Cache Coherence in Scalable Machines

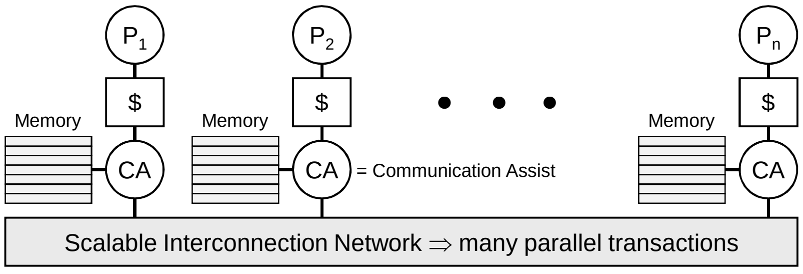
COE 502 – Parallel Processing Architectures

Prof. Muhamed Mudawar

Computer Engineering Department

King Fahd University of Petroleum and Minerals

Generic Scalable Multiprocessor


P_1
P_2
P_n
$
Memory
CA
= Communication Assist
Scalable Interconnection Network ⇒ many parallel transactions

- ❖ Scalable distributed memory machine
  - ➢ P-C-M nodes connected by a scalable network
  - ➢ Scalable memory bandwidth at reasonable latency
- ❖ Communication Assist (CA)
  - ➢ Interprets network transactions, forms an interface
  - ➢ Provides a shared address space in hardware

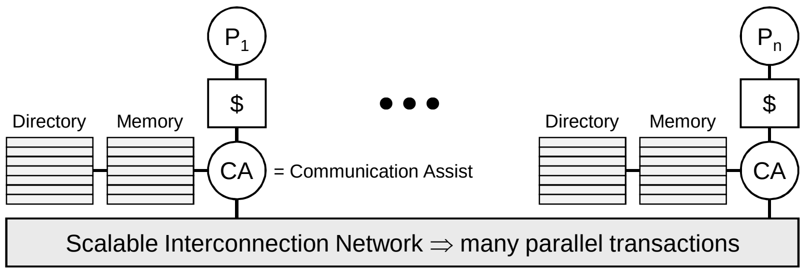
What Must a Coherent System do?

- ❖ Provide set of states, transition diagram, and actions
- ❖ Determine when to invoke coherence protocol
  - ➢ 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
      - ▪ Read miss, Write miss, Writing to a shared block
- ❖ Manage coherence protocol
  1. Find information about state of block in other caches
     - ✧ Whether need to communicate with other cached copies
  2. Locate the other cached copies
  3. Communicate with those copies (invalidate / update)
  - ➢ Handled differently by different approaches

Bus-Based Cache Coherence

- ❖ Functions 1, 2, and 3 are accomplished through …
  - ➢ Broadcast and snooping on bus
  - ➢ Processor initiating bus transaction sends out a "search"
  - ➢ Others respond and take necessary action
- ❖ Could be done in a scalable network too
  - ➢ Broadcast to all processors, and let them respond
- ❖ Conceptually simple, but broadcast doesn't scale
  - ➢ On a bus, bus bandwidth doesn't scale
  - ➢ On a scalable network
    - ✧ Every **access fault** leads to at least ***p*** **network transactions**
- ❖ Scalable Cache Coherence needs
  - ➢ Different mechanisms to manage protocol

Directory-Based Cache Coherence



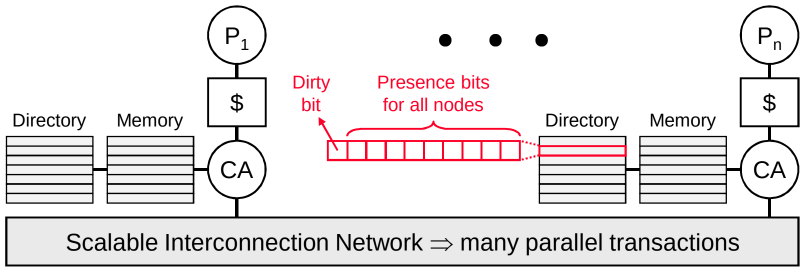
P_1
P_n
$
Directory
Memory
CA
= Communication Assist
Scalable Interconnection Network ⇒ many parallel transactions

- ❖ Scalable cache coherence is based on directories
- ❖ Distributed directories for distributed memories
- ❖ Each cache-line-sized block of a memory …
  - ➢ Has a directory entry that keeps track of …
    - ✧ State and the nodes that are currently sharing the block

Simple Directory Scheme

- ❖ Simple way to organize a directory is to associate …
  - ➢ Every memory block with a corresponding directory entry
  - ➢ To keep track of copies of cached blocks and their states
- ❖ On a miss
  - ➢ Locate home node and the corresponding directory entry
  - ➢ Look it up to find its state and the nodes that have copies
  - ➢ Communicate with the nodes that have copies if necessary
- ❖ On a read miss
  - ➢ Directory indicates from which node data may be obtained
- ❖ On a write miss
  - ➢ Directory identifies shared copies to be invalidated/updated
- ❖ Many alternatives for organizing directory

Directory Information



P1
$
CA
Directory
Memory
Dirty bit
Presence bits for all nodes
Directory
Memory
Pn
$
CA
Scalable Interconnection Network ⇒ many parallel transactions

- ❖ A simple organization of a directory entry is to have
  - ➢ Bit vector of *p* presence bits for each of the *p* nodes
    - ✧ Indicating which nodes are sharing that block
  - ➢ One or more state bits per block reflecting memory view
    - ✧ One dirty bit is used indicating whether block is modified
    - ✧ If dirty bit is 1 then block is in modified state in only one node

Definitions

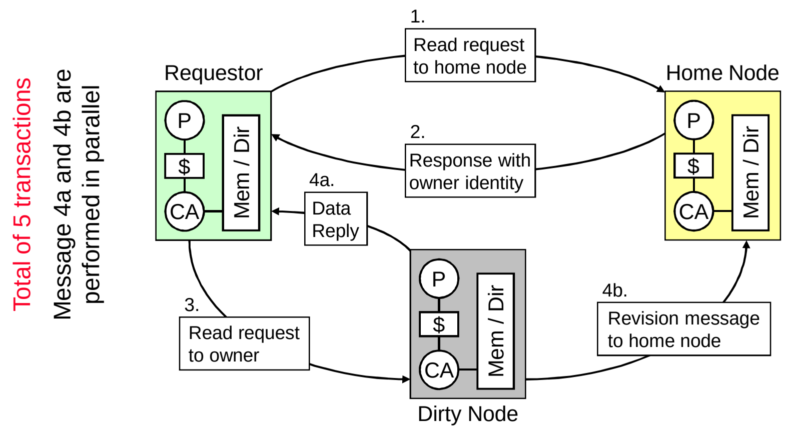
- ❖ Home Node
  - ➢ Node in whose main memory the block is allocated
- ❖ Dirty Node
  - ➢ Node that has copy of block in its cache in modified state
- ❖ Owner Node
  - ➢ Node that currently holds the valid copy of a block
  - ➢ Can be either the home node or dirty node
- ❖ Exclusive Node
  - ➢ Node that has block in its cache in an exclusive state
  - ➢ Either exclusive clean or exclusive modified
- ❖ Local or Requesting Node
  - ➢ Node that issued the request for the block

Basic Operation on a Read Miss

- Read miss by processor **i**
  - Requestor sends **read request** message to **home** node
  - Assist looks-up directory entry, if **dirty-bit** is **OFF**
    - Reads block from memory
    - Sends **data reply** message containing to requestor
    - Turns **ON Presence[i]**
  - If **dirty-bit** is **ON**
    - Requestor sends **read request** message to **dirty** node
    - Dirty node sends **data reply** message to requestor
    - Dirty node also sends **revision** message to **home** node
    - Cache block state is changed to **shared**
    - Home node updates its main memory and directory entry
    - Turns **dirty-bit OFF**
    - Turns **Presence[i] ON**

Read Miss to a Block in Dirty State

- The number 1, 2, and so on show serialization of transactions
- Letters to same number indicate overlapped transactions


Total of 5 transactions
Message 4a and 4b are performed in parallel
1. Read request to home node
Requestor
Home Node
2. Response with owner identity
4a. Data Reply
3. Read request to owner
4b. Revision message to home node
Dirty Node
P
$
CA
Mem / Dir

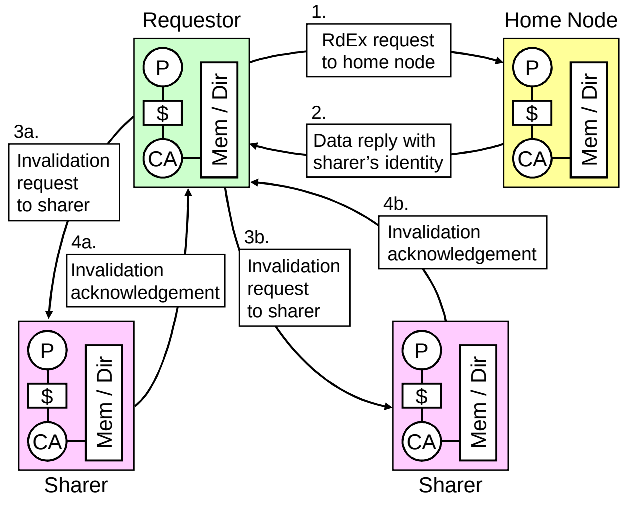
Basic Operation on a Write Miss

- Write miss caused by processor **i**
  - Requestor sends **read exclusive** message to **home** node
  - Assist looks up directory entry, if **dirty-bit** is **OFF**
    - **Home** node sends a **data reply** message to processor **i**
      - Message contains data and presence bits identifying **all sharers**
    - Requestor node sends invalidation messages to all sharers
    - Sharer nodes invalidate their cached copies and
      - Send acknowledgement messages to requestor node
  - If **dirty-bit** is **ON**
    - **Home** node sends a response message identifying **dirty** node
    - Requestor sends a **read exclusive** message to **dirty** node
    - **Dirty** node sends a **data reply** message to processor **i**
      - And changes its cache state to **Invalid**
  - In both cases, **home** node clears presence bits
    - But turns **ON Presence[i]** and **dirty-bit**

Write Miss to a Block with Sharers

Protocol is orchestrated by the **assists**

Called **coherence** or **directory controllers**


Requestor
Home Node
1.
RdEx request to home node
2.
Data reply with sharer's identity
3a.
Invalidation request to sharer
3b.
Invalidation request to sharer
4a.
Invalidation acknowledgement
4b.
Invalidation acknowledgement
P
$
CA
Mem / Dir
Sharer
Sharer

Data Sharing Patterns

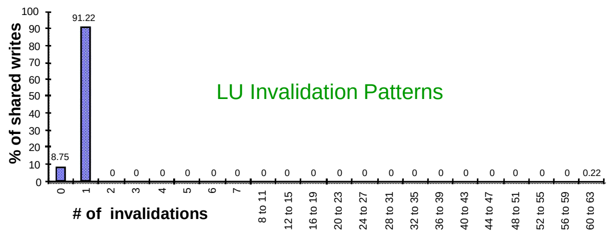
- ❖ Provide insight into directory requirements
- ❖ If most misses involve *O*(*P*) transactions then
  - ➢ Broadcast might be a good solution
- ❖ However generally, there are few sharers at a write
- ❖ Important to understand two aspects of data sharing
  - ➢ Frequency of shared writes or invalidating writes
    - ✧ On a write miss or when writing to a block in the shared state
    - ✧ Called invalidation frequency in invalidation-based protocols
  - ➢ Distribution of sharers called the invalidation size distribution
- ❖ Invalidation size distribution also provides …
  - ➢ Insight into how to organize and store directory information

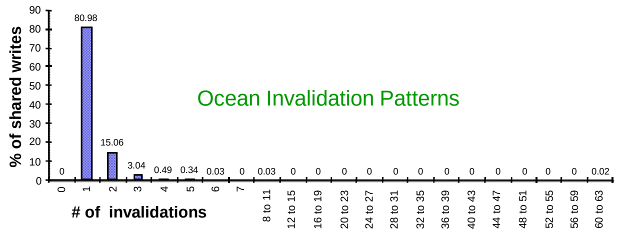
Cache Invalidation Patterns

Simulation running on 64 processors

Figures show invalidation size distribution

MSI protocol is used

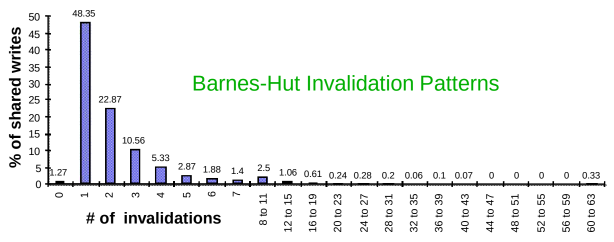


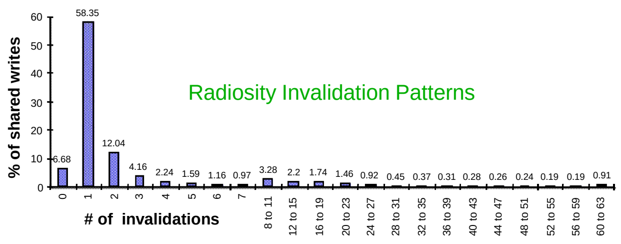


Cache Invalidation Patterns – cont'd

Infinite per-processor caches are used

To capture inherent sharing patterns





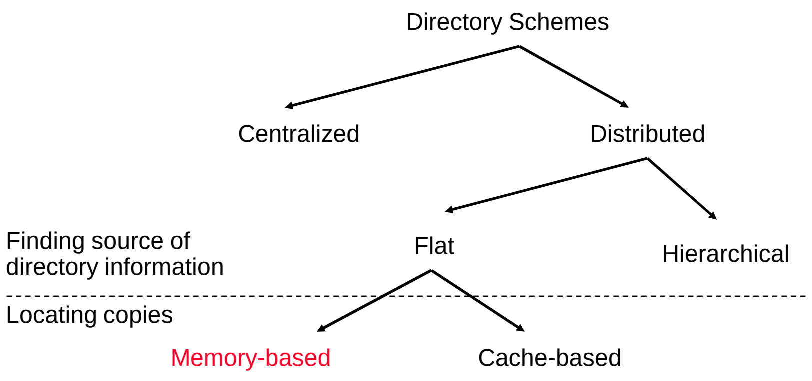
Framework for Sharing Patterns

- ❖ Shared-Data access patterns can be categorized:
  - ➢ Code and read-only data structures are never written
    - ✧ Not an issue for directories
  - ➢ Producer-consumer
    - ✧ One processor produces data and others consume it
    - ✧ Invalidation size is determined by number of consumers
  - ➢ Migratory data
    - ✧ Data migrates from one processor to another
    - ✧ Example: computing a global sum, where sum migrates
    - ✧ Invalidation size is small (typically 1) even as *P* scales
  - ➢ Irregular read-write
    - ✧ Example: distributed task-queue (processes probe head ptr)
    - ✧ Invalidations usually remain small, though frequent

Sharing Patterns Summary

- ❖ Generally, few sharers at a write
  - ➢ Scales slowly with *P*
- ❖ A write may send 0 invalidations in MSI protocol
  - ➢ Since block is loaded in shared state
  - ➢ This would not happen in MESI protocol
- ❖ Infinite per-processor caches are used
  - ➢ To capture inherent sharing patterns
  - ➢ Finite caches send replacement hints on block replacement
    - ✧ Which turn off presence bits and reduce # of invalidations
    - ✧ However, traffic will not be reduced
- ❖ Non-zero frequencies of very large invalidation sizes
  - ➢ Due to spinning on a synchronization variable by all

Alternatives for Directories



Directory information co-located with memory module in home node for each memory block

Examples: Stanford DASH/FLASH, MIT Alewife, SGI Origin, HAL

Caches holding a copy of the memory block form a doubly linked list
Directory hold pointer to head of list

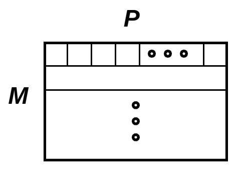
Examples:
IEEE SCI, Sequent NUMA-Q

Flat Memory-based Schemes

- ❖ Information about cached (shared) copies is
  - ➢ Stored with the block at the home node
- ❖ Performance Scaling
  - ➢ Traffic on a shared write: proportional to number of sharers
  - ➢ Latency on shared write: can issue invalidations in parallel
- ❖ Simplest Representation: one presence bit per node
  - ➢ Storage overhead is proportional to $P \times M$
    - ✧ For M memory blocks in memory, and ignoring state bits
  - ➢ Directory storage overhead scale poorly with P
    - ✧ Given 64-byte cache block size
    - ✧ For 64 nodes: 64 presence bits / 64 bytes = 12.5% overhead
    - ✧ 256 nodes: 256 presence bits / 64 bytes = 50% overhead
    - ✧ 1024 nodes: 1024 presence bits / 64 bytes = 200% overhead

Reducing Directory Storage

- ❖ Optimizations for full bit vector schemes
  - ➢ Increase cache block size
    - ✧ Reduces storage overhead proportionally
  - ➢ Use more than one processor (SMP) per node
    - ✧ Presence bit is per node, not per processor
  - ➢ But still scales as $P \times M$
    - ✧ Reasonable and simple enough for all but very large machines
    - ✧ Example: 256 processors, 4-processor nodes, 128-byte lines
      - ⇒ Overhead = (256/4) / (128*8) = 64 / 1024 = 6.25% (attractive)
- ❖ Need to reduce "width"
  - ➢ Addressing the P term
- ❖ Need to reduce "height"
  - ➢ Addressing the M term

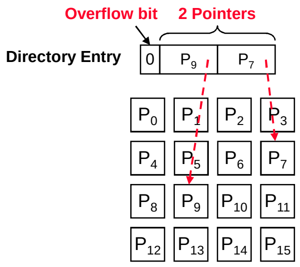


Directory Storage Reductions

- ❖ Width observation:
  - Most blocks cached (shared) by only few nodes
  - Don't need a bit per node
  - Sharing patterns indicate a few pointers should suffice
    - Entry can contain a few (5 or so) pointers to sharing nodes
  - $P = 1024 \Rightarrow 10$ bit pointers
    - 5 pointers need only 50 bits rather than 1024 bits
  - 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

Limited Pointers

- ❖ Dir_i : only i pointers are used per entry
  - Works in most cases when # sharers ≤ i
- ❖ Overflow mechanism needed when # sharers > i
  - Overflow bit indicates that number of sharers exceed i
- ❖ Overflow methods include
  - Broadcast
  - No-broadcast
  - Coarse Vector
  - Software Overflow
  - Dynamic Pointers



Overflow Methods

❖ Broadcast (Dir_iB)

- Broadcast bit turned on upon overflow
- Invalidations broadcast to all nodes when block is written
  - Regardless of whether or not they were sharing the block
- Network bandwidth may be wasted for overflow cases
- Latency increases if processor wait for acknowledgements

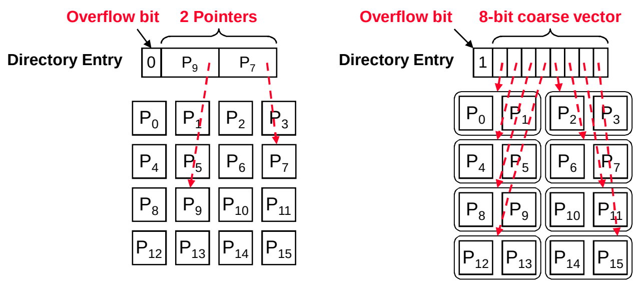
❖ No-Broadcast (Dir_iNB)

- Avoids broadcast by never allowing # of sharers to exceed i
- When number of sharers is equal to i
  - New sharer replaces (invalidates) one of the old ones
  - And frees up its pointer in the directory entry
- Drawback: does not deal with widely shared data

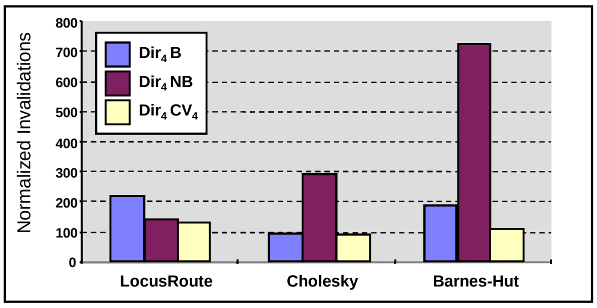
Coarse Vector Overflow Method

❖ Coarse vector (Dir_iCV_r)

- Uses i pointers in its initial representation
- But on overflow, changes representation to a coarse vector
- Each bit indicates a unique group of r nodes
- On a write, invalidate all r nodes that a bit corresponds to



Robustness of Coarse Vector



- ❖ 64 processors (one per node), 4 pointers per entry
- ❖ 16-bit coarse vector, 4 processors per group
- ❖ Normalized to full-bit-vector (100 invalidations)
- ❖ Conclusion: coarse vector scheme is quite robust

Software Overflow Schemes

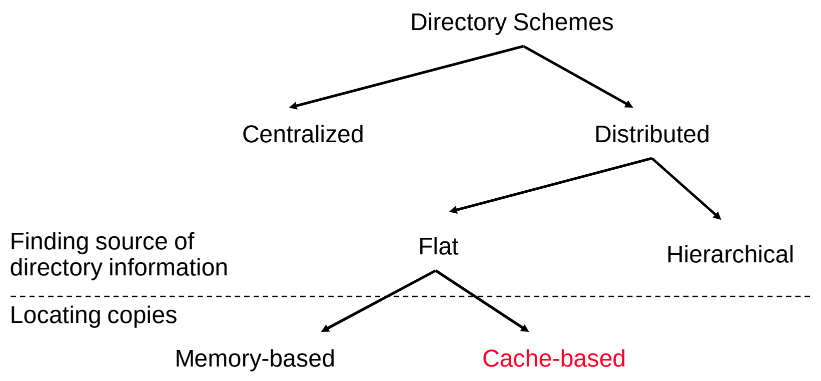
- ❖ Software ($Dir_i SW$)
  - ➢ On overflow, trap to system software
  - ➢ Overflow pointers are saved in local memory
  - ➢ Frees directory entry to handle **i** new sharers in hardware
  - ➢ Used by MIT Alewife: 5 pointers, plus one bit for local node
  - ➢ But large overhead for interrupt processing
    - ✧ 84 cycles for 5 invalidations, but 707 cycles for 6.
- ❖ Dynamic Pointers ($Dir_i DP$) is a variation of $Dir_i SW$
  - ➢ Each directory entry contains extra pointer to local memory
  - ➢ Extra pointer uses a free list in special area of memory
  - ➢ Free list is manipulated by hardware, not software
  - ➢ Example: Stanford FLASH multiprocessor

Reducing Directory Height

- ❖ Sparse Directory: reducing M term in P × M
- ❖ Observation: cached entries << main memory
  - ➢ Most directory entries are unused most of the time
  - ➢ Example: 2MB cache and 512MB local memory per node

    => 510 / 512 or 99.6% of directory entries are unused
- ❖ Organize directory as a cache to save space
  - ➢ Dynamically allocate directory entries, as cache lines
  - ➢ Allow use of faster SRAMs, instead of slower DRAMs
    - ✧ Reducing access time to directory information in critical path
  - ➢ When an entry is replaced, send invalidations to all sharers
  - ➢ Handles references from potentially all processors
  - ➢ Essential to be large enough and with enough associativity

Alternatives for Directories



Directory information co-located with memory module in home node for each memory block

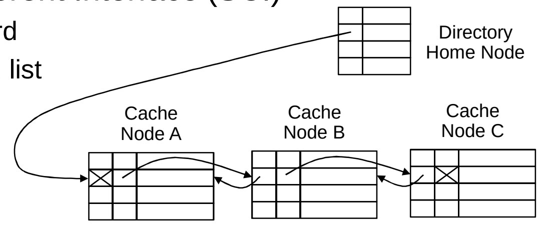
Examples: Stanford DASH/FLASH, MIT Alewife, SGI Origin, HAL

Caches holding a copy of the memory block from a doubly linked list; Directory holds pointer to head of list

Examples:
IEEE SCI, Sequent NUMA-Q

Flat Cache-based Schemes

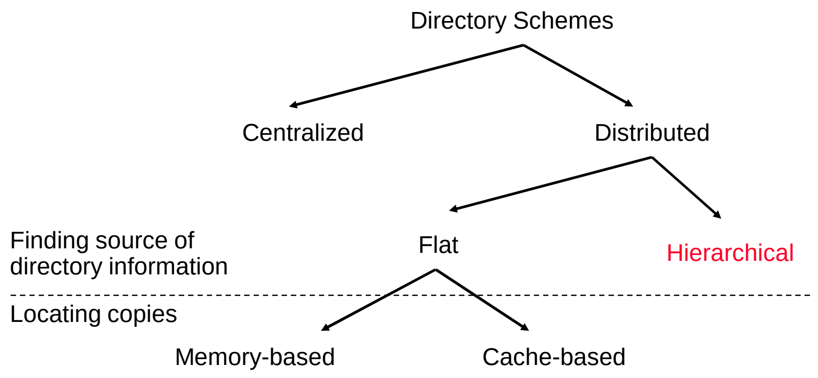
- ❖ How they work:
  - ➢ Home node only holds pointer to rest of directory info
  - ➢ Distributed doubly linked list, weaves through caches
    - ✧ Cache line has two associated pointers
    - ✧ Points to next and previous nodes with a copy
  - ➢ On read, add yourself to head of the list
  - ➢ On write, propagate chain of invalidations 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 assist
    - ✧ Home and first sharer on list
- ❖ Storage overhead
  - ➢ Quite good scaling along both axes
  - ➢ Only one head pointer per directory entry
    - ✧ Rest is all proportional to cache size
- ❖ But complex hardware implementation

Alternatives for Directories



Directory information co-located with memory module in home node for each memory block

Examples: Stanford DASH/FLASH, MIT Alewife, SGI Origin, HAL

Caches holding a copy of the memory block form a doubly linked list
Directory holds pointer to head of list

Examples:
IEEE SCI, Sequent NUMA-Q

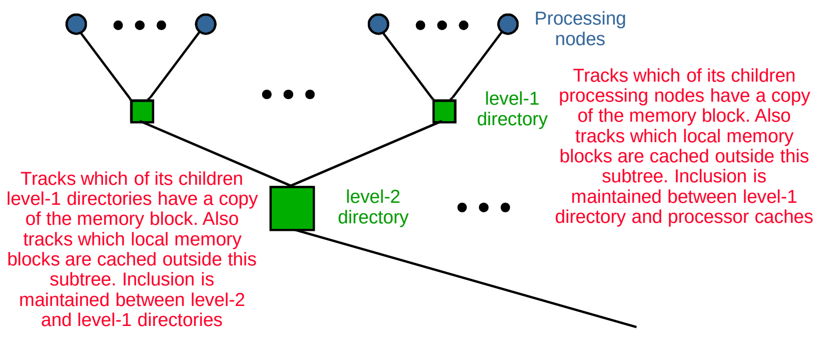
Finding Directory Information

- Flat schemes
  - Directory distributed with memory
  - Information at home node
  - Location based on address: transaction sent to home
- Hierarchical schemes
  - Directory organized as a hierarchical data structure
  - Leaves are processing nodes
  - Internal nodes have only directory information
    - Directory entry says whether subtree caches a block
  - To find directory info, send “search” message up to parent
    - Routes itself through directory lookups
  - Point-to-point messages between children and parents

Locating Shared Copies of a Block

- ❖ Flat Schemes
  - ➢ Memory-based schemes
    - ✧ Information about all copies stored at the home directory
    - ✧ Examples: Stanford Dash/Flash, MIT Alewife , SGI Origin
  - ➢ Cache-based schemes
    - ✧ Information about copies distributed among copies themselves
      - ▪ Inside caches, each copy points to next to form a linked list
    - ✧ Scalable Coherent Interface (SCI: IEEE standard)
- ❖ Hierarchical Schemes
  - ➢ Through the directory hierarchy
  - ➢ Each directory entry has presence bits
    - ✧ To parent and children subtrees

Hierarchical Directories

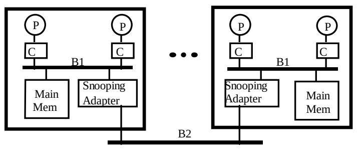

Processing nodes
level-1 directory
Tracks which of its children processing nodes have a copy of the memory block. Also tracks which local memory blocks are cached outside this subtree. Inclusion is maintained between level-1 directory and processor caches
level-2 directory
Tracks which of its children level-1 directories have a copy of the memory block. Also tracks which local memory blocks are cached outside this subtree. Inclusion is maintained between level-2 and level-1 directories

- ❖ Directory is a hierarchical data structure
  - ➢ Leaves are processing nodes, internal nodes are just directories
  - ➢ Logical hierarchy, not necessarily a physical one
    - ✧ Can be embedded in a general network topology

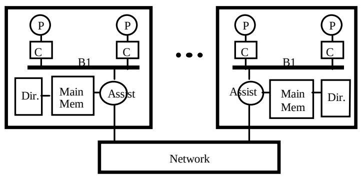
Two-Level Hierarchy

- ❖ Individual nodes are multiprocessors
  - ➢ Example: 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

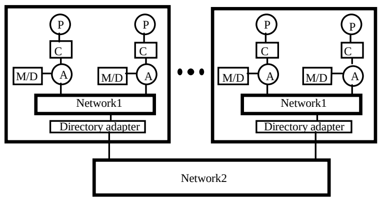
Examples of Two-level Hierarchies



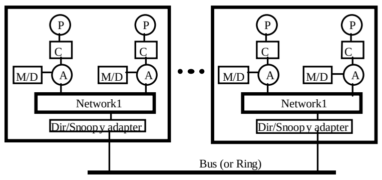
(a) Snooping-snooping



(b) Snooping-directory



(c) Directory-directory



(d) Directory-snooping

Hierarchical Approaches: Snooping

- Extend snooping approach: hierarchy of broadcast media
  - Tree of buses or rings
  - 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
- Handled similarly to bus, but not full broadcast
  - faulting processor sends out “search” bus transaction on its bus
  - propagates up and down hieararchy based on snoop results
- Problems:
  - High latency: multiple levels, and snoop/lookup at every level
  - Bandwidth bottleneck at root
- Not popular today

Advantages of Multiprocessor Nodes

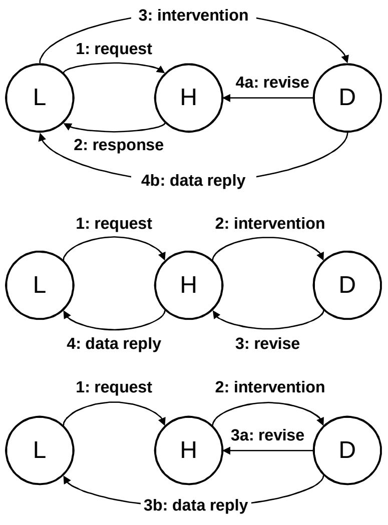
- Potential for cost and performance advantages
  - Amortization of node fixed costs over multiple processors
  - Can use commodity SMPs and CMPs
  - Less nodes for directory to keep track of
  - Much communication may be contained within node
  - Processors within a node prefetch data for each other
  - Processors can share caches (overlapping of working sets)
  - Benefits depend on sharing pattern (and mapping)
    - Good for widely read-shared data
    - Good for nearest-neighbor, if properly mapped
    - Not so good for all-to-all communication

Protocol Design Optimizations

- ❖ Reduce Bandwidth demands
  - ➢ By reducing number of protocol transactions per operation
- ❖ Reduce Latency
  - ➢ By reducing network transactions in critical path
  - ➢ Overlap network activities or make them faster
- ❖ Reduce endpoint assist occupancy per transaction
  - ➢ Especially when the assists are programmable
- ❖ Traffic, latency, and occupancy …
  - ➢ Should not scale up too quickly with number of nodes

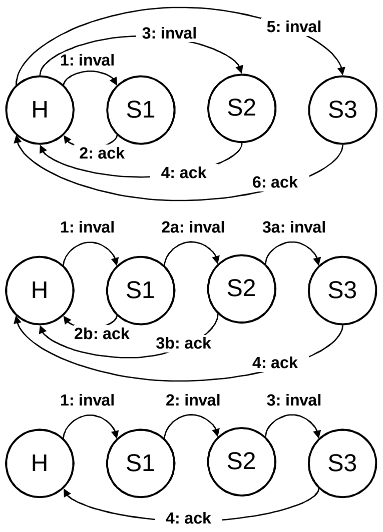
Reducing Read Miss Latency

- ❖ Strict Request-Response:
  - ➢ L: Local or Requesting node
  - ➢ H: Home node
  - ➢ D: Dirty node
  - ➢ 4 transactions in critical path
  - ➢ 5 transactions in all
- ❖ Intervention Forwarding:
  - ➢ Home forwards intervention
  - ➢ Directed to owner's cache
  - ➢ Home keeps track of requestor
  - ➢ 4 transactions in all
- ❖ Reply Forwarding:
  - ➢ Owner replies to requestor
  - ➢ 3 transactions in critical path



Reducing Invalidation Latency

- ❖ In Cache-Based Protocol
- ❖ Invalidations sent from Home
  - ➢ To all sharer nodes S_i
- ❖ Strict Request-Response:
  - ➢ 2s transactions in total
  - ➢ 2s transactions in critical path
  - ➢ s = number of sharers
- ❖ Invalidation Forwarding:
  - ➢ Each sharer forwards invalidation to next sharer
  - ➢ s acknowledgements to Home
  - ➢ s+1 transactions in critical path
- ❖ Single acknowledgement:
  - ➢ Sent by last sharer
  - ➢ s+1 transactions in total



Correctness

- ❖ Ensure basics of coherence at state transition level
  - ➢ Relevant lines are invalidated, updated, and retrieved
  - ➢ Correct state transitions and actions happen
- ❖ Ensure serialization and ordering constraints
  - ➢ Ensure serialization for coherence (on a single location)
  - ➢ Ensure atomicity for consistency (on multiple locations)
- ❖ Avoid deadlocks, livelocks, and starvation
- ❖ Problems:
  - ➢ Multiple copies and multiple paths through network
  - ➢ Large latency makes optimizations attractive
    - ✧ But optimizations complicate correctness

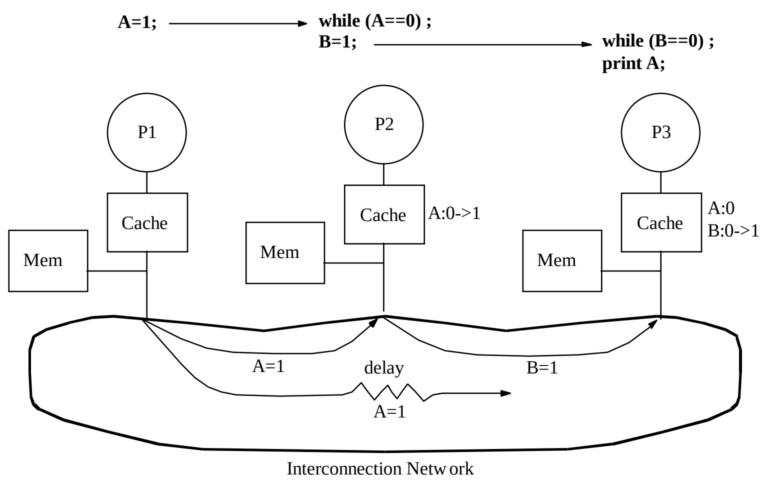
Serialization for Coherence

- ❖ Serialization means that writes to a given location
  - ➢ Are seen in the same order by all processors
- ❖ In a bus-based system:
  - ➢ Multiple copies, but write serialization is imposed by bus
- ❖ In a scalable multiprocessor with cache coherence
  - ➢ Home node can impose serialization on a given location
    - ✧ All relevant operations go to the home node first
  - ➢ But home node cannot satisfy all requests
    - ✧ Valid copy can be in a dirty node
    - ✧ Requests may reach the home node in one order, …
      - ▪ But reach the dirty node in a different order
    - ✧ Then writes to same location are not seen in same order by all

Ensuring Serialization

- ❖ Use additional 'busy' or 'pending' directory states
- ❖ Indicate that previous operation is in progress
  - ➢ Further operations on same location must be delayed
- ❖ Ensure serialization using one of the following …
  - ➢ Buffer at the home node
    - ✧ Until previous (in progress) request has completed
  - ➢ Buffer at the requestor nodes
    - ✧ By constructing a distributed list of pending requests
  - ➢ NACK and retry
    - ✧ Negative acknowledgement sent to the requestor
    - ✧ Request retried later by the requestor's assist
  - ➢ Forward to the dirty node
    - ✧ Dirty node determines their serialization when block is dirty

Write Atomicity Problem



Ensuring Write Atomicity

❖ Multiple copies and a distributed interconnect

❖ In Invalidation-based scheme

- ➢ Block can be shared by multiple nodes
- ➢ Owner provides appearance of write atomicity by not allowing read access to the new value until all shared copies are invalidated and invalidations are acknowledged.

❖ Much harder in update schemes!

- ➢ Since the data is sent directly to the sharers and is accessible immediately

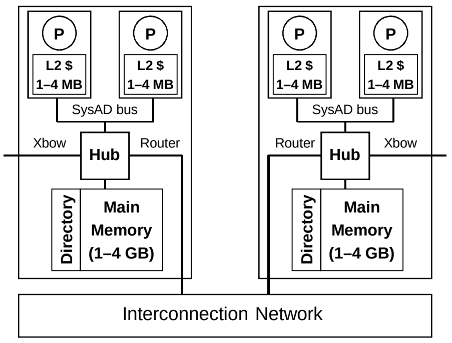
Deadlock, Livelock, & Starvation

- ❖ Potential source for deadlock
  - ➢ A node may receive too many messages
  - ➢ If no sufficient buffer space then flow control causes deadlock
- ❖ Possible solutions for buffer deadlock
  - ➢ Provide enough buffer space or use main memory
  - ➢ Use NACKs (negative acknowledgments)
  - ➢ Provide separate request and response networks with separate buffers
- ❖ Potential source for Livelock
  - ➢ NACKs that cause potential livelock and traffic problems
- ❖ Potential source for Starvation
  - ➢ Unfairness in handling requests
  - ➢ Solution is to buffer all requests in FIFO order

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 distributed linked list
- ❖ Issue (c): communicating with those copies
  - ➢ Memory-based: point-to-point messages (coarser on overflow)
    - ✧ Can be multicast or overlapped
  - ➢ Cache-based: point-to-point linked list traversal to find them
- ❖ 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

SGI Origin Case Study



❖ Each node consists of …

- ➢ Two MIPS R10k processors with 1st and 2nd level caches
- ➢ A fraction of the total main memory and directory
- ➢ A Hub: communication assist or coherence controller

SGI Origin System Overview

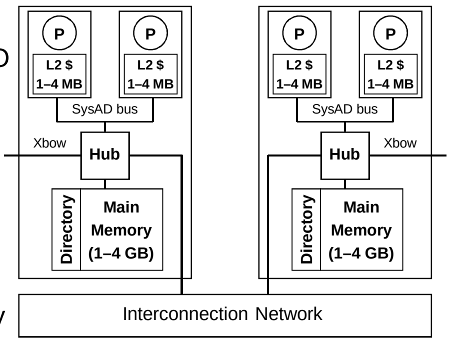
❖ Directory is in DRAM and is accessed in parallel

❖ Hub is the communication assist

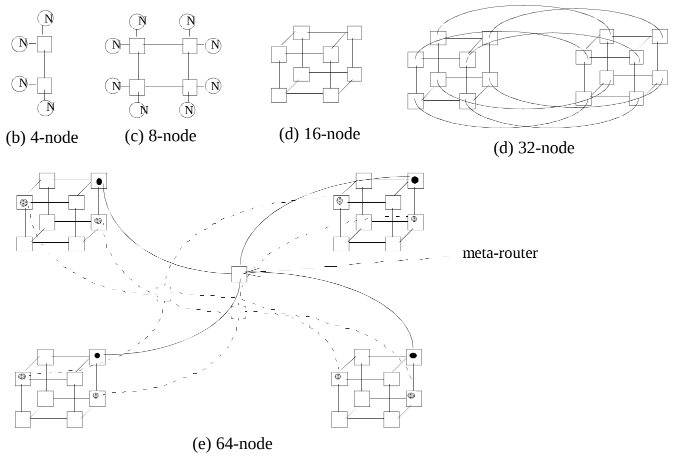
- ➢ Implements the network interface and coherence protocol
- ➢ Sees all second-level cache misses
- ➢ Connects to router chip
- ➢ Connects to Xbow for I/O
- ➢ Connects to memory

❖ SysAD Bus

- ➢ System Address/Data
- ➢ Shared by 2 processors
- ➢ No snooping coherence
- ➢ Coherence thru directory



Origin Network



Up to 512 nodes or 1024 processors

Origin Network – cont'd

- ❖ Interconnection network has a hypercube topology
  - ➢ With up to 32 nodes or 64 processors
- ❖ Fat cube topology for beyond 64 processors
  - ➢ Up to 512 nodes or 1024 processors
- ❖ Each router has six pairs of 1.56GB/s 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 per physical link
  - ➢ Request, reply, other two for priority or I/O

Origin Cache and Directory States

- ❖ Cache states: MESI
- ❖ Seven directory states
  - ➢ Unowned: no cache has a copy, memory copy is valid
  - ➢ Exclusive: one cache has block in M or E state
    - ✧ Memory may or may not be up to date
  - ➢ Shared: multiple (can be zero) caches have a shared copy
    - ✧ Shared copy is clean and memory has a valid copy
  - ➢ Three Busy states for three types of pending requests:
    - ✧ Read, read exclusive (or upgrade), and uncached read
    - ✧ Indicates home 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

Origin Directory Structure

- ❖ Flat Memory based: directory information at home
- ❖ Three directory formats:

  1) If exclusive, entry is pointer to specific processor (not node)

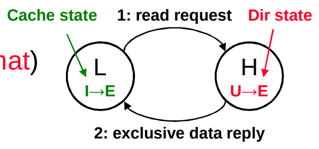
  2) If shared, bit vector: each bit points to a node, not processor
    - ✧ Invalidation sent to a Hub is broadcast to both processors
    - ✧ 16-bit format (32 processors), kept in main memory DRAM
    - ✧ 64-bit format (128 processors), uses extended directory memory
    - ✧ 128-byte blocks

  3) For larger machines, coarse bit vector
    - ✧ Each bit corresponds to n/64 nodes (64-bit format)
    - ✧ Invalidations are sent to all Hubs and processors in a group

  - ➢ Machine chooses format dynamically
    - ✧ When application is confined to 64 nodes or less

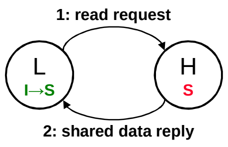
Handling a Read Miss

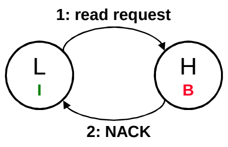
- ❖ Hub examines address
  - ➢ If remote, sends request to home node
  - ➢ If local, looks up local directory entry and memory
  - ➢ Memory block is fetched in parallel with directory entry
    - ✧ Called speculative data access
    - ✧ Directory lookup returns one cycle earlier
    - ✧ If directory is unowned or shared, it's a win
      - ▪ Data already obtained by Hub
  - ➢ Directory may indicate one of many states
- ❖ If directory is in Unowned State
  - ➢ Goes to Exclusive state (pointer format)
  - ➢ Replies with data block to requestor
    - ✧ Exclusive data reply
    - ✧ No network transactions if home is local



Read Block in Shared / Busy State

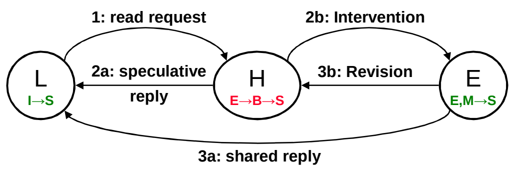
- ❖ If directory is in Shared state:
  - ➢ Directory sets presence bit of requesting node
  - ➢ Replies with data block to requestor
    - ✧ Shared data reply
    - ✧ Strict request-reply
    - ✧ Block is cached in shared state
- ❖ If directory is in a Busy state:
  - ➢ Directory not ready to handle request
  - ➢ NACK to requestor
  - ➢ Asking requestor to try again
  - ➢ So as not to buffer request at home
  - ➢ NACK is a response that does not carry data





Read Block in Exclusive State

- ❖ If directory is in Exclusive state (interesting case):
  - ➢ If home is not owner, need to get data from owner
    - ✧ To home and to requestor
  - ➢ Uses reply forwarding for lowest latency and traffic
  - ➢ Directory state is set to busy-shared state
  - ➢ Home optimistically sends a speculative reply to requestor
  - ➢ At same time, request is forwarded to exclusive node (owner)
  - ➢ Owner sends revision message to home & reply to requestor



Read Block in Exclusive State (2)

- ❖ At the home node:
  - ➢ Set directory to busy state to NACK subsequent requests
  - ➢ Set requestor presence bit
  - ➢ Assume block is clean and send speculative reply
- ❖ At the owner node:
  - ➢ If block is dirty
    - ✧ Send data reply to requestor and a sharing write back to home
    - ✧ Both messages carry modified block data
  - ➢ If block is clean
    - ✧ Similar to above, but messages don’t carry data
    - ✧ Revision message to home is called a downgrade
- ❖ Home changes state from busy-shared to shared
  - ➢ When it receives revision message

Speculative Replies

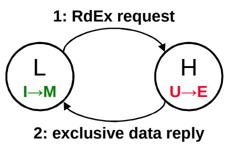
- ❖ Why speculative replies?
  - ➢ Requestor needs to wait for reply from owner anyway
  - ➢ No latency savings
  - ➢ Could just get data from owner always
- ❖ Two reasons when block is in exclusive-clean state
  1. L2 cache controller does not reply with data
     - ✧ When data is in exclusive-clean state in cache
     - ✧ So there is a need to get data from home (speculative reply)
  2. Enables protocol optimization
     - ✧ No need to send notification messages back to home
       - ▪ When exclusive-clean blocks are replaced in caches
     - ✧ Home will supply data (speculatively) and then checks

Handling a Write Miss

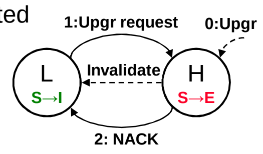
- ❖ Request to home can be read exclusive or upgrade
  - ➢ Read-exclusive when block is not present in cache
    - ✧ Request both data and ownership
  - ➢ Upgrade when writing a block in shared state
    - ✧ Request ownership only
  - ➢ Copies in other caches must be invalidated
    - ✧ Except when directory state in Unowned
  - ➢ Invalidations must be explicitly acknowledged
  - ➢ Home node updates directory and sends the invalidations
  - ➢ Includes the requestor identity in the invalidations
    - ✧ So that acknowledgements are sent back to requestor directly

Write to Block in Unowned State

- ❖ If Read Exclusive:
  - ➢ Directory state is changed to Exclusive
  - ➢ Directory entry is set to point to requestor
  - ➢ Home node sends a data reply to requestor
  - ➢ Requestor cache changes state to Modified



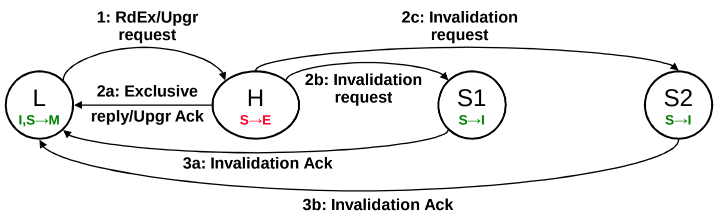
- ❖ if Upgrade:
  - ➢ The expected directory state is Shared
  - ➢ But Exclusive means block has been invalidated
    - ✧ In requestor's cache and directory notified
    - ✧ Can happen in Origin Protocol
      - ▪ Someone else has upgraded same block



  - ➢ NACK reply: request should be retried as Read Exclusive

Write to Block in Shared State

- ❖ At the home:
  - ➢ Set directory state to exclusive
  - ➢ Set directory pointer for requestor processor
    - ✧ Ensures that subsequent requests are forwarded to requestor
- ❖ Home sends invalidation requests to all sharers
  - ➢ Which will acknowledge requestor (Not home)

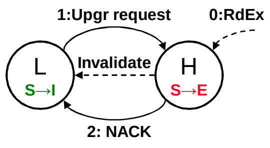


Write to Block in Shared State (2)

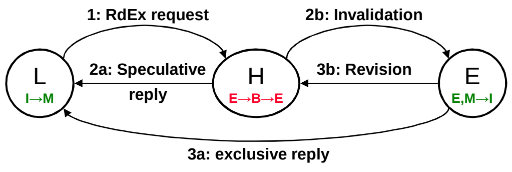
- ❖ If Read Exclusive, home sends
  - ➢ Exclusive reply with invalidations pending to requestor
    - ✧ Reply contains data and number of sharers to be invalidated
- ❖ If Upgrade, home sends
  - ➢ Upgrade Ack (No data) with invalidations pending
- ❖ At requestor:
  - ➢ Wait for all acks to come back before "closing" the operation
  - ➢ Subsequent request for block to home
    - ✧ Forwarded as intervention to requestor
  - ➢ For proper serialization
    - ✧ Requestor does not handle intervention request until
      - ▪ All invalidation acks received for its outstanding request

Write to Block in Exclusive State

- ❖ If Upgrade
  - ➢ Expected directory state is Shared
  - ➢ But another write has beaten this one
  - ➢ Request not valid so NACK response
  - ➢ Request should be retried as Read Exclusive



- ❖ If Read Exclusive
  - ➢ Set dir state to busy-exclusive and send speculative reply
  - ➢ Send invalidation to owner with identity of requestor

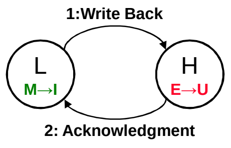


Write Block in Exclusive State (2)

- ❖ At owner:
  - ➢ Send ownership transfer revision message to home
    - ✧ No data is sent to home
  - ➢ If block is dirty in cache
    - ✧ Send exclusive reply with data to requestor
    - ✧ Overrides speculative reply sent by home
  - ➢ If block is clean in exclusive state
    - ✧ Send acknowledgment to requestor
    - ✧ No data to requestor because got that from speculative reply
- ❖ Write to a block in Busy state: NACK
  - ➢ Must try again to avoid buffering request at home node

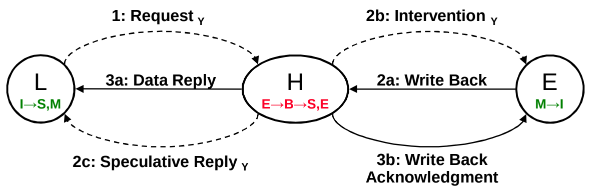
Handling Writeback Requests

- ❖ Directory state cannot be shared or unowned
  - ➢ Requestor is the owner and has the block as modified
- ❖ If a read request had come in to set state to shared
  - ➢ Request would have been forwarded to owner, and
  - ➢ State would have been Busy
- ❖ If directory state is Exclusive
  - ➢ Directory state is set to unowned
  - ➢ Acknowledgment is returned



Handling Writeback Requests (2)

- ❖ If directory state is Busy: interesting race condition
  - ➢ Busy because of request from another node Y
  - ➢ Intervention has been forwarded to node X doing writeback
    - ✧ Intervention and writeback have crossed each other
  - ➢ Y's operation has had it's effect on directory
  - ➢ Cannot drop or NACK writeback (only valid copy)
  - ➢ Can't even retry after Y's ref completes



Solution to Writeback Race

- ❖ Combine the two operations
  - ➢ Request coming from node Y and Writeback coming from X
- ❖ When writeback reaches directory, state changes to
  - ➢ Shared if it was busy-shared (Y requested read copy)
  - ➢ Exclusive if busy-exclusive (Y requested exclusive copy)
- ❖ Home forwards the writeback data to the requestor Y
  - ➢ Sends writeback acknowledgment to X
- ❖ When X receives the intervention, it ignores it
  - ➢ Since it has an outstanding writeback for the cache block
- ❖ Y's operation completes when it gets the reply
- ❖ X's writeback completes when it gets writeback ack

Replacement of Shared Block

- ❖ Could send a replacement hint to the directory
  - ➢ To clear the presence bit of the sharing node
  - ➢ Reduces occurrence in limited-pointer representation
- ❖ Can eliminate an invalidation
  - ➢ Next time the block is written
- ❖ But does not reduce traffic
  - ➢ Have to send a replacement hint
  - ➢ Incurs the traffic at a different time
  - ➢ If block is not written again then replacement hint is a waste
- ❖ Origin protocol does not use replacement hints
  - ➢ Can have a directory in shared state while no copy exists

Total Transaction Types

- ❖ Coherent memory transactions
  - ➢ 9 request types
  - ➢ 6 invalidation/intervention types
  - ➢ 39 reply/response types
- ❖ Non-coherent memory transactions
  - ➢ Uncached memory, I/O, and synchronization operations
  - ➢ 19 request transaction types
  - ➢ 14 reply types
  - ➢ NO invalidation/intervention types

Serialization for Coherence

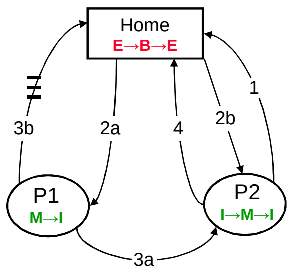
- ❖ Serialization means that writes to a given location
  - ➢ Are seen in the same order by all processors
- ❖ In a bus-based system:
  - ➢ Multiple copies, but write serialization is imposed by bus
- ❖ In a scalable multiprocessor with cache coherence
  - ➢ Home node can impose serialization on a given location
    - ✧ All relevant operations go to the home node first
  - ➢ But home node cannot satisfy all requests
    - ✧ Valid copy may not be in main memory but in a dirty node
    - ✧ Requests may reach the home node in one order, …
      - ▪ But reach the dirty node in a different order
    - ✧ Then writes to same location are not seen in same order by all

Possible Solutions for Serialization

- ❖ Buffer (FIFO) requests at the home (MIT Alewife)
  - ➢ All requests go to home first (good for serialization)
  - ➢ Buffer until previous request has completed
    - ✧ But buffering requests becomes acute at the home (full buffer)
    - ✧ Let buffer overflow into main memory (MIT Alewife)
- ❖ Buffer at the requestors (SCI Protocol)
  - ➢ Distributed linked list of pending requests
  - ➢ Used in cache-based approach
- ❖ Use busy state and NACK to retry (Origin2000)
  - ➢ Negative acknowledgement sent to requestor
  - ➢ Serialization order is for accepted requests (not NACKed)
- ❖ Forward to owner (Stanford DASH)
  - ➢ Serialization determined by home when clean
  - ➢ Serialization determined by owner when exclusive

Need Serialization at Home

❖ Example shows need for serialization at home



Initial Condition: block A is in modified state in P1's cache.

1. P2 sends read-exclusive request for A to home node.
2a. Home forwards request to P1 (dirty node).
2b. Home sends speculative reply to P2.
3a. P1 sends data reply to P2 (replaces 2b).
3b. P1 sends "Ownership transfer" revision message to Home.
4. P2 having received its reply, considers write complete. Proceeds, but incurs a replacement of the just dirtied block, causing it to be written back in transaction 4.

Writeback transaction is received at home before "Ownership-transfer" message (3b) and the block is written into memory. Then when the revision message arrives at the home, the directory is made to point to P2 as having the dirty copy. This corrupts coherence.

❖ Origin prevents this from happening using busy states

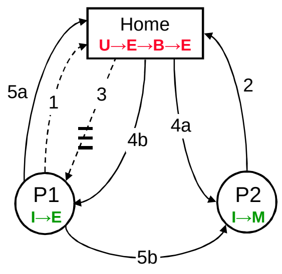
➢ Directory detects write-back from same node as request

➢ Write-back is NACKed and must be retried

Need Serialization at Requestor

❖ Having home node determine order is not enough

➢ Following example shows need for serialization at requestor



1. P1 sends read request to home node for block A
2. P2 sends read-exclusive request to home for the write of A. But won't process it until it is done with read.
3. In response to (1), home sends reply to P1 (and sets directory pointer). Home now thinks read is complete. Unfortunately, the reply does not get to P1 right away.
4a. In response to (2), home sends speculative reply to P2
4b. In response to (2), home sends invalidation to P1; it reaches P1 before (3) (no point-to-point order).
5. P1 receives and applies invalidate, sends acknowledgment 5a to home and 5b to requestor.

Finally, transaction 3 (exclusive data reply) reaches P1. P1 has block in exclusive state, while P2 has it modified.

❖ Origin: serialization is applied to all nodes

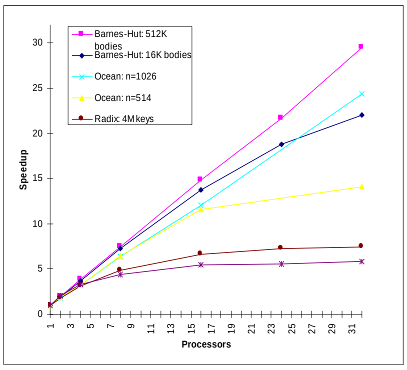
➢ Any node does not begin a new transaction …

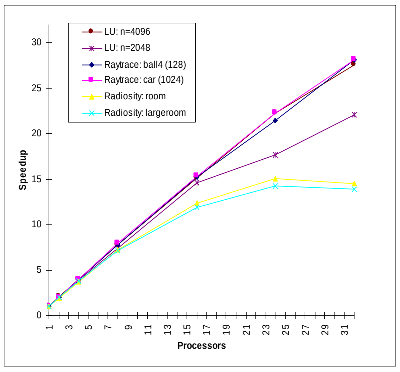
✧ Until previous transaction on same block is complete

Starvation

- ❖ NACKs can cause starvation
- ❖ Possible solutions:
  - ➢ Do nothing: starvation shouldn't happen often (DASH)
  - ➢ Random delay between retries
  - ➢ Priorities: increase priority of NACKed requests (Origin)

Application Speedups





Summary

- In directory protocol
  - Substantial implementation complexity below state diagram
  - Directory versus cache states
  - Transient states
  - Race conditions
  - Conditional actions
  - Speculation
- Origin philosophy:
  - Memory-less: a node reacts to incoming events using only local state
  - An operation does not hold shared resources while requesting others