Cache Coherent NUMA (CC-NUMA)

Scalable Cache Coherent Systems

- Scalable, distributed memory plus coherent replication
- Shared physical address space
  - cache miss satisfied transparently from local or remote memory
- Natural tendency of cache is to replicate
  - but coherence?
  - no broadcast medium to snoop on
- Not only hardware latency/bw, but also protocol must scale

What Must a Coherent System Do?

- Provide set of states, state transition diagram, and actions
- Manage coherence protocol
  - (0) Determine when to invoke coherence protocol
  - (a) Find source of info about state of line in other caches
    - whether need to communicate with other cached copies
  - (b) Find out where the other copies are
  - (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

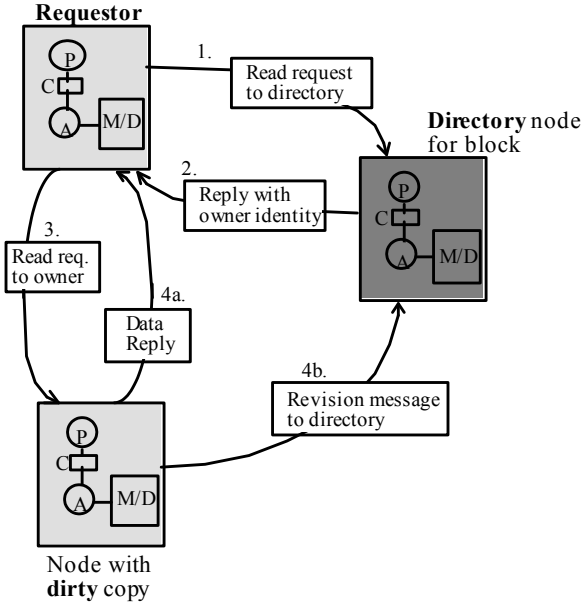
- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a “search”
  - others respond to the search probe and take necessary action
- Could do it in scalable network too
  - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn’t scale 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

Approach #1: Hierarchical Snooping

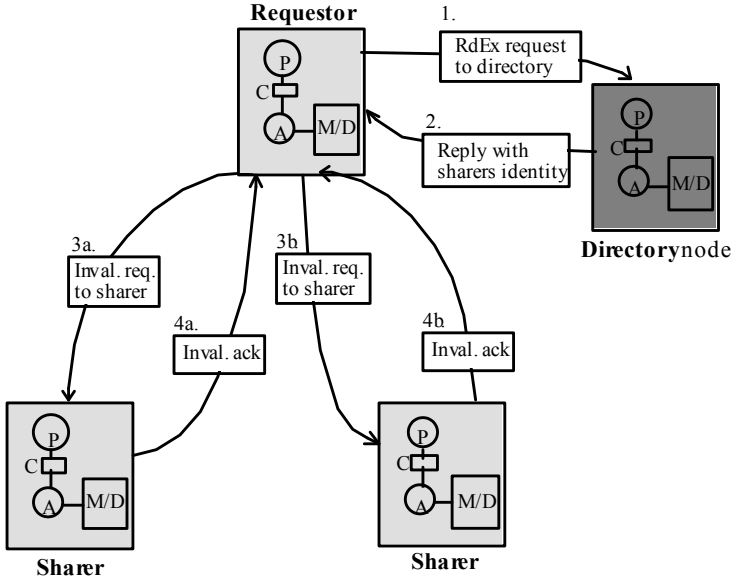
- Extend snooping approach: hierarchy of broadcast media
  - tree of buses or rings (KSR-1)
  - processors are in the bus- or ring-based multiprocessors at the leaves
  - parents and children connected by two-way snoopy interfaces
    - snoop both buses and propagate relevant transactions
  - main memory 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 hiearchy based on snoop results
- Problems:
  - high latency: multiple levels, and snoop/lookup at every level
  - bandwidth bottleneck at root
- Not popular today

Scalable Approach #2: 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, comm. with directory and copies is through *network transactions*



(a) Read miss to a block in dirty state



(b) Write miss to a block with tw o sharers

- Many alternatives for organizing directory information

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

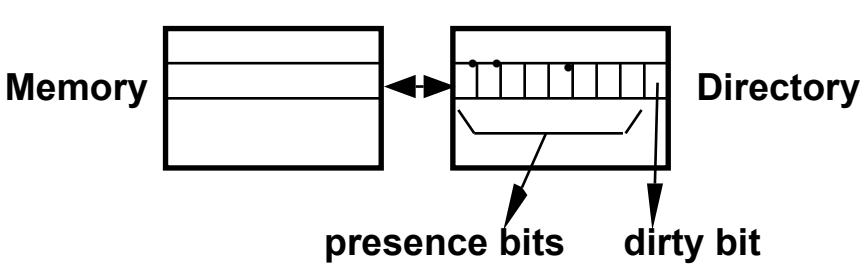
Advantages of Multiprocessor Nodes

- Potential for cost and performance advantages
  - amortization of node fixed costs over multiple processors
    - applies even if processors simply packaged together but not coherent
  - can use commodity SMPs
  - less nodes for directory to keep track of
  - much communication may be contained within node (cheaper)
  - nodes prefetch data for each other (fewer “remote” misses)
  - combining of requests (like hierarchical, only two-level)
  - can even share caches (overlapping of working sets)
  - benefits depend on sharing pattern (and mapping)
    - good for widely read-shared: e.g. tree data in Barnes-Hut
    - good for nearest-neighbor, if properly mapped
    - not so good for all-to-all communication

Disadvantages of Coherent MP Nodes

- Bandwidth shared among nodes
  - all-to-all example
  - applies to coherent or not
- Bus increases latency to local memory
- With coherence, typically wait for local snoop results before sending remote requests
- Snoopy bus at remote node increases delays there too, increasing latency and reducing bandwidth
- Overall, may hurt performance if sharing patterns don’t comply

Basic Operation of Directory



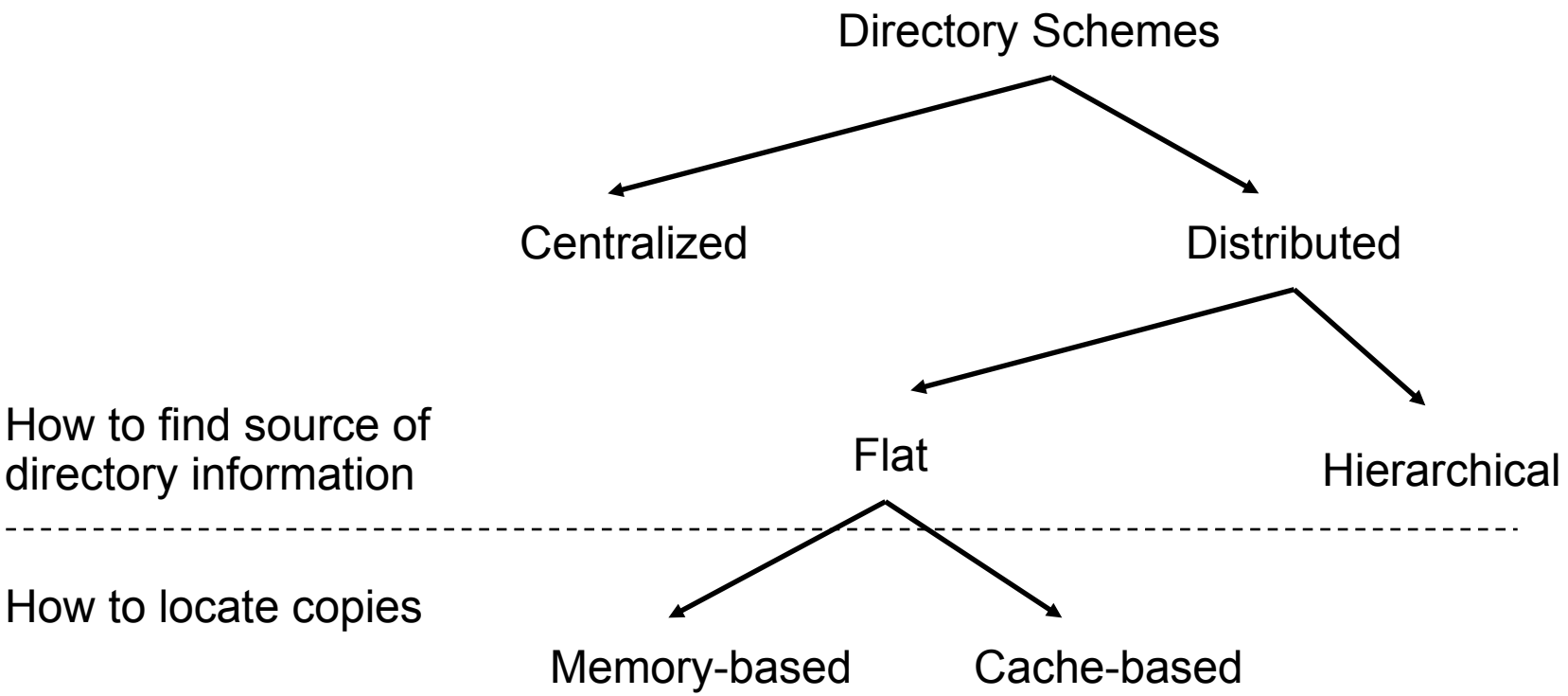
- k processors.
- With each cache-block in memory: k presence-bits, 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit

- 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; ... }
  - ...

Scaling with No. of Processors

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

Organizing Directories



Let’s see how they work and their scaling characteristics with P

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
    - directory organized as a hierarchical data structure
    - leaves are processing nodes, internal nodes have only directory state
    - 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 for its children (subtrees), and dirty bit

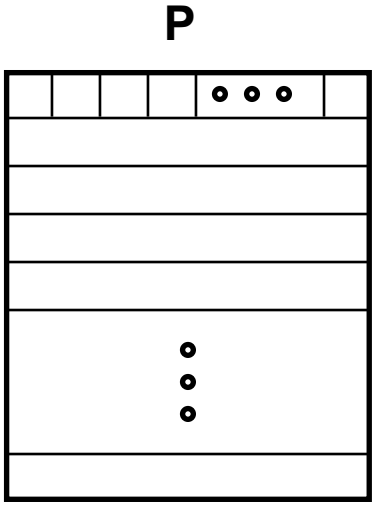
- Flat Schemes
  - varies a lot
  - different storage overheads and performance characteristics

  - 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

- All info about copies colocated with block itself at the home
  - work just like centralized scheme, except distributed

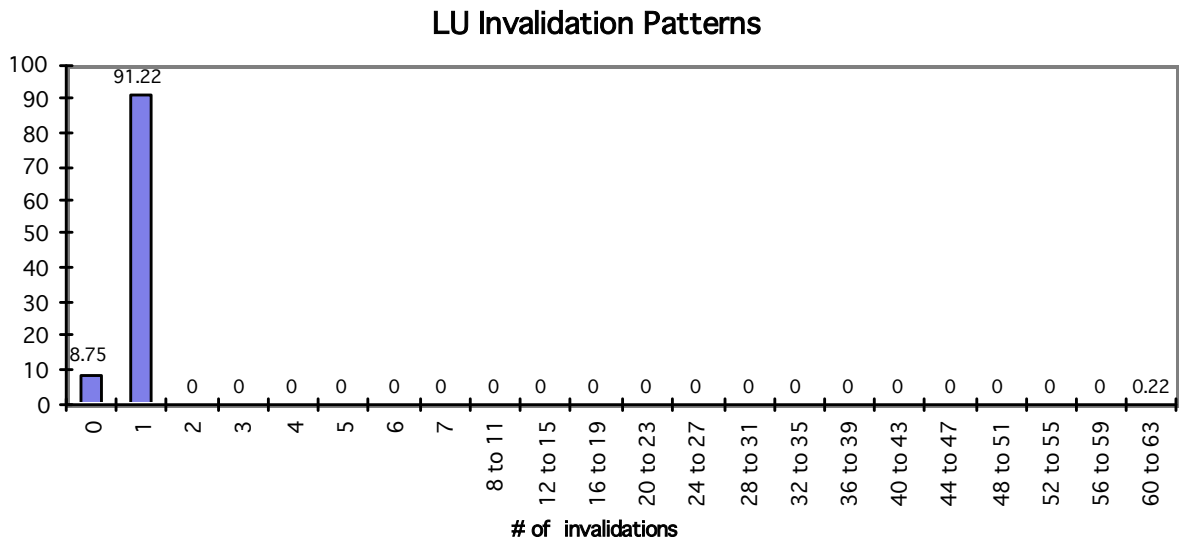


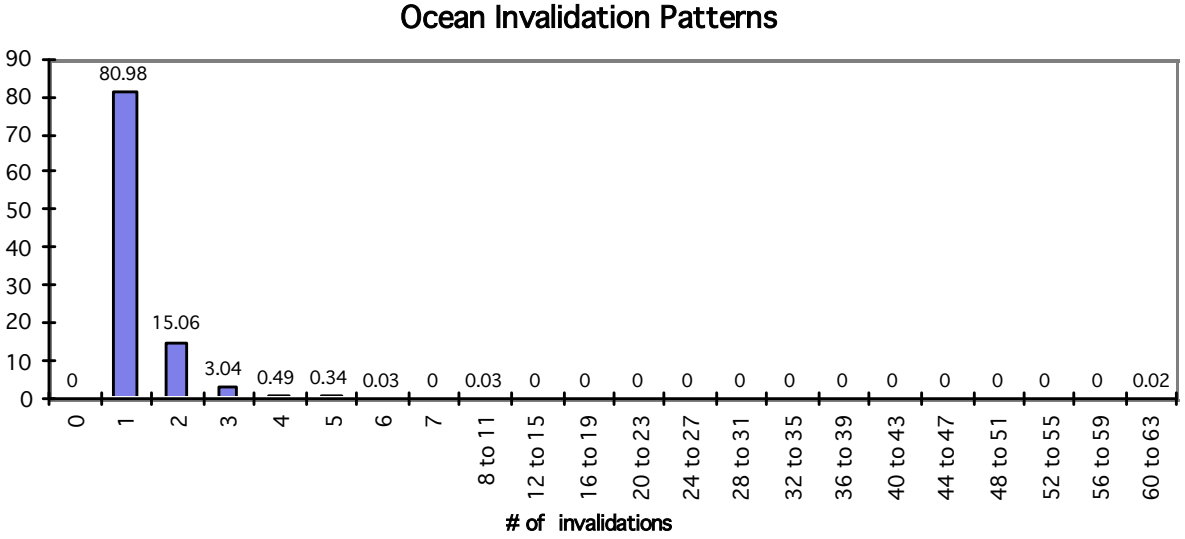
- Scaling of performance characteristics
  - traffic on a write: proportional to number of sharers
  - latency a write: can issue invalidations to sharers in parallel
- Scaling of storage overhead
  - simplest representation: *full bit vector*, i.e. one presence bit per node
  - storage overhead doesn't 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

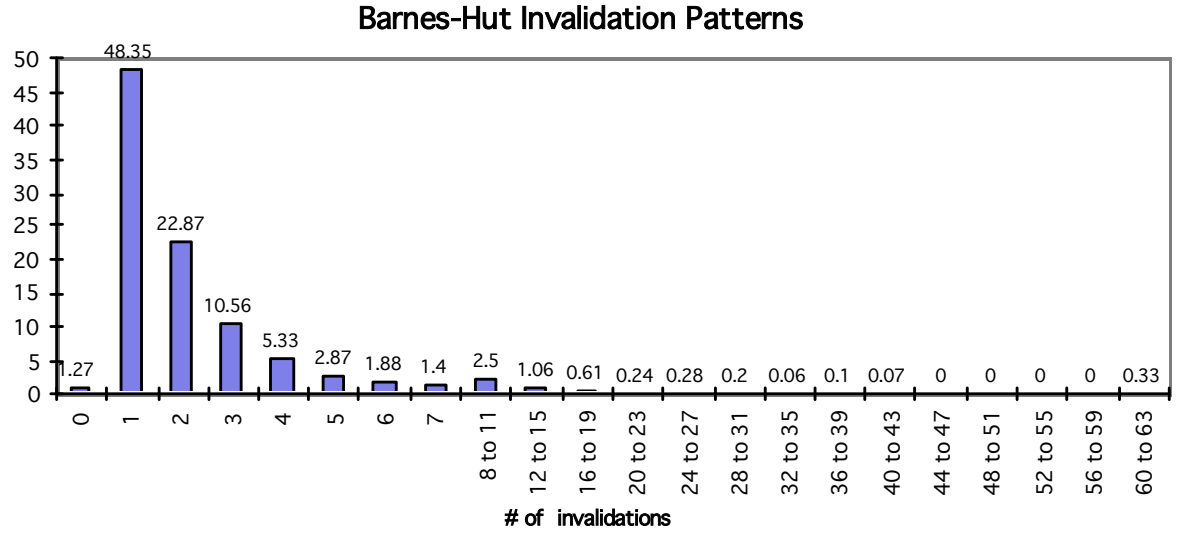
- Optimizations for full bit vector schemes
  - increase cache block size (reduces storage overhead proportionally)
  - use multiprocessor nodes (bit per multiprocessor node, not per processor)
  - still scales as P*M, but not a problem for all but very large machines
    - 256-procs, 4 per cluster,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

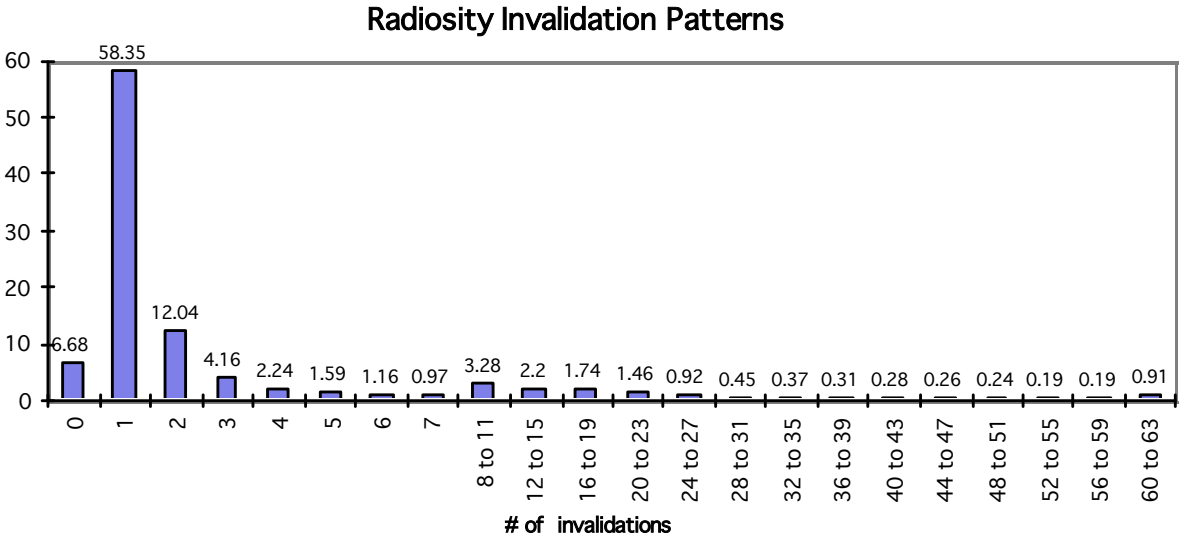
Cache Invalidation Patterns





Cache Invalidation Patterns



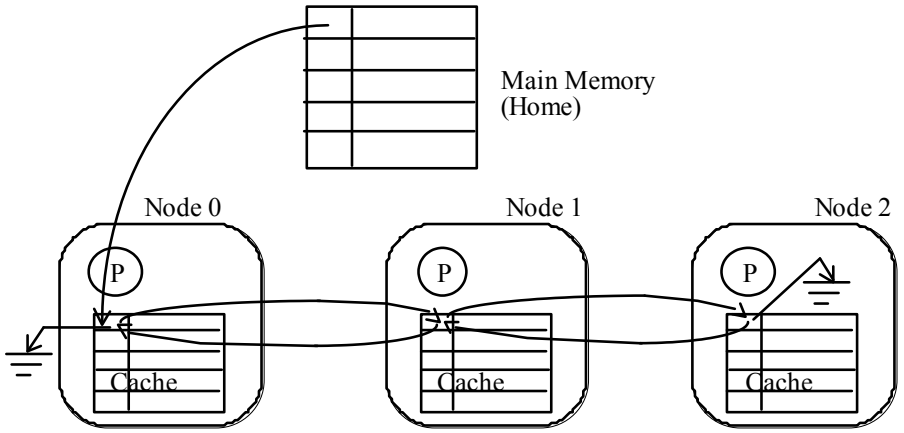


Sharing Patterns Summary

- Generally, only a 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

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

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

Issues for Directory Protocols

- Correctness
- Performance
- Complexity and dealing with errors

Correctness

- Ensure basics of coherence at state transition level
  - 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 still
- Avoid deadlock, livelock, starvation

Coherence: Serialization to A Location

- on a bus, multiple copies but serialization by bus imposed order
- If you did not have copies/caches, main memory module determined order
- could use main memory module here too, 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 (later)

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
- Must address livelock and starvation too

Will see how protocols address these correctness issues

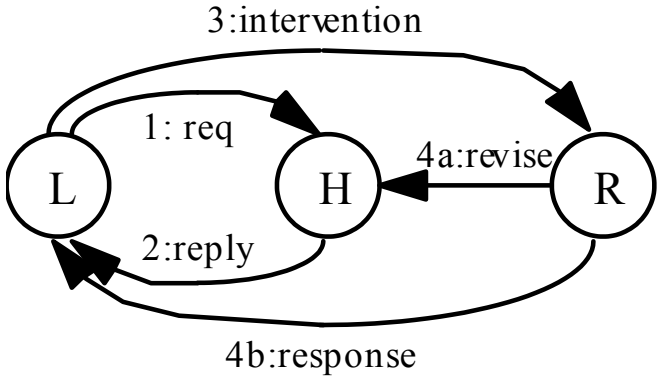
Performanc e

- Latency
  - protocol optimizations to reduce network xactions in critical path
  - overlap activities or make them faster
- Throughput
  - reduce number of protocol operations per invocation

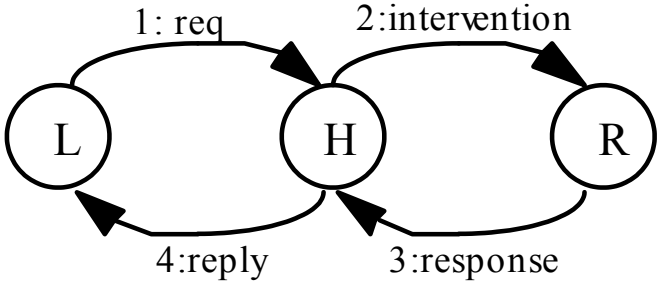
Care about how these scale with the number of nodes

Protocol Enhancements for Latency

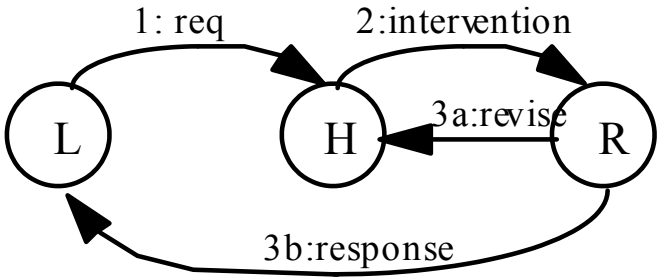
- Forwarding messages: memory-based protocols



(a) Strict request-reply



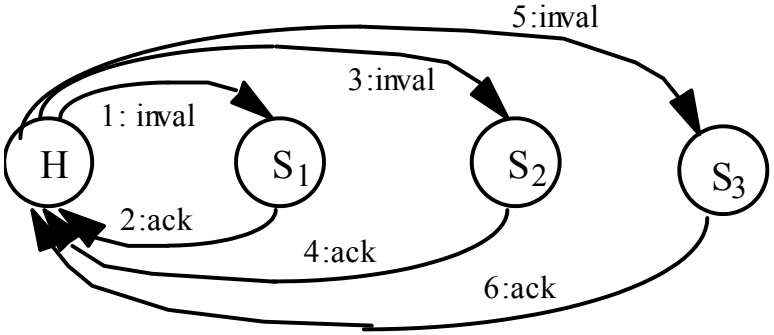
(a) Intervention forwarding



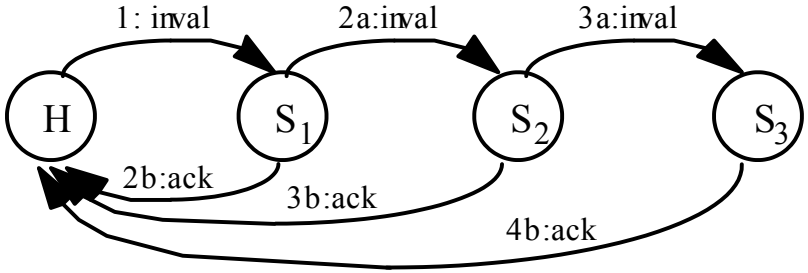
(a) Reply forwarding

Protocol Enhancements for Latency

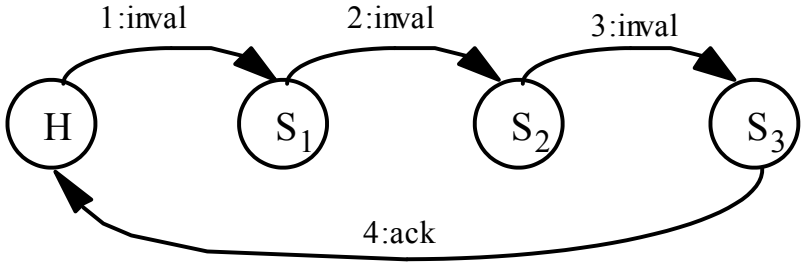
- Forwarding messages: cache-based protocols



(a)



(b)



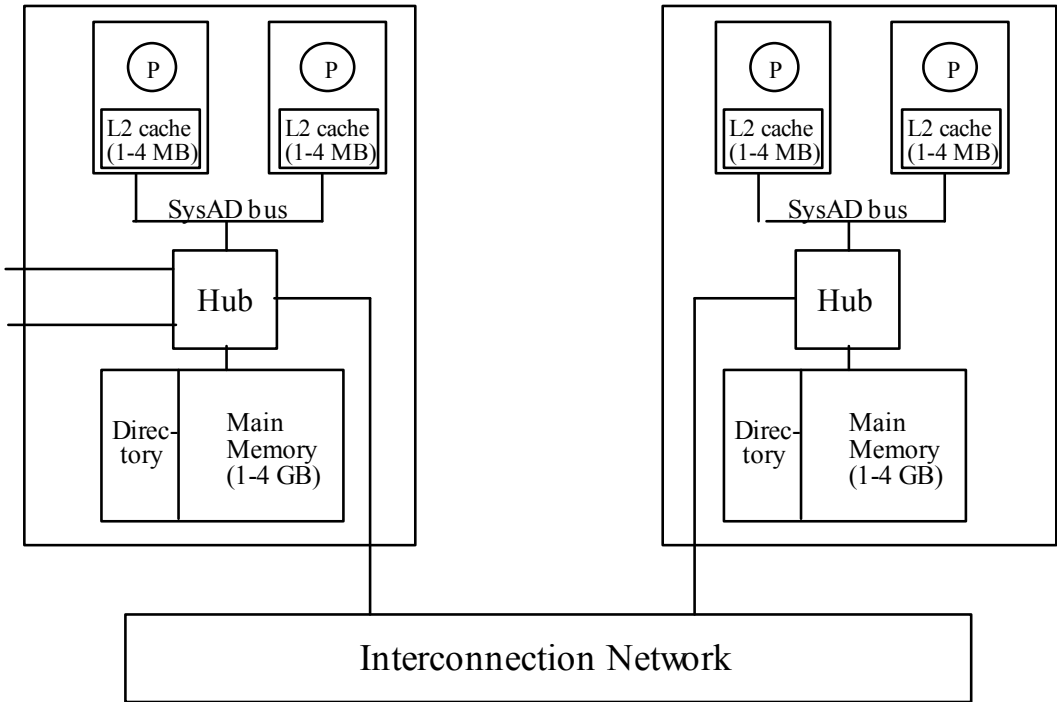
(c)

Flat, Memory-based Protocols

- Use SGI Origin2000 Case Study
  - Protocol similar to Stanford DASH, but with some different tradeoffs
  - Also Alewife, FLASH, HAL

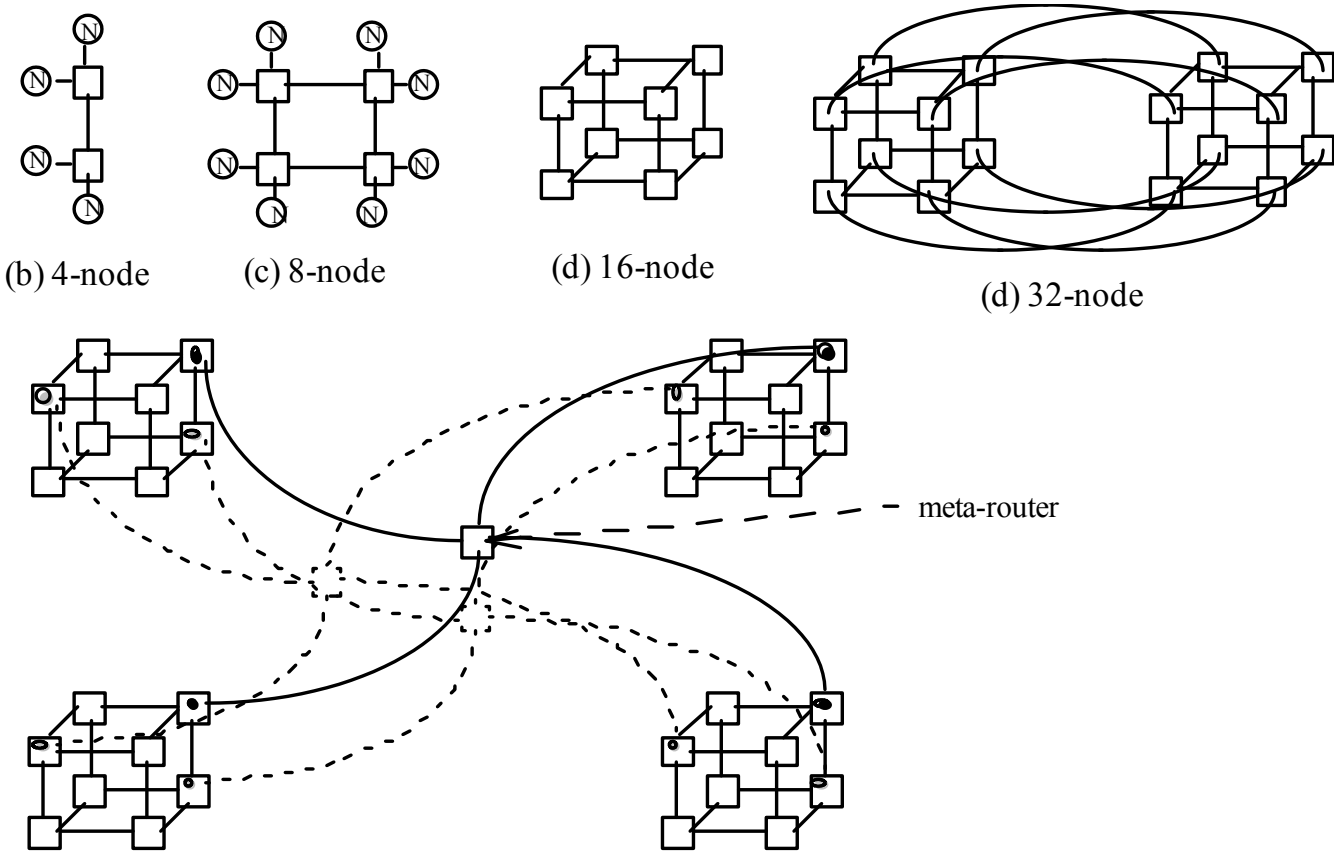
- Outline:
  - System Overview
  - Coherence States, Representation and Protocol
  - Correctness and Performance Tradeoffs
  - Implementation Issues
  - Quantiative Performance Characteristics

Origin2000 System Overview



- Single 16”-by-11” PCB
- Directory state in same or separate DRAMs, accessed in parallel
- Upto 512 nodes (1024 processors)
- With 195MHz R10K processor, peak 390MFLOPS or 780 MIPS per proc
- Peak SysAD bus bw is 780MB/s, so also Hub-Mem
- Hub to router chip and to Xbow is 1.56 GB/s (both are of-board)

Origin Network



- Each router has six pairs of 1.56MB/s unidirectional links
  - Two to nodes, four to other routers
  - latency: 41ns pin to pin across a router
- Flexible cables up to 3 ft long
- Four “virtual channels”: request, reply, other two for priority or I/O

Origin Directory Structure

- Flat, Memory based: all directory information at the home
- Three directory formats:
  - (1) if exclusive in a cache, entry is *pointer* to that specific processor (not node)
  - (2) if shared, *bit vector*: each bit points to a node (Hub), not processor
  - invalidation sent to a Hub is broadcast to both processors in the node
  - two sizes, depending on scale
    - 16-bit format (32 procs), kept in main memory DRAM
    - 64-bit format (128 procs), extra bits kept in extension memory
  - (3) for larger machines, *coarse vector*: each bit corresponds to p/64 nodes
  - invalidation is sent to all Hubs in that group, which each bcast to their 2 procs
  - machine can choose between bit vector and coarse vector dynamically
    - is application confined to a 64-node or less part of machine?
- Ignore coarse vector in discussion for simplicity

Origin Cache and Directory States

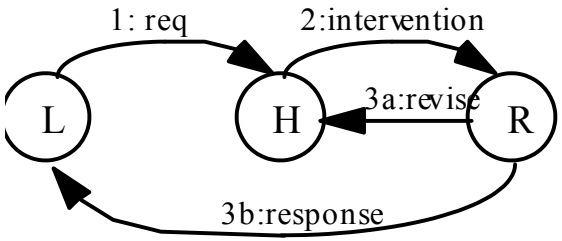
- Cache states: MESI
- Seven directory states
  - *unowned*: no cache has a copy, memory copy is valid
  - *shared*: one or more caches has a shared copy, memory is valid
  - *exclusive*: one cache (pointed to) has block in modified or exclusive state
  - three *pending* or *busy* states, one for each of the above:
    - indicates directory has received a previous request for the block
    - couldn't satisfy it itself, sent it to another node and is waiting
    - cannot take another request for the block yet
  - *poisoned* state, used for efficient page migration (later)
- Let's see how it handles read and "write" requests
  - no point-to-point order assumed in network

Handling a Read Miss

- Hub looks at address
  - if remote, sends request to home
  - if local, looks up directory entry and memory itself
  - directory may indicate one of many states
- *Shared or Unowned* State:
  - if shared, directory sets presence bit
  - if unowned, goes to exclusive state and uses pointer format
  - replies with block to requestor
    - strict request-reply (no network transactions if home is local)
  - actually, also looks up memory speculatively to get data, in parallel with dir
    - directory lookup returns one cycle earlier
    - if directory is shared or unowned, it's a win: data already obtained by Hub
    - if not one of these, speculative memory access is wasted
- Busy state: not ready to handle
  - NACK, so as not to hold up buffer space for long

Read Miss to Block in Exclusive State

- Most interesting case
  - if owner is not home, need to get data to home and requestor from owner
  - Uses reply forwarding for lowest latency and traffic
    - not strict request-reply



- Problems with “intervention forwarding” option
  - replies come to home (which then replies to requestor)
  - a node may have to keep track of P*k outstanding requests as home
    - with reply forwarding only k since replies go to requestor
  - more complex, and lower performance

Actions at Home and Owner

- At the home:
  - set directory to busy state and NACK subsequent requests
    - general philosophy of protocol
    - can't set to shared or exclusive
    - alternative is to buffer at home until done, but input buffer problem
  - set and unset appropriate presence bits
  - assume block is clean-exclusive and send speculative reply
- At the owner:
  - If block is dirty
    - send data reply to requestor, and "sharing writeback" with data to home
  - If block is clean exclusive
    - similar, but don't send data (message to home is called "downgrade"
- Home changes state to shared when it receives revision msg

Influence of Processor on Protocol

- Why speculative replies?
  - requestor needs to wait for reply from owner anyway to know
  - no latency savings
  - could just get data from owner always
- Processor designed to not reply with data if clean-exclusive
  - so needed to get data from home
  - wouldn't have needed speculative replies with intervention forwarding
- Also enables another optimization (later)
  - needn't send data back to home when a clean-exclusive block is replaced

Handling a Write Miss

- Request to home could be upgrade or read-exclusive
- State is busy: NACK
- State is unowned:
  - if RdEx, set bit, change state to dirty, reply with data
  - if Upgrade, means block has been replaced from cache and directory already notified, so upgrade is inappropriate request
    - NACKed (will be retried as RdEx)
- State is shared or exclusive:
  - invalidations must be sent
  - use reply forwarding; i.e. invalidations acks sent to requestor, not home

Write to Block in Shared State

- At the home:
  - set directory state to exclusive and set presence bit for requestor
    - ensures that subsequent requests willbe forwarded to requestor
  - If RdEx, send “excl. reply with invals pending” to requestor (contains data)
    - how many sharers to expect invalidations from
  - If Upgrade, similar “upgrade ack with invals pending” reply, no data
  - Send invals to sharers, which will ack requestor

- At requestor, wait for all acks to come back before “closing” the operation
  - subsequent request for block to home is forwarded as intervention to requestor
  - for proper serialization, requestor does not handle it until all acks received for its outstanding request

Write to Block in Exclusive State

- If upgrade, not valid so NACKed
  - another write has beaten this one to the home, so requestor's data not valid
- If RdEx:
  - like read, set to busy state, set presence bit, send speculative reply
  - send invalidation to owner with identity of requestor
- At owner:
  - if block is dirty in cache
    - send "ownership xfer" revision msg to home (no data)
    - send response with data to requestor (overrides speculative reply)
  - if block in clean exclusive state
    - send "ownership xfer" revision msg to home (no data)
    - send ack to requestor (no data; got that from speculative reply)

Handling Writeback Requests

- Directory state cannot be shared or unowned
  - requestor (owner) has block dirty
  - if another request had come in to set state to shared, would have been forwarded to owner and state would be busy
- State is exclusive
  - directory state set to unowned, and ack returned
- State is busy: interesting race condition
  - busy because intervention due to request from another node (Y) has been forwarded to the node X that is doing the writeback
    - intervention and writeback have crossed each other
  - Y's operation is already in flight and has had it's effect on directory
  - can't drop writeback (only valid copy)
  - can't NACK writeback and retry after Y's ref completes
    - Y's cache will have valid copy while a different dirty copy is written back

Solution to Writeback Race

- Combine the two operations
- When writeback reaches directory, it changes the state
  - to shared if it was busy-shared (i.e. Y requested a read copy)
  - to exclusive if it was busy-exclusive
- Home forwards the writeback data to the requestor Y
  - sends writeback ack to X
- When X receives the intervention, it ignores it
  - knows to do this since it has an outstanding writeback for the line
- Y's operation completes when it gets the reply
- X's writeback completes when it gets the writeback ack

What if WB is coming from different node?

- Block is Dirty in P1
- P2 sends Rd_Ex to Home
- Home forwards to P1 (and goes to Busy state)
- P1 sends (a) Data to P2 , (b) Transfer Ownership msg to Home
- Before 3(b) reaches Home, P2 replaces Block and sends WB to Home

NOTE: This is different from previous situation since WB is from P2 and NOT P1. In this case, Home NACKs P2.

Replacement of Shared Block

- Could send a replacement hint to the directory
  - to remove the node from the sharing list
- Can eliminate an invalidation the next time block is written
- But does not reduce traffic
  - have to send replacement hint
  - incurs the traffic at a different time
- Origin protocol does not use replacement hints
- Total transaction types:
  - coherent memory: 9 request transaction types, 6 inval/intervention, 39 reply
  - noncoherent (I/O, synch, special ops): 19 request, 14 reply (no inval/intervention)

Preserving Sequential Consistency

- R10000 is dynamically scheduled
  - allows memory operations to issue and execute out of program order
  - but ensures that they become visible and complete in order
  - doesn't satisfy sufficient conditions, but provides SC

- An interesting issue w.r.t. preserving SC
  - On a write to a shared block, requestor gets two types of replies:
    - exclusive reply from the home, indicates write is serialized at memory
    - invalidation acks, indicate that write has completed wrt processors
  - But microprocessor expects only one reply (as in a uniprocessor system)
    - so replies have to be dealt with by requestor's HUB (processor interface)
  - To ensure SC, Hub must wait till inval acks are received before replying to proc
    - can't reply as soon as exclusive reply is received
      - would allow later accesses from proc to complete (writes become visible) before this write

Dealing with Correctness Issues

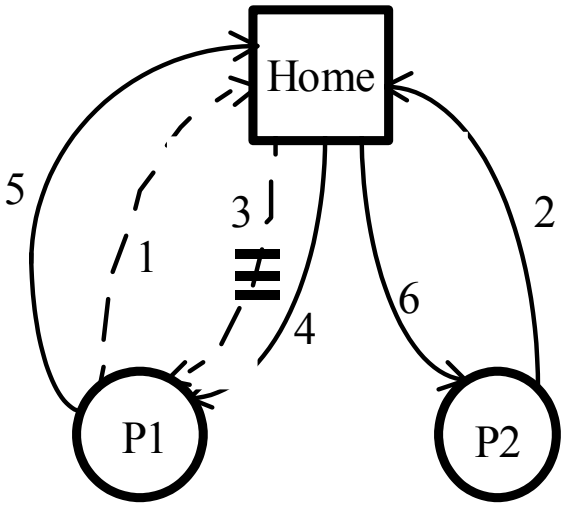
- Serialization of operations
- Deadlock
- Livelock
- Starvation

Serialization of Operations

- Need a serializing agent
  - home memory is a good candidate, since all misses go there first
- Possible Mechanism: FIFO buffering requests at the home
  - until previous requests forwarded from home have returned replies to it
  - but input buffer problem becomes acute at the home
- Possible Solutions:
  - let input buffer overflow into main memory (MIT Alewife)
  - don't buffer at home, but forward to the owner node (Stanford DASH)
    - serialization determined by home when clean, by owner when exclusive
    - if cannot be satisfied at "owner", e.g. written back or ownership given up, NACKed bak to requestor without being serialized
      - serialized when retried
  - don't buffer at home, use busy state to NACK (Origin)
    - serialization order is that in which requests are accepted (not NACKed)
  - maintain the FIFO buffer in a distributed way (SCI, later)

Serialization to a Location (contd)

- Having single entity determine order is not enough
  - it may not know when all xactions for that operation are done everywhere



1. P1 issues read request to home node for A
2. P2 issues read-exclusive request to home corresponding to write of A. But won't process it until it is done with read
3. Home recieves 1, and in response sends reply to P1 (and sets directory presence bit). Home now thinks read is complete. Unfortunately , the reply does not get to P1 right away .
4. In response to 2, home sends invalidate to P1; it reaches P1 before transaction 3 (no point-to-point order among requests and replies).
5. P1 receives and applies invalideate, sends ack to home.
6. Home sends data reply to P2 corresponding to request 2.

Finally , transaction 3 (read reply) reaches P1.

- Home deals with write access before prev. is fully done
- P1 should not allow new access to line until old one "done"

Deadlock

- Two networks not enough when protocol not request-reply
  - Additional networks expensive and underutilized
- Use two, but detect potential deadlock and circumvent
  - e.g. when input request and output request buffers fill more than a threshold, and request at head of input queue is one that generates more requests
  - or when output request buffer is full and has had no relief for T cycles
- Two major techniques:
  - take requests out of queue and NACK them, until the one at head will not generate further requests or ouput request queue has eased up (DASH)
  - fall back to strict request-reply (Origin)
    - instead of NACK, send a reply saying to request directly from owner
    - better because NACKs can lead to many retries, and even livelock
- Origin philosophy:
  - memory-less: node reacts to incoming events using only local state
  - an operation does not hold shared resources while requesting others

Livelock

- Classical problem of two processors trying to write a block
  - Origin solves with busy states and NACKs
    - first to get there makes progress, others are NACKed
- Problem with NACKs
  - useful for resolving race conditions (as above)
  - Not so good when used to ease contention in deadlock-prone situations
    - can cause livelock
    - e.g. DASH NACKs may cause all requests to be retried immediately, regenerating problem continually
      - DASH implementation avoids by using a large enough input buffer
- No livelock when backing off to strict request-reply

Starvation

- Not a problem with FIFO buffering
  - but has earlier problems
- Distributed FIFO list (see SCI later)
- NACKs can cause starvation
- Possible solutions:
  - do nothing; starvation shouldn't happen often (DASH)
  - random delay between request retries
  - priorities (Origin)

Support for I/O

- Protocol supports “Uncached Read-Shared Requests”.
- The Request gives the DMA a snapshot of the “Coherent” data at that.
- It may not be coherent any longer beyond that time.
- “Write Invalidate” makes the DMA overwrite into memory and invalidates all cached copies (Unowned State)

Page Migration

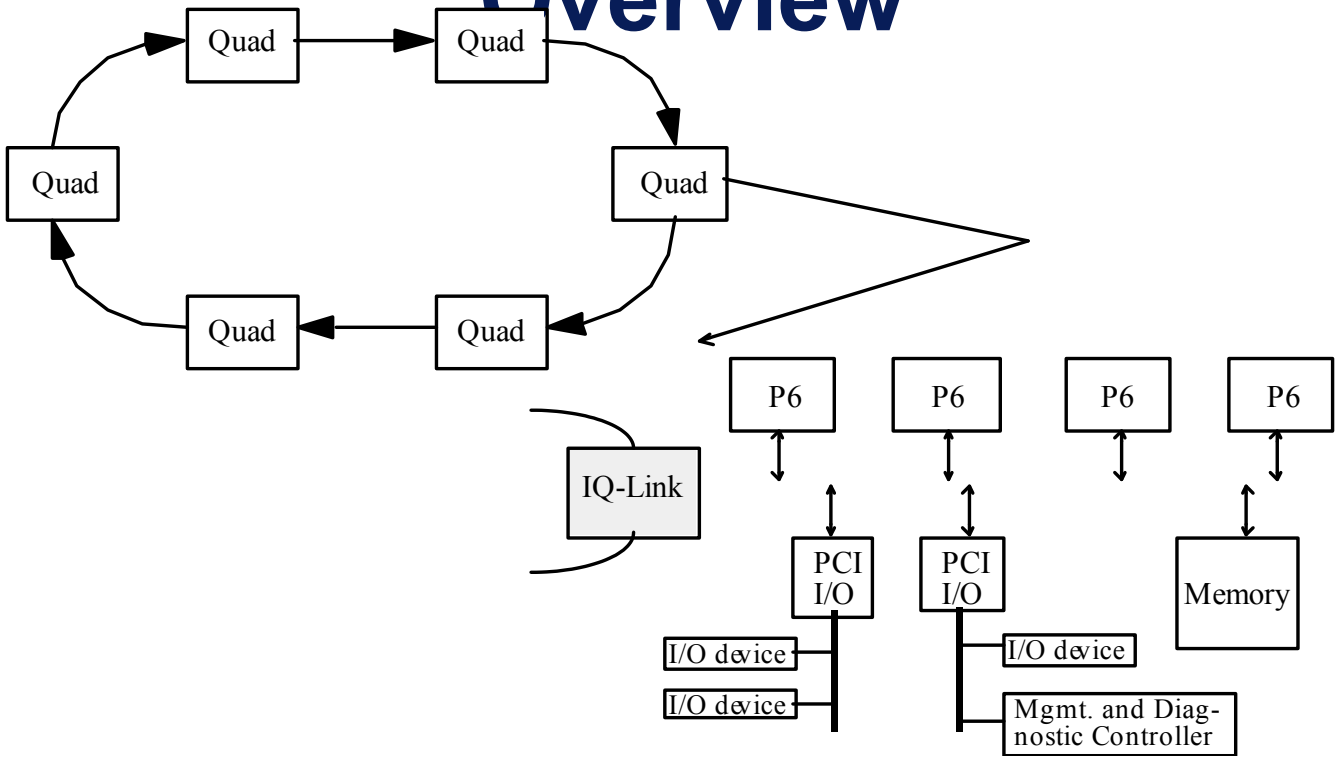
- In addition to caches, we may want to move the page itself to the node which accesses it more.
- When to migrate?
  - At Home, maintain “miss counters” for each page, for each node.
  - If diff > threshold, migrate.
- However, how do you know the page has migrated?
- TLBs have to be updated everywhere (TLB coherence is costly)
  - Origin uses a Lazy scheme
  - Do migration using “Uncached Rd_Ex” requests
  - Set Home state to “Poisoned”
  - Each subsequent request/miss, will see this, incur a Bus Error and update its TLB.

Flat, Cache-based Protocols

- Use Sequent NUMA-Q Case Study
  - Protocol is Scalalble Coherent Interface across nodes, snooping with node
  - Also Convex Exemplar, Data General

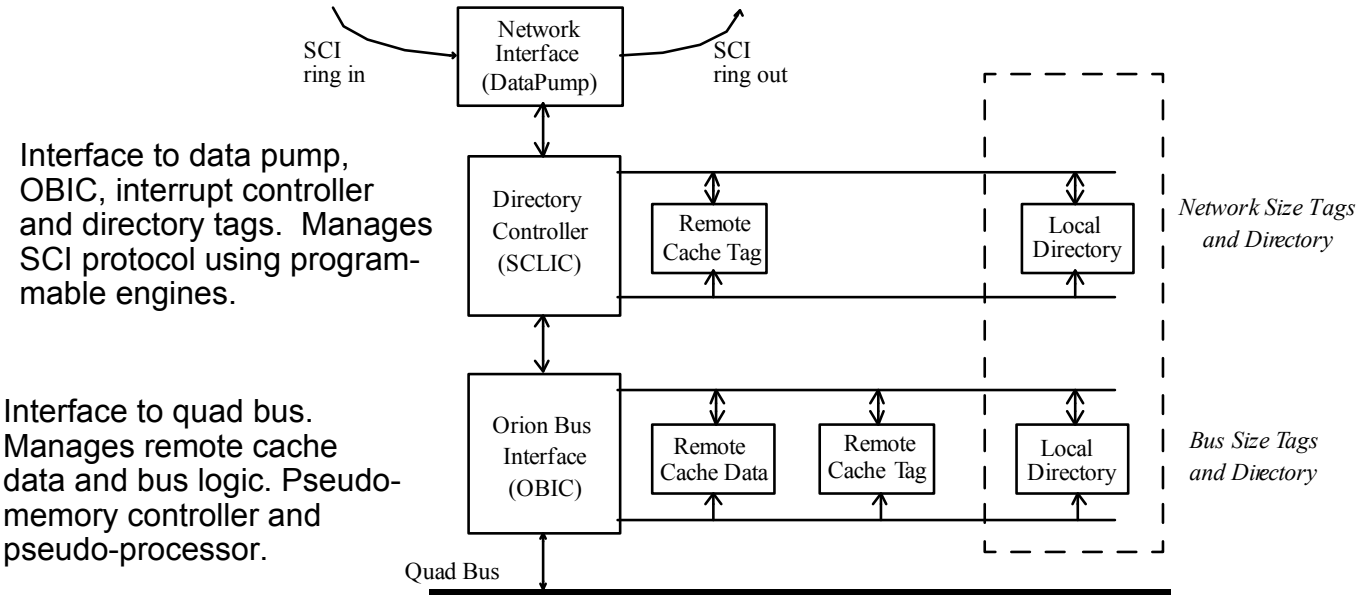
- Outline:
  - System Overview
  - SCI Coherence States, Representation and Protocol
  - Correctness and Performance Tradeoffs
  - Implementation Issues
  - Quantiative Performance Characteristics

NUMA-Q System Overview



- Use of high-volume SMPs as building blocks
- Quad bus is 532MB/s split-transation in-order responses
  - limited facility for out-of-order responses for off-node accesses
- Cross-node interconnect is 1GB/s unidirectional ring
- Larger SCI systems built out of multiple rings connected by bridges
- Within each QUAD, bus-based protocol (MESI) not visible outside

NUMA-Q IQ-Link Board



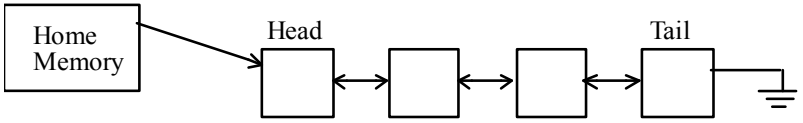
- Plays the role of Hub Chip in SGI Origin
- Can generate interrupts between quads
- Remote cache (visible to SC I) block size is 64 bytes (32MB, 4-way)
  - processor caches not visible (snoopy-coherent and with remote cache)
  - Inclusion Property Obeyed
- Data Pump (GaAs) implements SCI transport, pulls off relevant packets

NUMA-Q Interconnect

- Single ring for initial offering of 8 nodes
  - larger systems are multiple rings connected by LANs
- 18-bit wide SCI ring driven by Data Pump at 1GB/s
- Strict request-reply transport protocol
  - keep copy of packet in outgoing buffer until ack (echo) is returned
  - when take a packet off the ring, replace by “positive echo”
  - if detect a relevant packet but cannot take it in, send “negative echo” (NACK)
  - sender data pump seeing NACK return will retry automatically

SCI Directory Structure

- Flat, Cache-based: sharing list is distributed with caches
  - head, tail and middle nodes, downstream (fwd) and upstream (bkwd) pointers
  - directory entries and pointers stored in S-DRAM in IQ-Link board



- 2-level coherence in NUMA-Q
  - remote cache and SCLIC of 4 procs looks like one node to SCI
  - SCI protocol does not care how many processors and caches are within node

- Directory States = (HOME, FRESH, GONE)
  - HOME = No other Quad has a copy
  - FRESH = Other Quads may have Read-Only copy
  - GONE = Another Quad has a writeable copy

- + Pointer to the Head (Quad) of the sharer list.

Remote Cache (Managed by IQ Link)

- States = (INVALID, ONLY_FRESH, ONLY_DIRTY, HEAD_DIRTY, HEAD_FRESH, MID, TAIL, …. Transitionary states)

- Three Standard Operations:
  - List Construction: Add a node to the head
  - Roll Out: Remove a node from the list
  - Purge: Head invalidates all other nodes

- All network transactions are Request-Response

Read Miss at A

- Read Req sent to HOME after setting local state to Pending/Busy
- At Directory, state is:
  - HOME:
    - Dir changes state to FRESH
    - Sets Ptr to A
    - Sends back Data
    - State at A is set to ONLY_FRESH
  - FRESH:
    - Dir sets ptr to A
    - Send Data and Prev Head to A
    - A changes to another pending state
    - A sends req to prev head asking to attach itself before
    - Prev head changes state (if HEAD_FRESH -> MID, if ONLY_FRESH -> TAIL)
    - Updates Back ptr and sends back ACK to A
    - A sets fwd ptr to prev head and state to HEAD_FRESH

(contd.)

- GONE:
  - Dir send ptr of prev hd (B) and sets hd ptr to A
  - A goes to Pending, sends Req to B
  - B changes state (If HEAD_DIRTY -> MID, ONLY_DIRTY-> TAIL)
  - Sets Back ptr to A, and returns data to A
  - A updates the data, sets state to HEAD_DIRTY, sets fwd ptr to B

- What if B is “Busy/Pending” when A’s request reaches?
  - B just updates its Back ptr.
  - When it becomes non-busy, it can respond to A (which in turn could respond to some one else, etc.)

Write Miss at A

- Only the “Head” is allowed to write and issue invalidates!
- If A is Not Head,
  - Roll out of list
  - Add to Head (List construction) – NOTE: Dir must be in GONE before doing invalidates
- Block at A is:
  - HEAD DIRTY:
    - Change to Pending
    - Purge Linked List (send Inval to one after another, getting back ACK and Next in line from each)
  - HEAD FRESH:
    - Change to Pending
    - Send Req to Home, Home changes from FRESH->GONE and Acks
    - Then do what was done for HEAD_DIRTY actions

(contd.)

- What if HOME is not FRESH, when Head's request reaches it?
  - Home NACKS, requester deletes itself from list, and tries to re-attach as HEAD_DIRTY or ONLY_DIRTY state

- Write Back/Replacement
  - Change to Pending
  - Send Requests (with appropriate ptr info) to either side
  - After getting ACKs, you can drop out
  - Use a priority scheme (more priority for tail side) to avoid Deadlocks

Order without Deadlock?

- SCI: serialize at home, use distributed pending list per line
  - just like sharing list: requestor adds itself to tail
  - no limited buffer, so no deadlock
  - node with request satisfied passes it on to next node in list
  - low space overhead, and fair
  - But high latency
    - on read, could reply to all requestors at once otherwise
- Memory-based schemes
  - use dedicated queues within node to avoid blocking requests that depend on each other
  - DASH: forward to dirty node, let it determine order
    - it replies to requestor directly, sends writeback to home
    - what if line written back while forwarded request is on the way?

Cache-based Schemes

- Protocol more complex
  - e.g. removing a line from list upon replacement
    - must coordinate and get mutual exclusion on adjacent nodes' ptrs
    - they may be replacing their same line at the same time
- Higher latency and overhead
  - every protocol action needs several controllers to do something
  - in memory-based, reads handled by just home
  - sending of invals serialized by list traversal
    - increases latency
- But IEEE Standard and being adopted
  - Convex Exemplar

Verification

- Coherence protocols are complex to design and implement
  - much more complex to verify
- Formal verification
- Generating test vectors
  - random
  - specialized for common and corner cases
  - using formal verification techniques