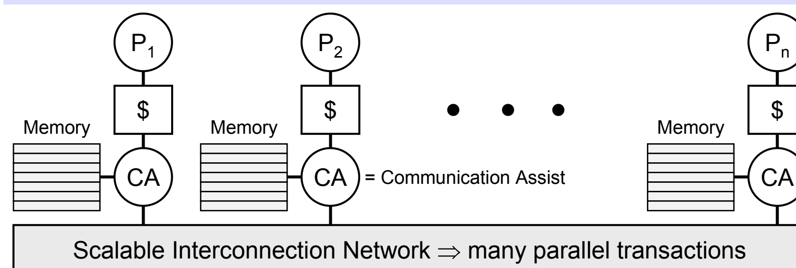
Cache Coherence in Scalable Machines

CSE 661 – Parallel and Vector Architectures

Prof. Muhamed Mudawar
Computer Engineering Department
King Fahd University of Petroleum and Minerals

Generic Scalable Multiprocessor



- 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

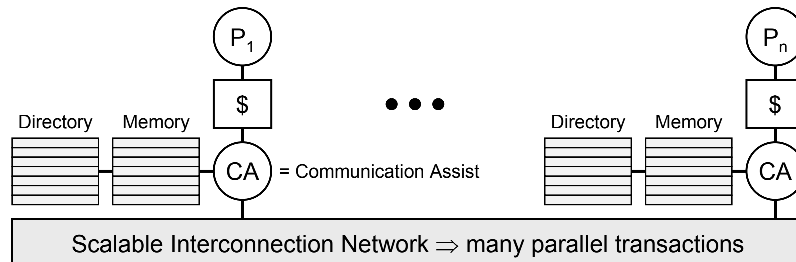
What Must a Coherent System do?

- 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

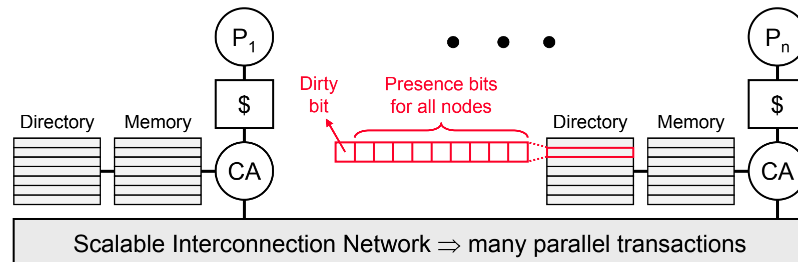


- Scalable cache coherence is based on directories
- Distributed directories for distributed memories
- Each cache-line-sized block of a memory …
  - Has a directory entry that keeps track of …
    - State and the nodes that are currently sharing the block

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



- 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

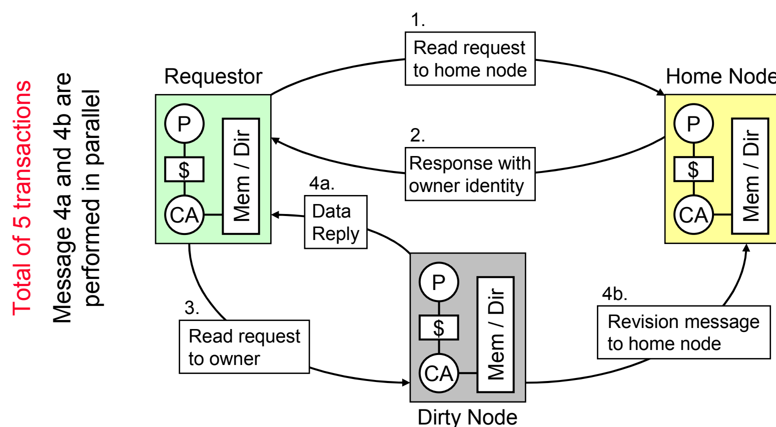
- Home Node
  - Node in whose main memory the block is allocated
- Dirty Node
  - Node that has copy of block in its cache in modified state
- Owner Node
  - Node that currently holds the valid copy of a block
  - Can be either the home node or dirty node
- Exclusive Node
  - Node that has block in its 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

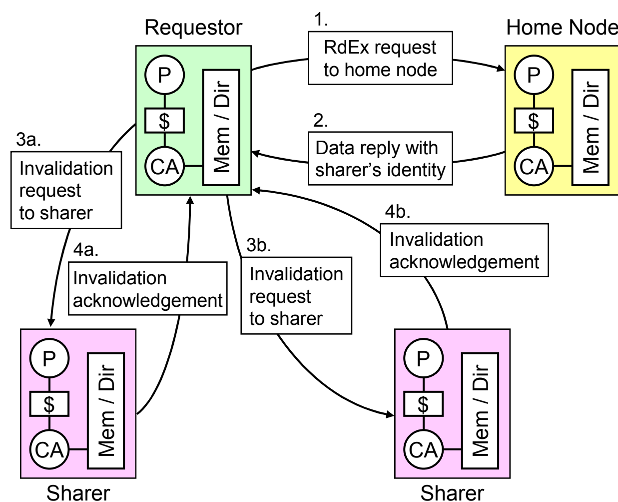


Basic Operation on a Write Miss

- 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



Data Sharing Patterns

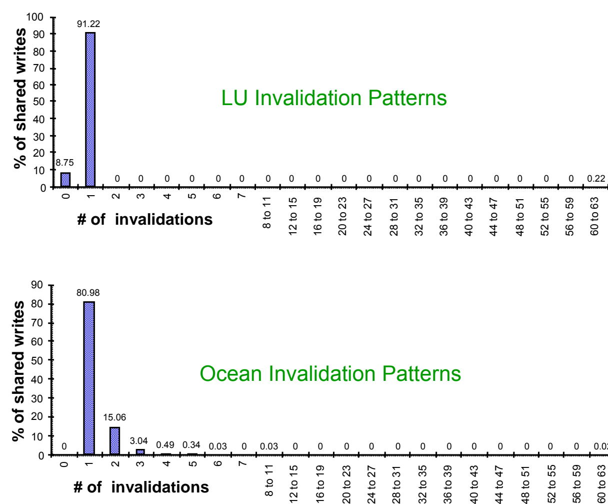
- Provide insight into directory requirements
- If most misses involve $O(P)$ transactions then
  - Broadcast might be a good solution
- However generally, there are few sharers at a write
- Important to understand two aspects of data sharing
  - Frequency of shared writes or invalidating writes
    - 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

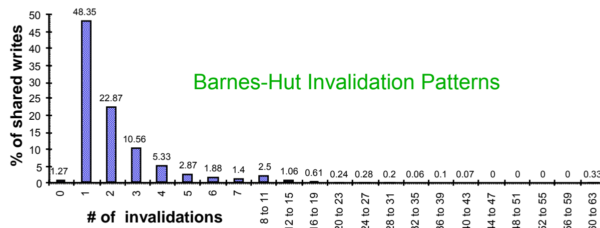
MSI protocol is used

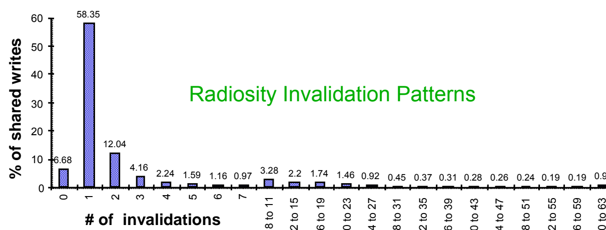


Cache Invalidation Patterns – cont'd

Infinite per-processor caches are used

To capture inherent sharing patterns





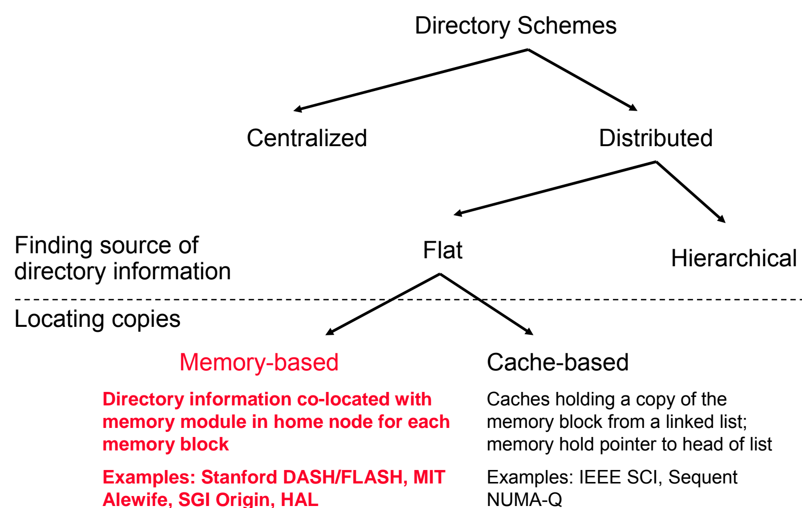
Framework for Sharing Patterns

- 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 from a linked list; memory hold pointer to head of list

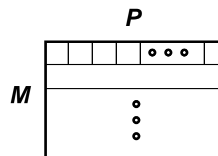
Examples: IEEE SCI, Sequent NUMA-Q

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 Storage Overhead

- ❖ 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

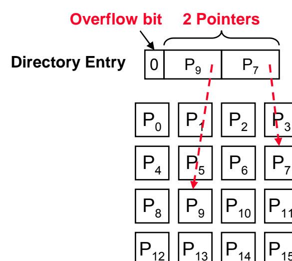


Directory Storage Reductions

- ❖ 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

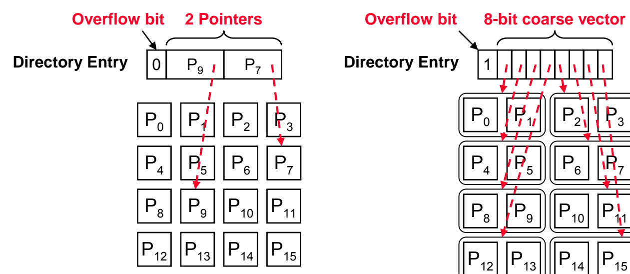


Overflow Methods

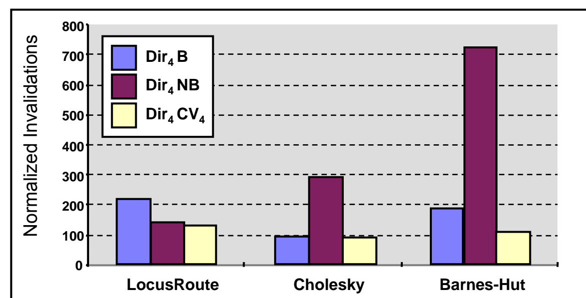
- Broadcast (Dir_iB)
  - Broadcast bit turned on upon overflow
  - Invalidations broadcast to all nodes when block is written
    - Regardless of whether or not they were sharing the block
  - Network bandwidth may be wasted for overflow cases
  - Latency increases if processor wait for acknowledgements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

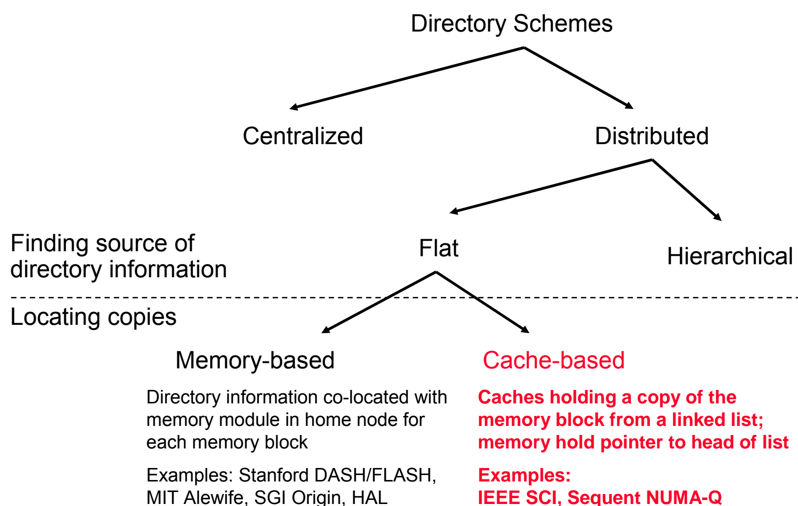
Software Overflow Schemes

- Software (Dir_iSW)
  - 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_iDP) is a variation of Dir_iSW
  - 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

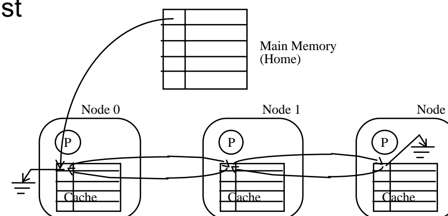
- ❖ Sparse Directory: reducing M term in P × M
- ❖ Observation: cached entries << main memory
  - ✸ Most directory entries are unused most of the time
  - ✸ Example: 2MB cache and 512MB local memory per node
    - => 510 / 512 or 99.6% of directory entries are unused
- ❖ Organize directory as a cache 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



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
  - On write, propagate chain of invalidations down the list
- Scalable Coherent Interface (SCI) IEEE Standard
  - Doubly linked list



Scaling Properties (Cache-based)

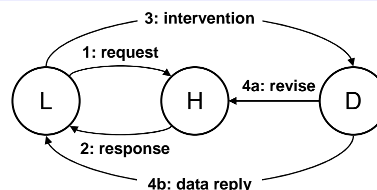
- Traffic on write: proportional to number of sharers
- Latency on write: proportional to number of sharers
  - Don't know identity of next sharer until reach current one
  - Also assist processing at each node along the way
  - Even reads involve more than one assist
    - Home and first sharer on list
- Storage overhead
  - Quite good scaling along both axes
  - Only one head pointer per memory block
    - Rest is all proportional to cache size
- But complex hardware implementation

Protocol Design Optimizations

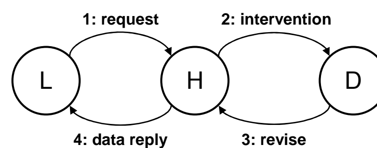
- Reduce Bandwidth demands
  - By reducing number of protocol transactions per operation
- Reduce Latency
  - By reducing network transactions in critical path
  - Overlap network activities or make them faster
- Reduce endpoint assist occupancy per transaction
  - Especially when the assists are programmable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

Reducing Read Miss Latency

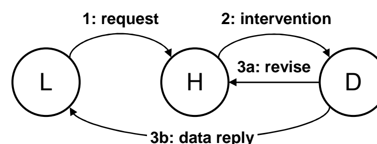
- Strict Request-Response:
  - L: Local or Requesting node
  - H: Home node
  - D: Dirty node
  - 4 transactions in critical path
  - 5 transactions in all



- Intervention Forwarding:
  - Home forwards intervention
  - Directed to owner's cache
  - Home keeps track of requestor
  - 4 transactions in all

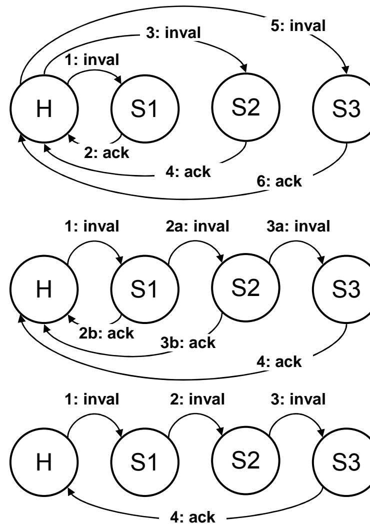


- Reply Forwarding:
  - Owner replies to requestor
  - 3 transactions in critical path



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 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

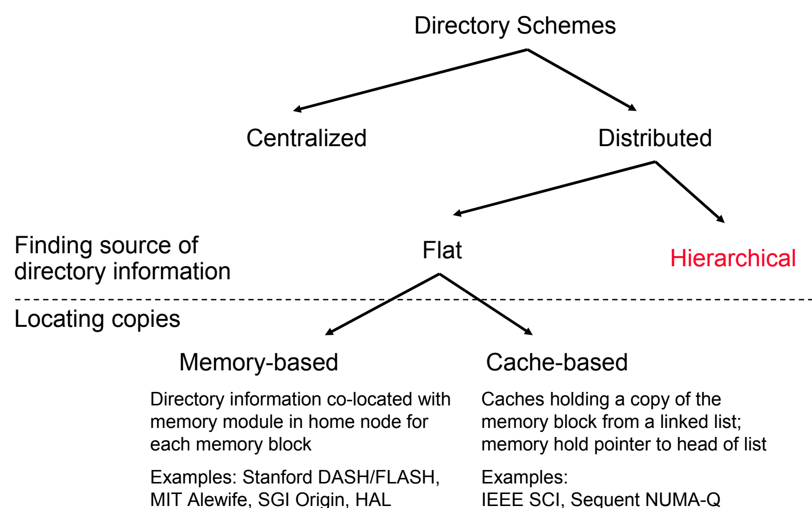
Ensuring Serialization

- 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

Sequential Consistency

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

Alternatives for Directories



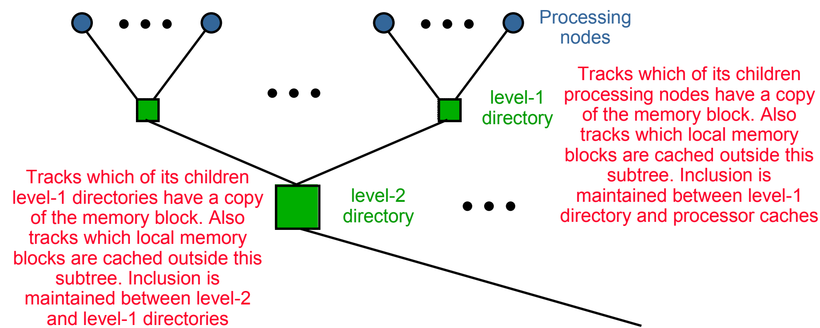
Finding Directory Information

- 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 copies stored all at the home with the block
    - 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 has presence bits
    - To parent and children subtrees

Hierarchical Directories

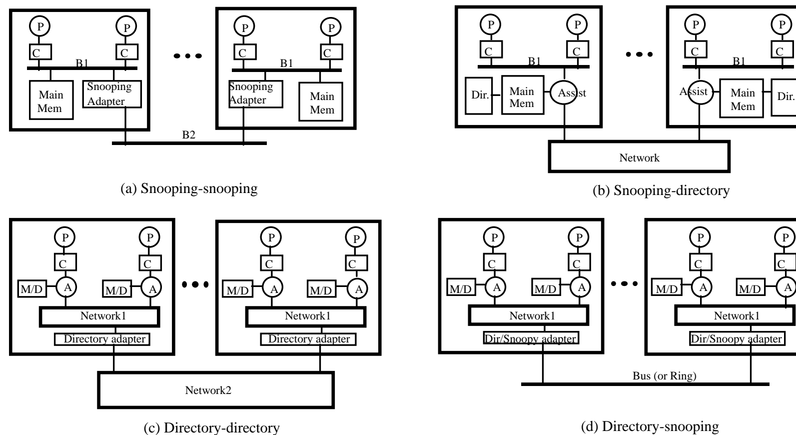


- ❖ Directory is a hierarchical data structure
  - ✸ Leaves are processing nodes, internal nodes are just directories
  - ✸ Logical hierarchy, not necessarily a physical one
    - ✧ Can be embedded in a general network topology

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

Example Two-level Hierarchies



Advantages of MP Nodes

- Potential for cost and performance advantages
  - Amortization of node fixed costs over multiple processors
  - Can use commodity SMPs
  - Less nodes for directory to keep track of
  - Much communication may be contained within node
  - 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 MP Nodes

- ❖ Bandwidth shared among nodes
- ❖ 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
- ❖ May hurt performance if sharing patterns don't comply