Correctness
- Performance
- Complexity and dealing with errors

Discuss major correctness and performance issues that a protocol must address

Then delve into memory- and cache-based protocols, tradeoffs in how they might address (case studies)

Complexity will become apparent through this

Correctness

- Ensure basics of coherence at state transition level
  - relevant lines are updated/invalidated/fetched
  - correct state transitions and actions happen
- Ensure ordering and serialization constraints are met
  - for coherence (single location)
  - for consistency (multiple locations): assume sequential consistency
- Avoid deadlock, livelock, starvation
- Problems:
  - multiple copies AND multiple paths through network (distributed pathways)
  - unlike bus and non cache-coherent (each had only one)
  - large latency makes optimizations attractive
    - » increase concurrency, complicate correctness

Coherence: Serialization to a Location

- Need entity that sees op's from many procs
- bus:
  - multiple copies, but serialization by bus imposed order
- scalable MP without coherence:
  - main memory module determined order
- scalable MP with cache coherence
  - home memory good candidate
    - » all relevant ops go home first
  - but multiple copies
    - » valid copy of data may not be in main memory
    - » reaching main memory in one order does not mean will reach valid copy in that order
    - » serialized in one place doesn't mean serialized wrt all copies

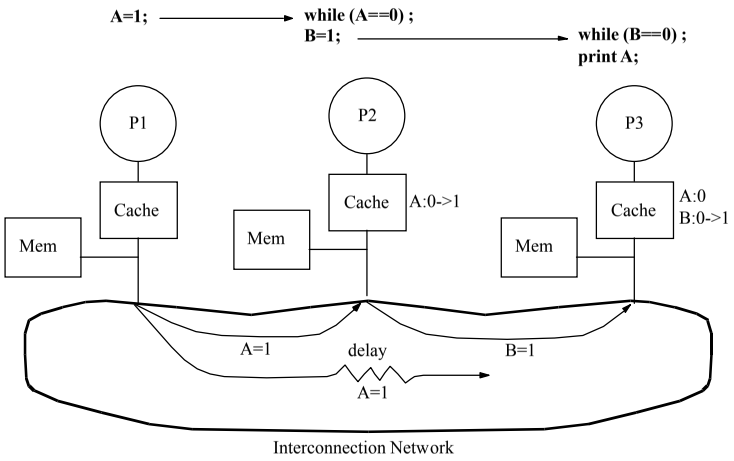
Basic Serialization Solution

- Use additional 'busy' or 'pending' directory states
- Indicate that operation is in progress, further operations on location must be delayed
  - buffer at home
  - buffer at requestor
  - NACK and retry
  - forward to dirty node

Sequential Consistency

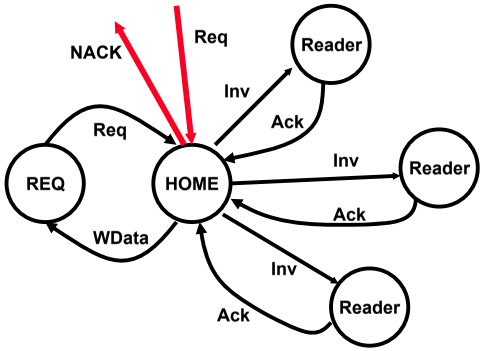
- bus-based:
  - write completion: wait till gets on bus
  - write atomicity: bus plus buffer ordering provides
- non-coherent scalable case
  - write completion: needed to wait for explicit ack from memory
  - write atomicity: easy due to single copy
- now, with multiple copies and distributed network pathways
  - write completion: need explicit acks from copies themselves
  - writes are not easily atomic
  - ... in addition to earlier issues with bus-based and non-coherent

Write Atomicity Problem



Basic Solution

- In invalidation-based scheme, block owner (mem to $) provides appearance of atomicity by waiting for all invalidations to be ack'd before allowing access to new value.
- much harder in update schemes!



Livelock???

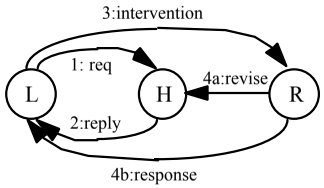
- What happens if popular item is written frequently?
  - Possible that some disadvantaged node never makes progress!
- Solutions?
  - Ignore
  - Queuing at directory: Possible scalability problems
  - Escalating priorities of requests (SGI Origin)
    - » Pending queue of length 1
    - » Keep item of highest priority in that queue
    - » New requests start at priority 0
    - » When NACK happens, increase priority

Performance

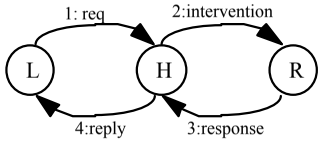
- Latency
  - protocol optimizations to reduce network xactions in critical path
  - overlap activities or make them faster
- Throughput
  - reduce number of protocol operations per invocation
- Care about how these scale with the number of nodes

Protocol Enhancements for Latency

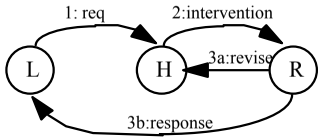
- Forwarding messages: memory-based protocols



(a) Strict request-reply



(a) Intervention forwarding



(a) Reply forwarding

Intervention is like a req, but issued in reaction to req. and sent to cache, rather than memory.

Other Latency Optimizations

- Throw hardware at critical path
  - SRAM for directory (sparse or cache)
  - bit per block in SRAM to tell if protocol should be invoked
- Overlap activities in critical path
  - multiple invalidations at a time in memory-based
  - overlap invalidations and acks in cache-based
  - lookups of directory and memory, or lookup with transaction
    - » speculative protocol operations

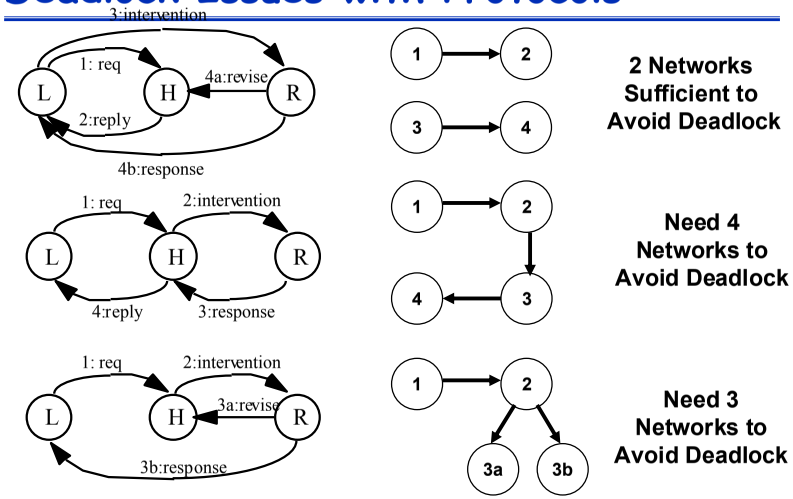
Increasing Throughput

- Reduce the number of transactions per operation
  - invals, acks, replacement hints
  - all incur bandwidth and assist occupancy
- Reduce assist occupancy or overhead of protocol processing
  - transactions small and frequent, so occupancy very important
- Pipeline the assist (protocol processing)
- Many ways to reduce latency also increase throughput
  - e.g. forwarding to dirty node, throwing hardware at critical path...

Deadlock, Livelock, Starvation

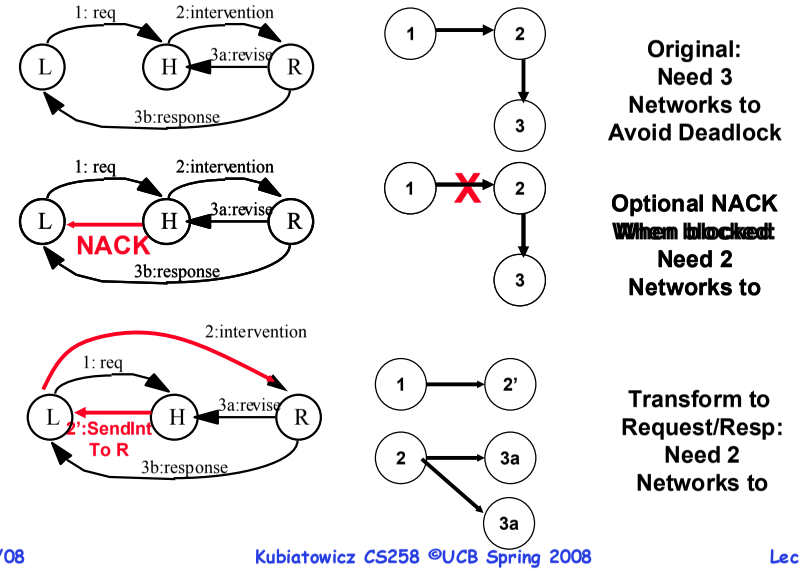
- Request-response protocol
- Similar issues to those discussed earlier
  - a node may receive too many messages
  - flow control can cause deadlock
  - separate request and reply networks with request-reply protocol
  - Or NACKs, but potential livelock and traffic problems
- New problem: protocols often are not strict request-reply
  - e.g. rd-excl generates inval requests (which generate ack replies)
  - other cases to reduce latency and allow concurrency

Deadlock Issues with Protocols



- Consider Dual graph of message dependencies
  - Number of networks = length of longest dependency
  - Must always make sure response (end) can be absorbed!

Mechanisms for reducing depth



Complexity?

- Cache coherence protocols are complex
- Choice of approach
  - conceptual and protocol design versus implementation
- Tradeoffs within an approach
  - performance enhancements often add complexity, complicate correctness
    - » more concurrency, potential race conditions
    - » not strict request-reply
- Many subtle corner cases
  - BUT, increasing understanding/adoption makes job much easier
  - automatic verification is important but hard
- Next time: Let's look at memory- and cache-based more deeply through case studies

Summary

- Types of Cache Coherence Schemes
  - UMA – Uniform Memory Access
  - NUMA – Non-uniform Memory Access
  - COMA – "Cache-Only" Memory Architecture
- Distributed Directory Structure
  - Flat: Each address has a "home node"
  - Hierarchical: directory spread along tree
- Mechanism for locating copies of data
  - Memory-based schemes
    - » info about copies stored all at the home with the memory block
    - » Dash, Alewife , SGI Origin, Flash
  - Cache-based schemes
    - » info about copies distributed among copies themselves
      - each copy points to next
    - » Scalable Coherent Interface (SCI: IEEE standard)