

---

## L10...

# Cache Coherence in Scalable Machines

## Scalable Cache Coherent Systems

- Scalable, distributed memory plus coherent replication
- Scalable distributed memory machines
  - P-C-M nodes connected by network
  - communication assist interprets network transactions, forms interface
- Final point was 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

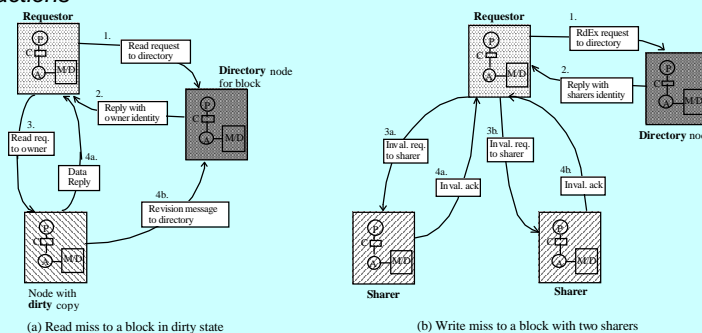
- All of (a), (b), (c) done through broadcast on bus
  - faulting processor sends out a “search”
  - others respond to the search probe and take necessary action
- Could do it in scalable network too
  - broadcast to all processors, and let them respond
- Conceptually simple, but broadcast doesn't scale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

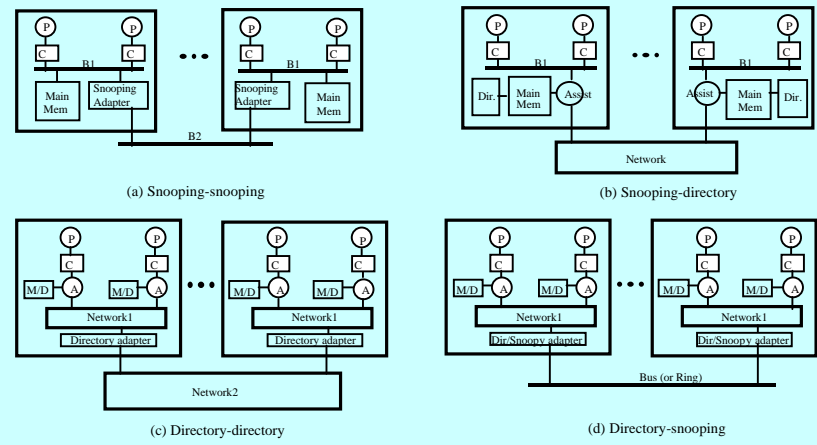


- Many alternatives for organizing directory information

# A Popular Middle Ground

- Two-level “hierarchy”
- Individual nodes are multiprocessors, connected non-hierarchically
  - 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

# Example Two-level Hierarchies



## 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

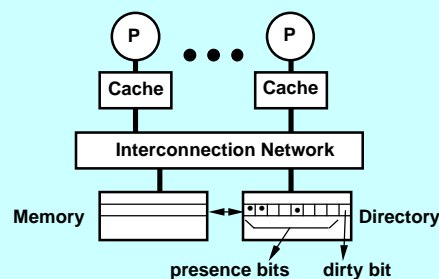
## Disadvantages of Coherent MP Nodes

- 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

## Outline

- Overview of directory-based approaches
- Directory Protocols
  - Correctness, including serialization and consistency
  - Implementation
  - study through case Studies: SGI Origin2000, Sequent NUMA-Q
  - discuss alternative approaches in the process
- Synchronization
- Implications for parallel software
- Relaxed memory consistency models
- Alternative approaches for a coherent shared address space

## 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

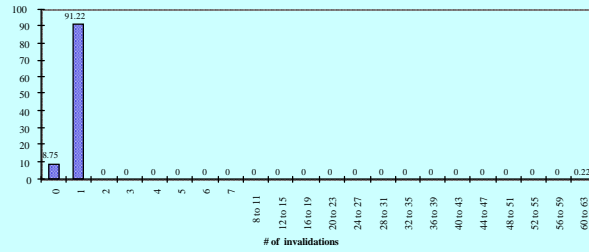
- **Scaling of memory and directory bandwidth provided**
  - Centralized directory is bandwidth bottleneck, just like centralized memory
  - How to maintain directory information in distributed way?
- **Scaling of performance characteristics**
  - traffic: no. of network transactions each time protocol 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**

## Insights into Directories

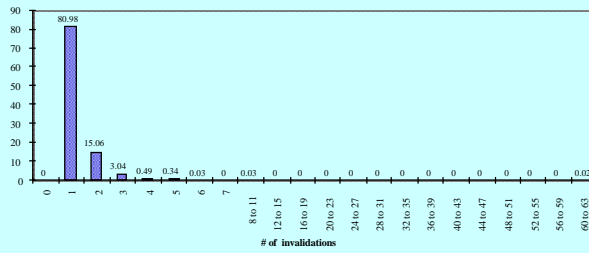
- **Inherent program characteristics:**
  - determine whether directories provide big advantages over broadcast
  - provide insights into how to organize and store directory information
- **Characteristics that matter**
  - frequency of write misses?
  - how many sharers on a write miss
  - how these scale

# Cache Invalidation Patterns

LU Invalidation Patterns

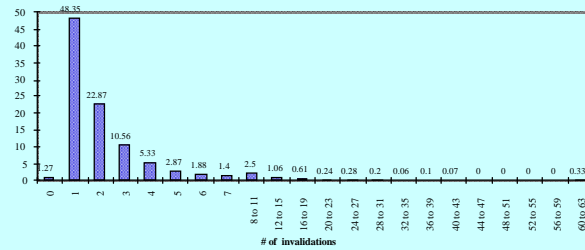


Ocean Invalidation Patterns

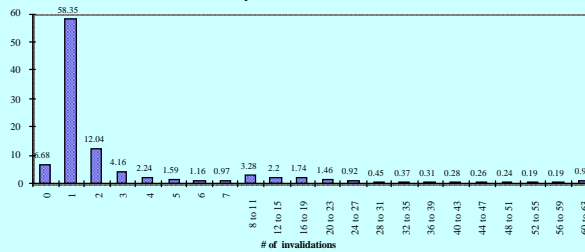


# Cache Invalidation Patterns

Barnes-Hut Invalidation Patterns



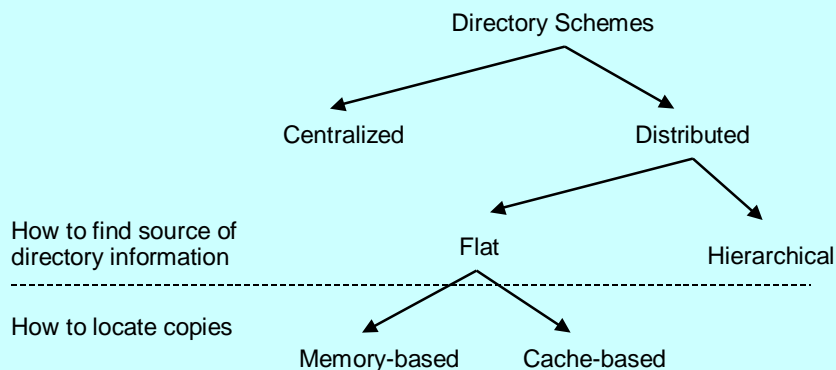
Radiosity Invalidation Patterns



## 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

## Organizing Directories

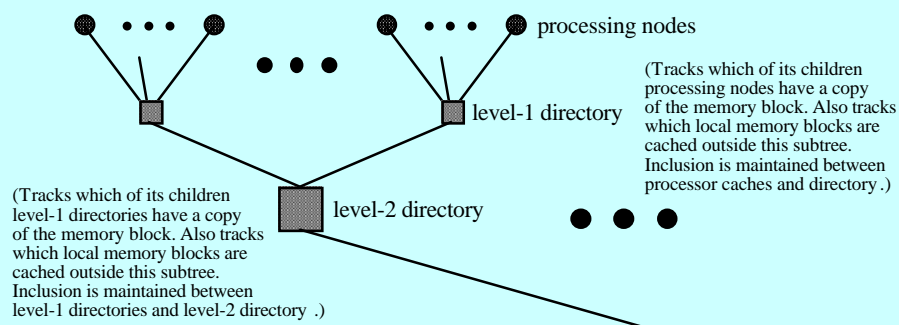


Let's see how they work and their scaling characteristics with P

## 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 action 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 hierarchical snooping, but point-to-point messages between children and parents

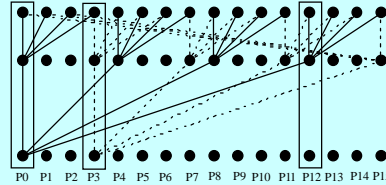
## How Hierarchical Directories Work



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

## Scaling Properties

- Bandwidth: root can become bottleneck
  - can use multi-rooted directories in general interconnect
- Traffic (no. of messages):
  - depends on locality in hierarchy
  - can be bad at low end
    - $4 \cdot \log P$  with only one copy!
  - may be able to exploit message combining
- Latency:
  - also depends on locality in hierarchy
  - can be better in large machines when don't have to travel far (distant home)
  - but can have multiple network transactions along hierarchy, and multiple directory lookups along the way
- Storage overhead:

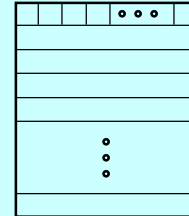


## How Is Location of Copies Stored?

- 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  $P \cdot M$

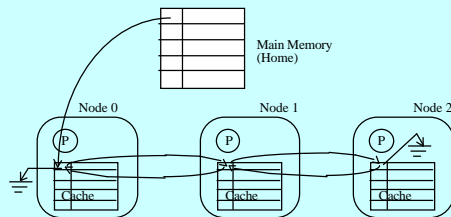


## Reducing Storage Overhead

- 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  $P \cdot M$ , 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
  - $P=1024 \Rightarrow$  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  $\gg$  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

## 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 invalids 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
- Other properties (discussed later):
  - good: mature, IEEE Standard, fairness
  - bad: complex

## 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 explicit 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 dir. 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
  - 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
- 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

---

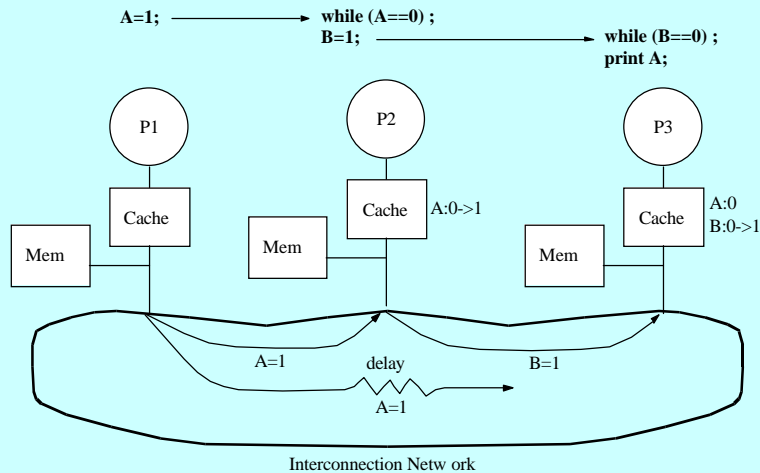
- on a bus, multiple copies but serialization by bus imposed order
- on scalable without coherence, main memory module determined order
- could use main memory module here too, but multiple copies
  - valid copy of data may not be in main memory
  - reaching main memory in one order does not mean will reach valid copy in that order
  - serialized in one place doesn't mean serialized wrt all copies (later)

## Sequential Consistency

---

- bus-based:
  - write completion: wait till gets on bus
  - write atomicity: bus plus buffer ordering provides
- in 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



## 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

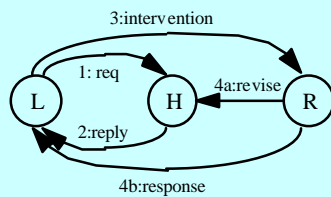
## Performance

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

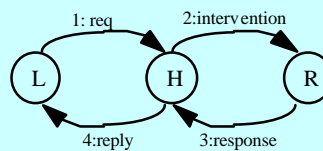
Care about how these scale with the number of nodes

## Protocol Enhancements for Latency

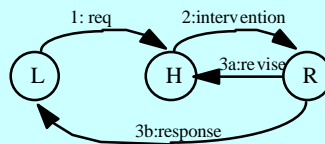
- Forwarding messages: memory-based protocols



(a) *Strict request-reply*



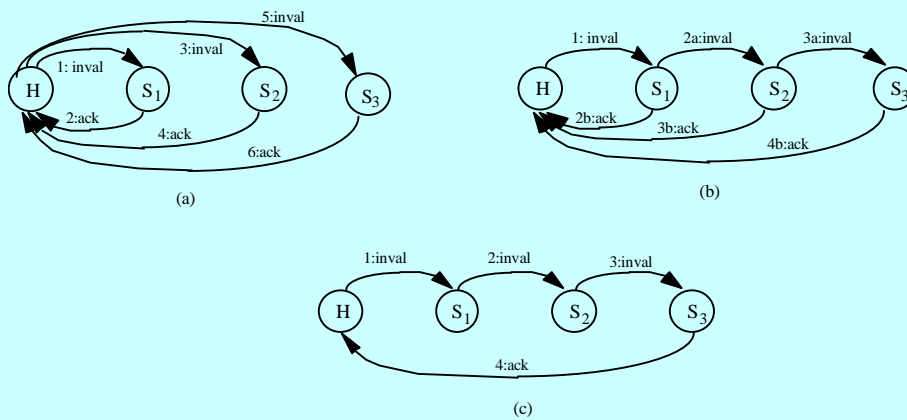
(a) *Intervention forwarding*



(a) *Reply forwarding*

## Protocol Enhancements for Latency

- Forwarding messages: cache-based protocols



## 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...

## Complexity

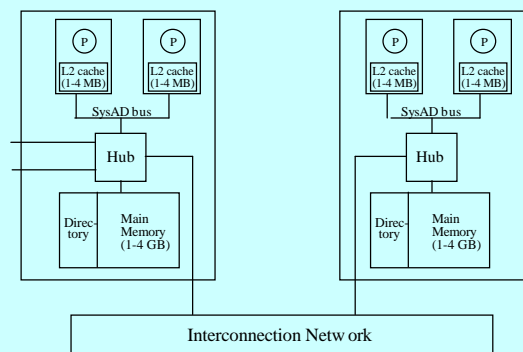
---

- 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
- Let's look at memory- and cache-based more deeply

## Flat, Memory-based Protocols

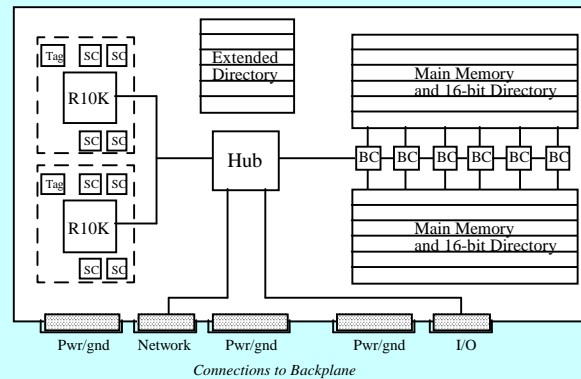
- Use SGI Origin2000 Case Study
  - Protocol similar to Stanford DASH, but with some different tradeoffs
  - Also Alewife, FLASH, HAL
- Outline:
  - System Overview
  - Coherence States, Representation and Protocol
  - Correctness and Performance Tradeoffs
  - Implementation Issues
  - Quantitative Performance Characteristics

## Origin2000 System Overview



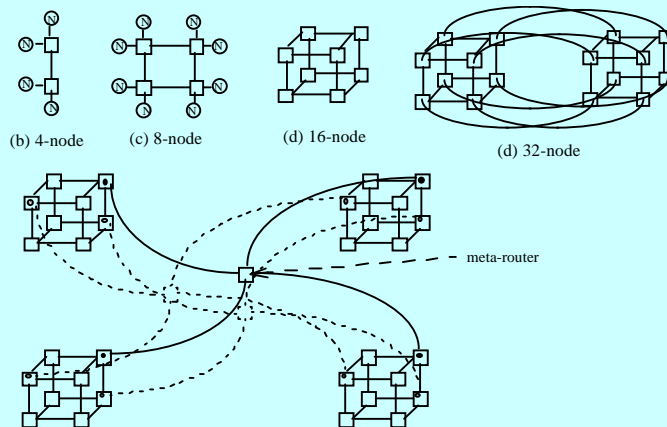
- Single 16"-by-11" PCB
- Directory state in same or separate DRAMs, accessed in parallel
- Upto 512 nodes (1024 processors)
- With 195MHz R10K processor, peak 390MFLOPS or 780 MIPS per proc
- Peak SysAD bus bw is 780MB/s, so also Hub-Mem
- Hub to router chip and to Xbow is 1.56 GB/s (both are of-board)

## Origin Node Board



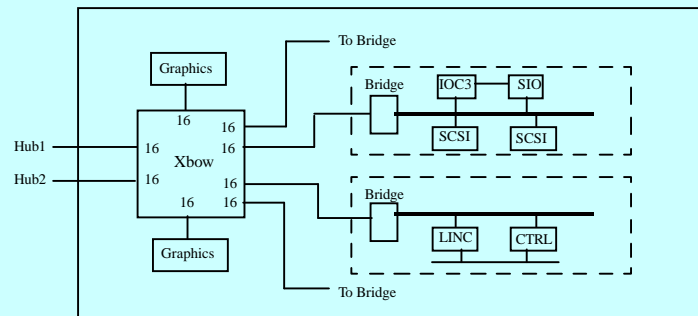
- Hub is 500K-gate in 0.5  $\mu$  CMOS
- Has outstanding transaction buffers for each processor (4 each)
- Has two block transfer engines (memory copy and fill)
- Interfaces to and connects processor, memory, network and I/O
- Provides support for synch primitives, and for page migration (later)
- Two processors within node not snoopy-coherent (motivation is cost)

## 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 I/O



- Xbow is 8-port crossbar, connects two Hubs (nodes) to six cards
- Similar to router, but simpler so can hold 8 ports
- Except graphics, most other devices connect through bridge and bus
  - can reserve bandwidth for things like video or real-time
- Global I/O space: any proc can access any I/O device
  - through uncached memory ops to I/O space or coherent DMA
  - any I/O device can write to or read from any memory (comm thru routers)

## 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

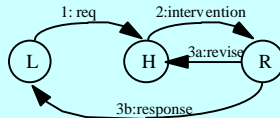
- Cache states: MESI
- Seven directory states
  - *unowned*: no cache has a copy, memory copy is valid
  - *shared*: one or more caches has a shared copy, memory is valid
  - *exclusive*: one cache (pointed to) has block in modified or exclusive state
  - three *pending* or *busy* states, one for each 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  $P \cdot k$  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 will be 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 its 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

## 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

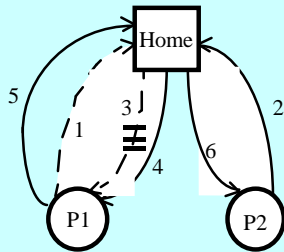
- Serialization of operations
- Deadlock
- Livelock
- Starvation

## 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 back 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 actions 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 receives 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 invalidate, 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 output 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

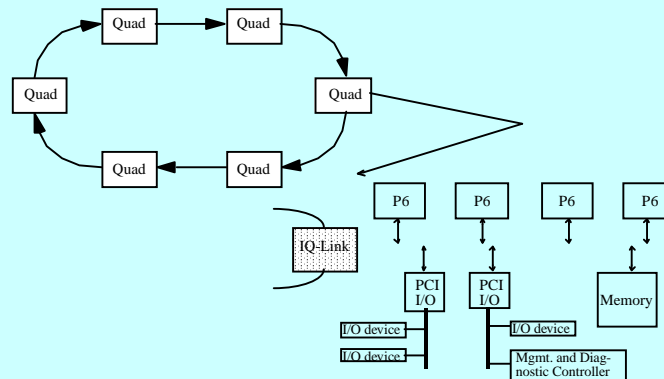
---

- 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)

## Flat, Cache-based Protocols

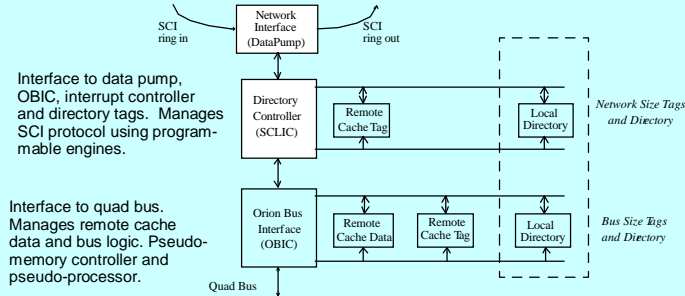
- Use Sequent NUMA-Q Case Study
  - Protocol is Scalable Coherent Interface across nodes, snooping with node
  - Also Convex Exemplar, Data General
- Outline:
  - System Overview
  - SCI Coherence States, Representation and Protocol
  - Correctness and Performance Tradeoffs
  - Implementation Issues
  - Quantitative Performance Characteristics

## NUMA-Q System Overview



- Use of high-volume SMPs as building blocks
- Quad bus is 532MB/s split-transaction 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

## NUMA-Q IQ-Link Board

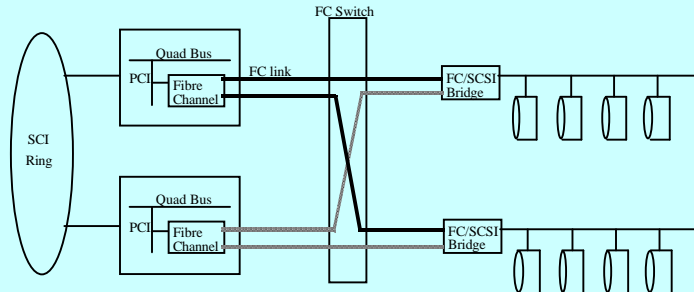


- Plays the role of Hub Chip in SGI Origin
- Can generate interrupts between quads
- Remote cache (visible to SC I) block size is 64 bytes (32MB, 4-way)
  - processor caches not visible (snoopy-coherent and with remote cache)
- Data Pump (GaAs) implements SCI transport, pulls off relevant packets

## 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

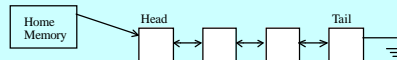
## NUMA-Q I/O



- Machine intended for commercial workloads; I/O is very important
- Globally addressible I/O, as in Origin
  - very convenient for commercial workloads
- Each PCI bus is half as wide as memory bus and half clock speed
- I/O devices on other nodes can be accessed through SCI or Fibre Channel
  - I/O through reads and writes to PCI devices, not DMA
  - Fibre channel can also be used to connect multiple NUMA-Q, or to shared disk
  - If I/O through local FC fails, OS can route it through SCI to other node and FC

## 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
  - keeping those coherent with remote cache is done by OBIC and SCLIC



## 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

---

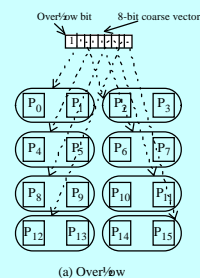
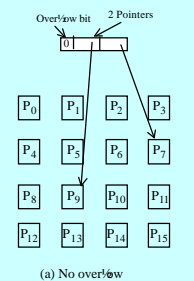
- **Protocol more complex**
  - e.g. removing a line from list upon replacement
    - must coordinate and get mutual exclusion on adjacent nodes' ptrs
    - they may be replacing their same line at the same time
- **Higher latency and overhead**
  - every protocol action needs several controllers to 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

## Overflow Schemes for Limited Pointers

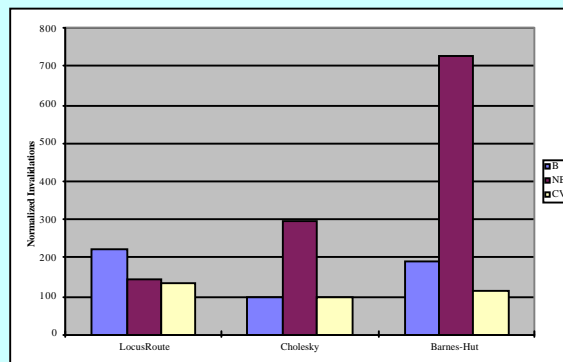
- Broadcast ( $Dir_iB$ )
  - broadcast bit turned on upon overflow
  - bad for widely-shared frequently read data
- No-broadcast ( $Dir_iNB$ )
  - on overflow, new sharer replaces one of the old ones (invalidated)
  - bad for widely read data
- Coarse vector ( $Dir_iCV$ )
  - change representation to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



## 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
      - 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



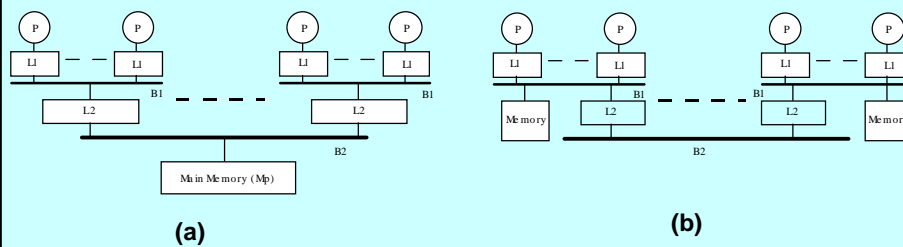
- 64 procs, 4 pointers, normalized to full-bit-vector
- Coarse vector quite robust
- General conclusions:
  - full bit vector simple and good for moderate-scale
  - several schemes should be fine for large-scale, no clear winner yet

## Reducing Height: Sparse Directories

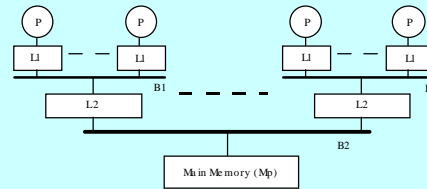
- Reduce M term in  $P \cdot M$
- Observation: total number of cache entries  $\ll$  total amount of memory.
  - most directory entries are idle most of the time
  - 1MB cache and 64MB per node  $\Rightarrow$  98.5% of entries are idle
- Organize directory as a cache
  - but no need for backup store
    - 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

## Hierarchical Snoopy Cache Coherence

- Simplest way: hierarchy of buses; snoopy coherence at each level.
  - or rings
- Consider buses. Two possibilities:
  - (a) All main memory at the global (B2) bus
  - (b) Main memory distributed among the clusters



## Bus Hierarchies with Centralized Memory



B1 follows standard snoopy protocol

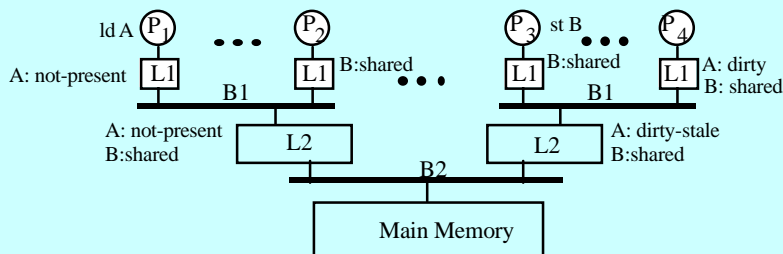
Need a monitor per B1 bus

- decides what transactions to pass back and forth between buses
- acts as a filter to reduce bandwidth needs

Use L2 cache

- Much larger than L1 caches (set assoc). Must maintain inclusion.
- Has dirty-but-stale bit per line
- L2 cache can be DRAM based, since fewer references get to it.

## Examples of References

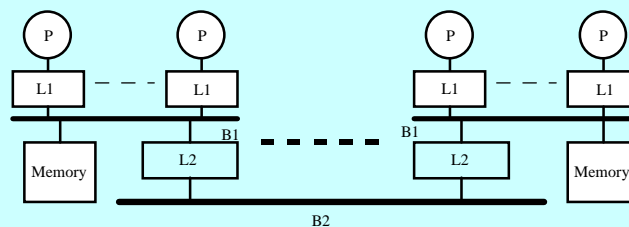


- How issues (a) through (c) are handled across clusters
  - (a) enough info about state in other clusters in dirty-but-stale bit
  - (b) to find other copies, broadcast on bus (hierarchically); they snoop
  - (c) comm with copies performed as part of finding them
- Ordering and consistency issues trickier than on one bus

## Advantages and Disadvantages

- **Advantages:**
  - Simple extension of bus-based scheme
  - Misses to main memory require single traversal to root of hierarchy
  - Placement of shared data is not an issue
  
- **Disadvantages:**
  - Misses to local data (e.g., stack) also traverse hierarchy
    - higher traffic and latency
  - Memory at global bus must be highly interleaved for bandwidth

## Bus Hierarchies with Distributed Memory

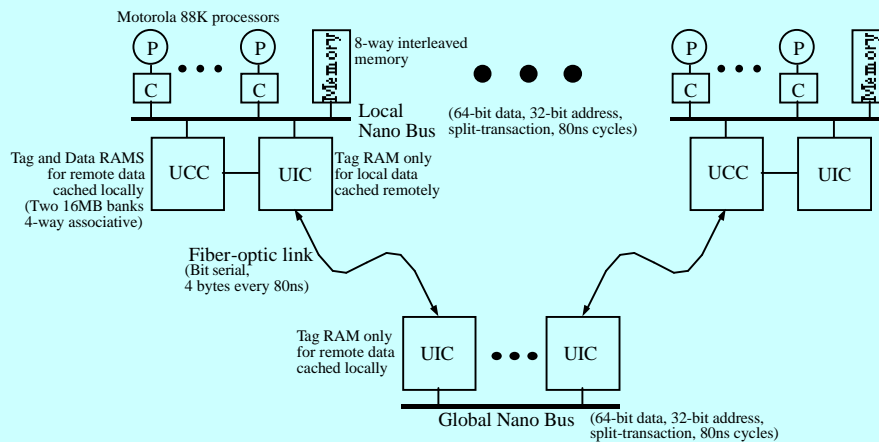


- Main memory distributed among clusters.
  - cluster is a full-fledged bus-based machine, memory and all
  - automatic scaling of memory (each cluster brings some with it)
  - good placement can reduce global bus traffic and latency
    - but latency to far-away memory may be larger than to root

## Maintaining Coherence

- L2 cache works fine as before for remotely allocated data
- What about locally allocated data that are cached remotely
  - don't enter L2 cache
- Need mechanism to monitor transactions for these data
  - on B1 and B2 buses
- Let's examine a case study

## Case Study: Encore Gigamax



## Cache Coherence in Gigamax

---

- Write to local-bus is passed to global-bus if:
  - data allocated in remote Mp
  - allocated local but present in some remote cache
- Read to local-bus passed to global-bus if:
  - allocated in remote Mp, and not in cluster cache
  - allocated local but dirty in a remote cache
- Write on global-bus passed to local-bus if:
  - allocated in to local Mp
  - allocated remote, but dirty in local cache
- ...
- Many race conditions possible (write-back going out as request coming in)

## Hierarchies of Rings (e.g. KSR)

---

- Hierarchical ring network, not bus
- Snoop on requests passing by on ring
- Point-to-point structure of ring implies:
  - potentially higher bandwidth than buses
  - higher latency
- (see Chapter 6 for details of rings)
- KSR is Cache-only Memory Architecture (discussed later)

## Hierarchies: Summary

---

- Advantages:

- Conceptually simple to build (apply snooping recursively)
- Can get merging and combining of requests in hardware

- Disadvantages:

- Low bisection bandwidth: bottleneck toward root
  - patch solution: multiple buses/rings at higher levels
- Latencies often larger than in direct networks