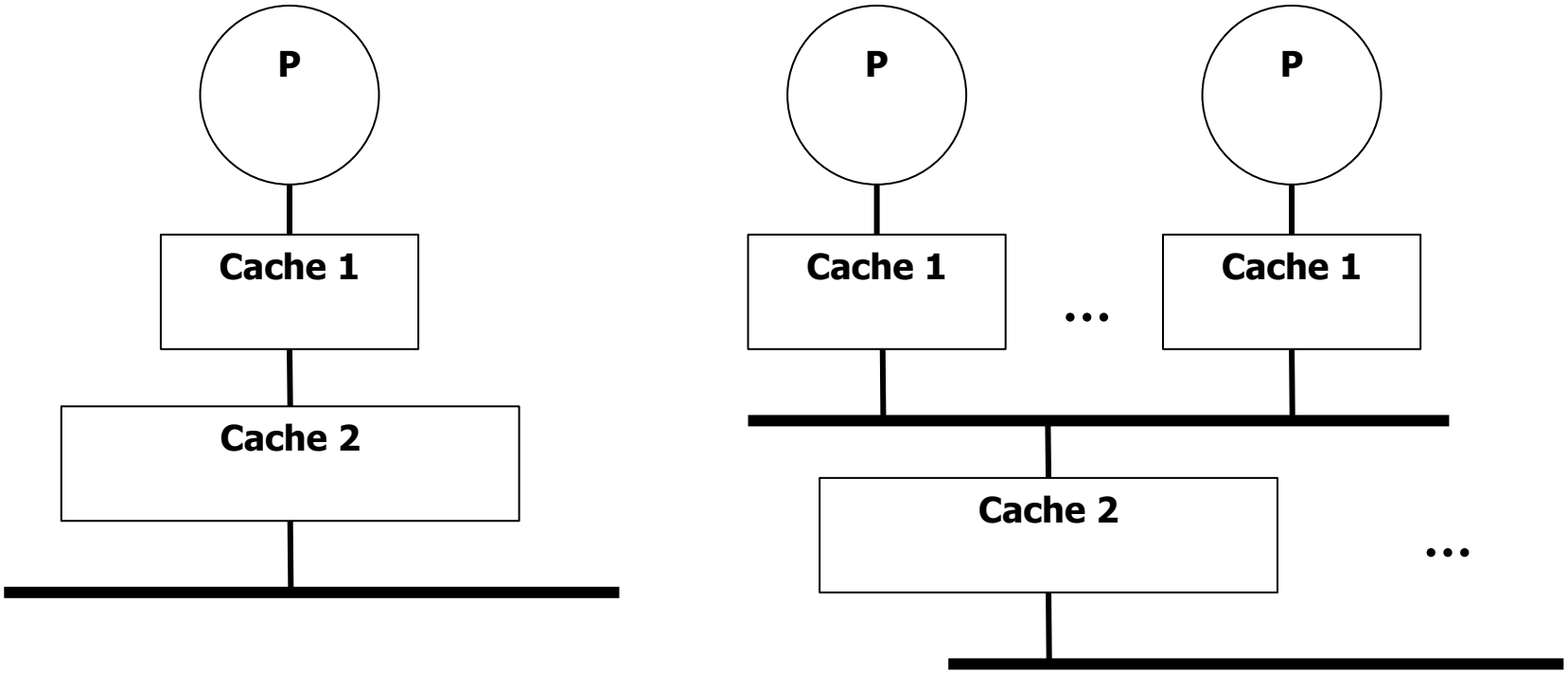
Scalable Cache Coherence

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu

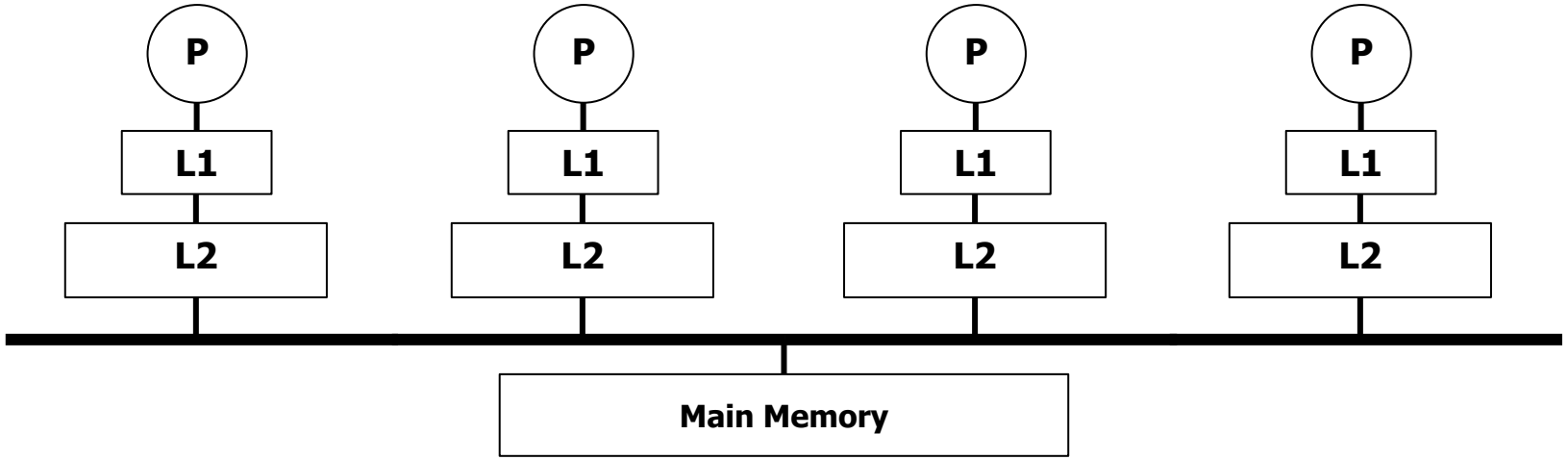
Hierarchical Cache Coherence

- Hierarchies in cache organization
  - Multiple levels of caches on a processor
  - Large scale multiprocessors with hierarchy of buses



Multilevel Caches

- Inclusion Property
  - Everything in L1 cache is also present in L2 cache
  - If L1 has the block in owned state (e.g. modified in MESI), L2 must have it in modified state
  - Snoop of L2 takes responsibility for flushing or invalidating data due to remote requests
  - It often helps if the block size in L1 is smaller or the same size as that in L2 cache



Maintaining Inclusion

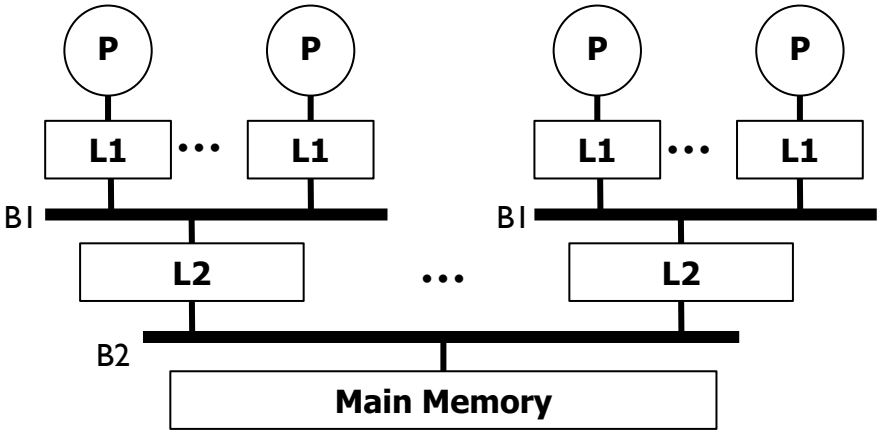
- In L2 cache
  - On cacheline replacement in L2 cache
    - Notify L1 cache of the address of victim cacheline to make L1 cache invalidate it
  - **Inclusion bit**
    - Set when a cacheline is also present in L1 cache
    - Filter interventions by cache-coherence transactions to L1 cache
  - On processor write (BusRDX)
    - Write-through L1 cache
      - Processor consumes substantial fraction of L2 cache bandwidth
    - Write-back L1 cache
      - Set corresponding cacheline in Modified-but-stale state (dirty + invalid)
      - Writes are absorbed in L1 cache

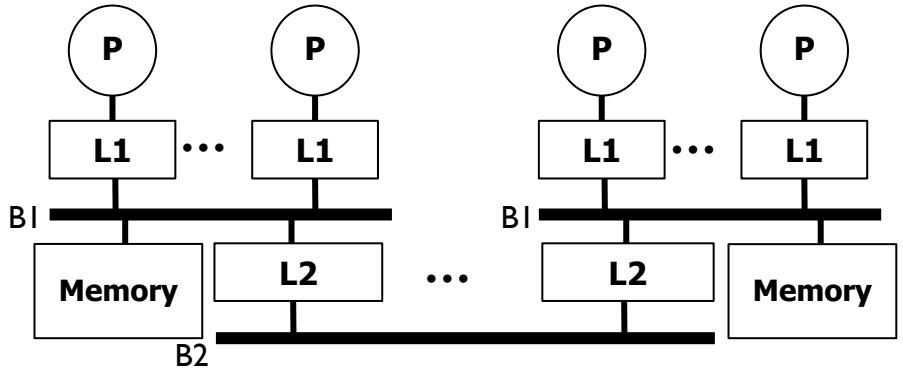
Propagating Coherence Transactions

- Requests from processor
  - Percolate them downward until requested block is found or request reaches bus
  - Read request
    - All caches → shared or exclusive state
  - Read-exclusive request
    - Innermost (L1) cache → modified state
    - Other caches → modified but stale state
- Transactions from bus
  - Percolate them upward to innermost cache
  - Invalidation request (BusRDX)
    - Copy-back corresponding cacheline to bus and invalidate it
  - Flush request (BusRD)
    - Copy-back corresponding cacheline to bus and change it into shared state

Hierarchical Snoopy Cache Coherence

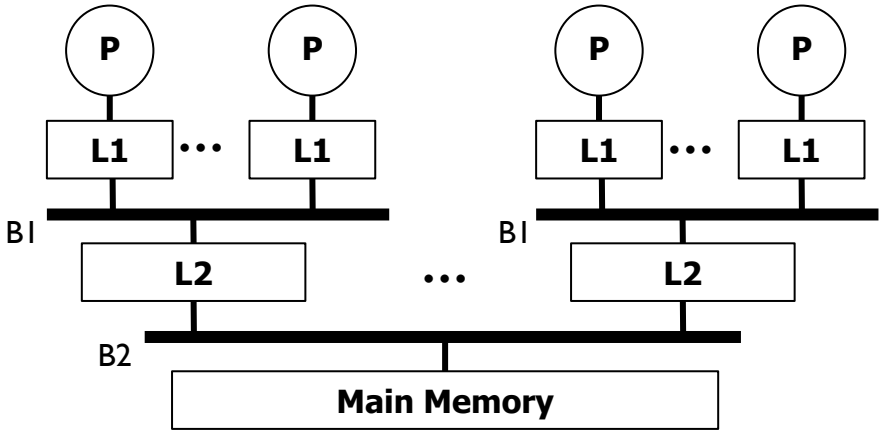
- Hierarchy of buses
  - Simplest way to build large-scale cache-coherent MPs
  - Use snoopy coherence at each level
- Memory location – two alternatives
  - Main memory centralized at the global (B2) bus
  - Main memory distributed among the clusters
    - L2 may not include local data in L1, but need to snoop for local data





Hierarchies with Global Memory

- First-level caches
  - Highest performance SRAM caches
  - B1 follows standard snoopy protocol
- Second-level caches
  - Much larger than L1 caches (set associative)
    - Must maintain inclusion property
  - L2 cache acts as filter for B1-bus and L1-caches
  - L2 cache can be DRAM based, since fewer references reach here



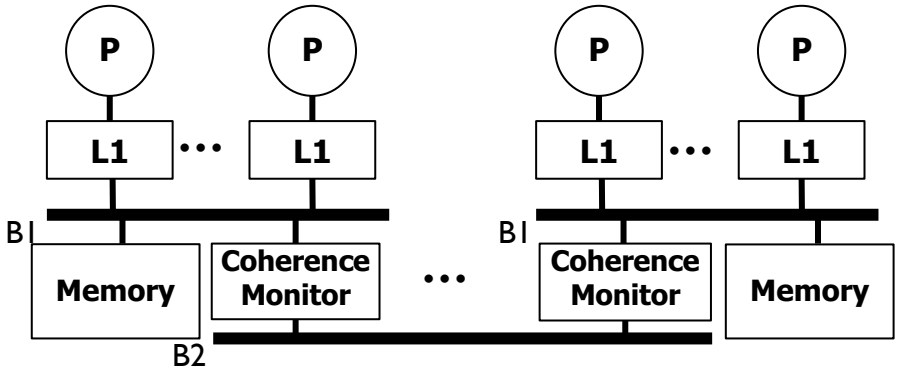
Hierarchies with Global Memory (cont'd)

- Advantages
  - Misses just require single traversal to the root of the hierarchy (global memory)
  - Placement of shared data is not an issue

- Disadvantages
  - Misses to local data structures (e.g., stack) also have to traverse the hierarchy, resulting in larger traffic and higher latency
  - Memory at the global bus must be highly interleaved.
    - Otherwise, bandwidth to the global memory will not scale

Hierarchies with Distributed Memory

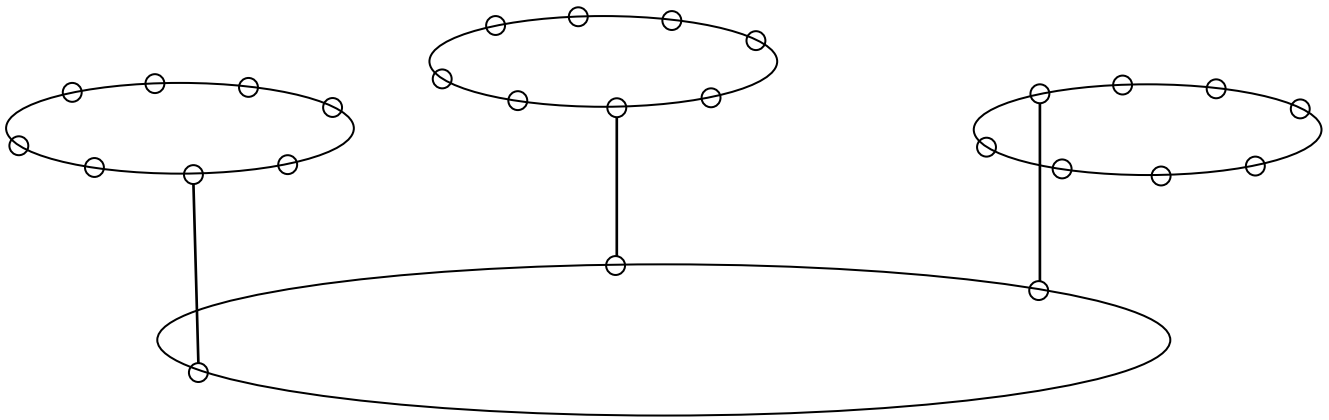
- Main memory is distributed among clusters
  - Reduces global bus traffic
    - Local data and suitably placed shared data
  - Reduce latency
    - Less contention and local accesses are faster
- Coherence monitor
  - Holds data allocated remotely but cached in the local node and its state → remote access cache
  - Holds state of data allocated locally but cached remotely → local state monitor
  - Snoops B1 and B2, and forwards transactions if necessary



Alternative: Hierarchy of Rings

- Hierarchical ring network, not bus
  - U of Toronto: Hector, NUMAchine, Kendall Square Research (KSR)
  - Snoop on requests passing by on ring

- Point-to-point structure of ring
  - Potentially higher bandwidth than buses
  - Still, high latency



Hierarchies

- Advantages
  - Conceptually simple to build
    - Apply snooping recursively
  - Merging and combining of requests in hardware

- Disadvantages
  - Physical hierarchies do not provide enough bandwidth
    - The root becomes a bottleneck
    - Patch solution: multiple buses/rings at higher levels
  - Latencies often larger than those in direct networks

Directory-Based Cache Coherence

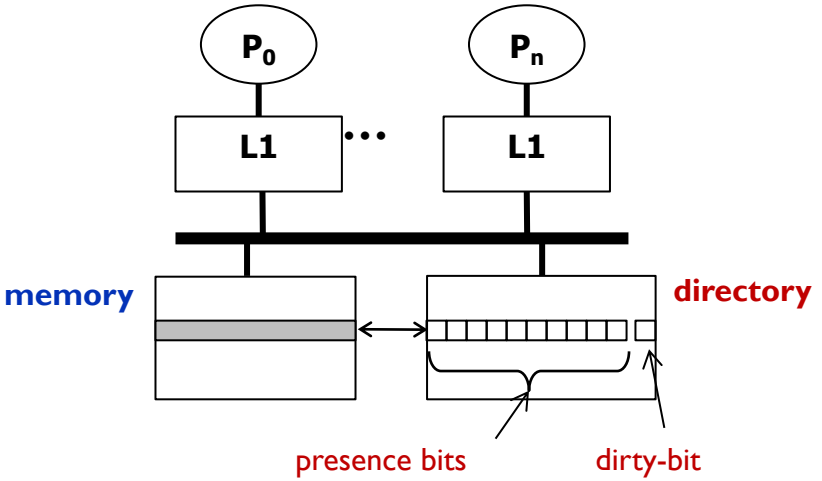
- More scalable solution for large scale machines

- Motivation
  - Snoopy schemes do not scale as they rely on broadcast

- Directory-based schemes allow scaling
  - Avoid broadcasts by:
    - Keeping track of all processors (PEs) which cache memory blocks
    - Using point-to-point messages to maintain coherence
  - Will work on any scalable point-to-point interconnect
    - No reliance on buses or other broadcast-based interconnects

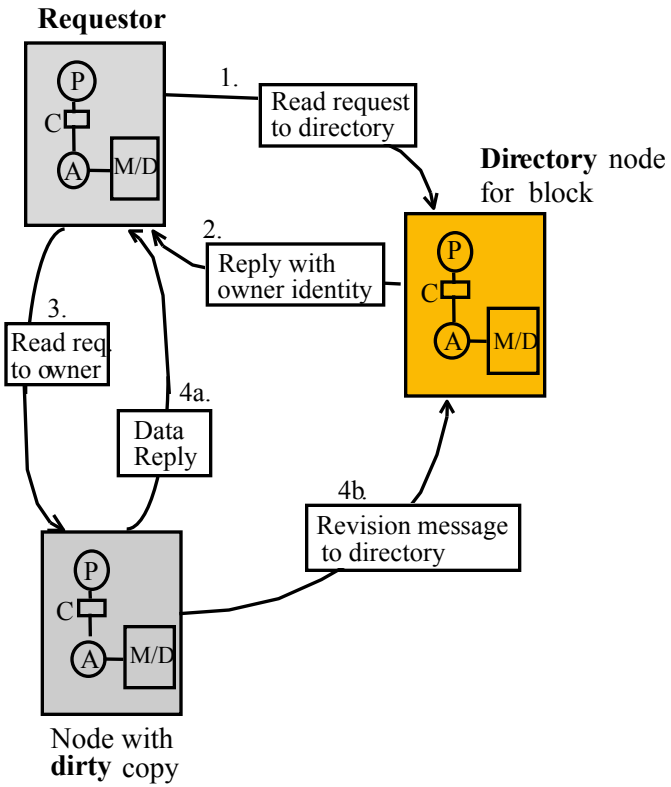
Basic Scheme of Directory Coherence

- Assume P processors
  - With each cache-block in memory
    - *P* presence bits, 1 dirty-bit
  - With each cache-block in cache
    - 1 valid bit, 1 dirty (owner) bit

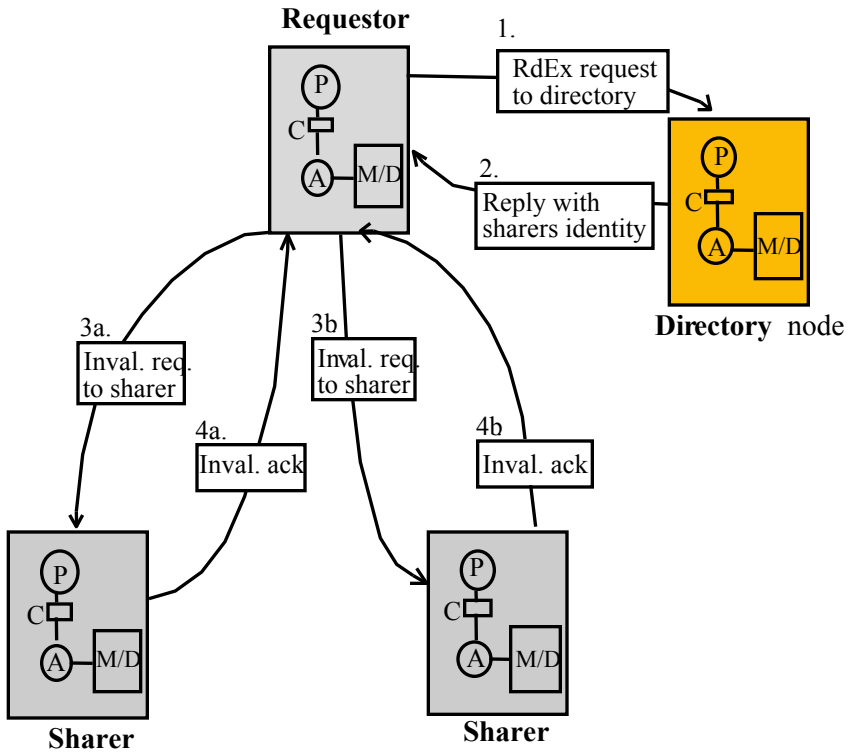


  - Read from main memory by P_i
    - If dirty-bit OFF { read from main memory; turn p[i] ON; }
    - If dirty-bit ON { recall line from dirty *P* (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to P_i }
  - Write to main memory by P_i
    - If dirty-bit OFF { supply data to P_i; send invalidations to all *P*s caching that block; turn dirty-bit ON; turn P[i] ON; others OFF }
    - If dirty-bit ON { recall line from dirty *P* (cache state to invalid); update memory; supply data to P_i; turn P[i] ON; others OFF }

Directory Protocol Examples



Read miss to a block in dirty state



Write miss to a block with two sharers

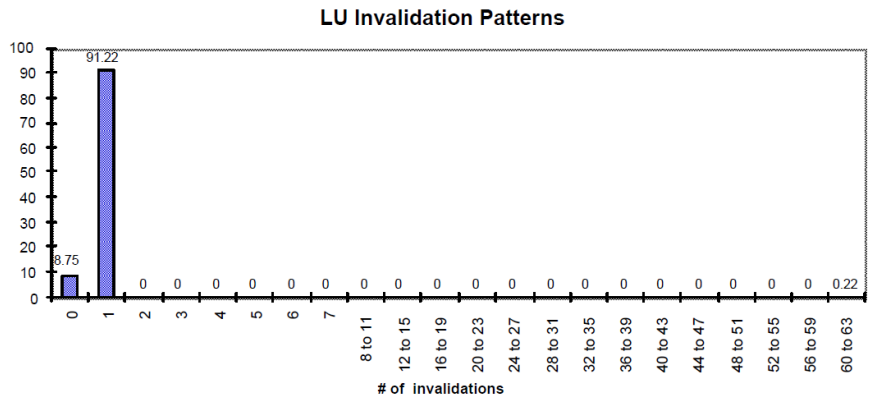
Scaling with Number of Processors

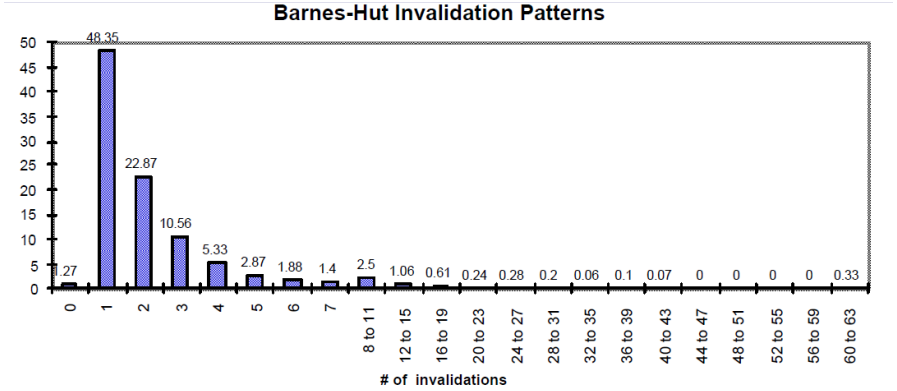
- Scaling of memory & directory
  - Centralized directory is BW bottleneck as in centralized memory
  - Maintain directory information in distributed way

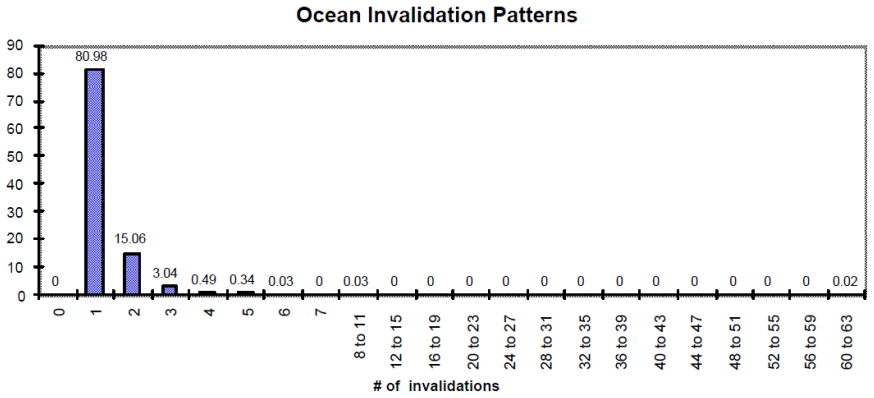
- Performance characteristics
  - Traffic: # of network transactions each time protocol is invoked
  - Latency: # of network transactions in critical path each time
- Directory storage requirements
  - Number of presence bits needed grows as # of processors grows
  - Lead to potentially large storage overhead
- Characteristics that matter
  - Frequency of write misses
  - How many sharers on a write miss

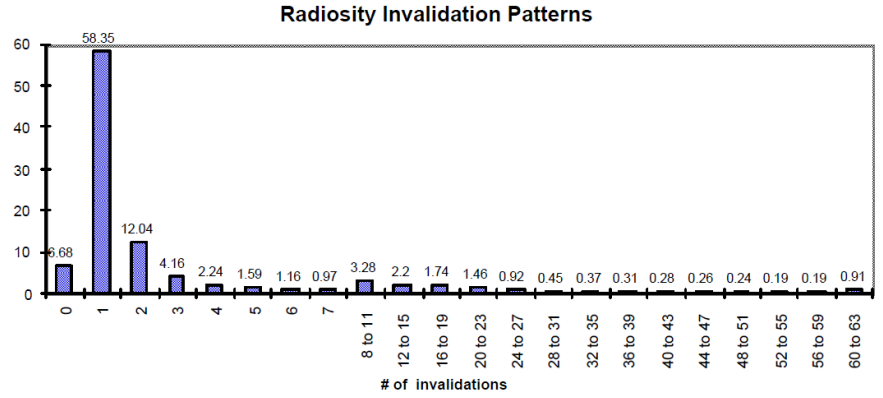
Cache Invalidation Patterns

- Number of sharers = number of invalidations
  - Small number of sharers for most of write misses - LU, Ocean
  - Several sharers, but a few sharers for most of write misses – B.-H., Radiosity





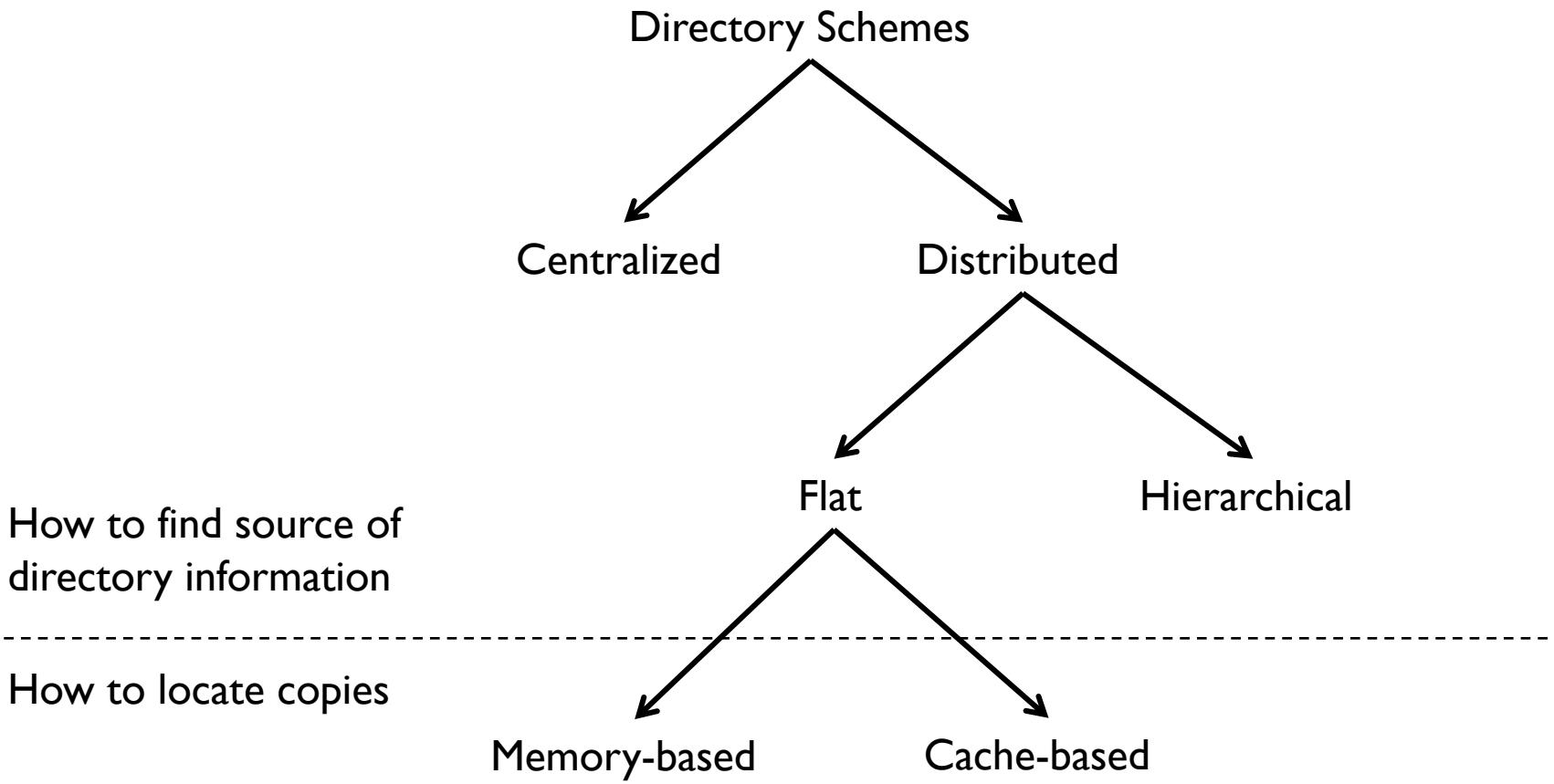




Sharing Patterns

- Generally, only a few sharers at a write, scales slowly with p
  - Code and read-only objects – no problems as rarely written
  - Migratory objects – even as # of Ps scales, only 1-2 invalidations
  - Mostly-read objects – invalidations are infrequent
  - Frequently read/written objects
    - Invalidations usually remain small, though frequent
  - Synchronization objects (e.g., lock variables)
    - Low contention locks results in small invalidation
    - High contention locks need special support (SW trees, queuing locks)
- Implies directories very useful in containing traffic
  - If organized properly, traffic and latency shouldn't scale too badly

Organizing Directories



Find Directory Information

- Centralized memory and directory
  - Easy but not scalable
- Distributed memory and directory
  - Flat schemes
    - Directory is distributed with memory at home node
    - Location of directory is obtained based on address (hashing)
    - Can send messages directly to home
  - Hierarchical schemes
    - Directory organized as a hierarchical data structure
    - Leaves are processors, internal nodes have directory states
      - Directory entry specifies whether the subtree caches the block
    - Send "search" message to parent and passed down to leaves

Locations of Copies (Presence Bits)

- Hierarchical schemes
  - Each directory has presence bits for its children (subtrees), dirty bit

- Flat schemes
  - Different storage overheads and performance characteristics
  - Memory-based schemes
    - Info about copies stored all at the home with the memory block
    - E.g., Dash, Alewife, SGI Origin, Flash
  - Cache-based schemes
    - Info about copies distributed among copies themselves
      - Each copy points to next – distributed doubly linked list
    - Scalable Coherent Interface (IEEE 1596-1992 SCI standard)

Flat, Memory-Based Schemes

- All info about copies
  - Co-located with the block itself at home
  - Works just like centralized scheme, except physically distributed

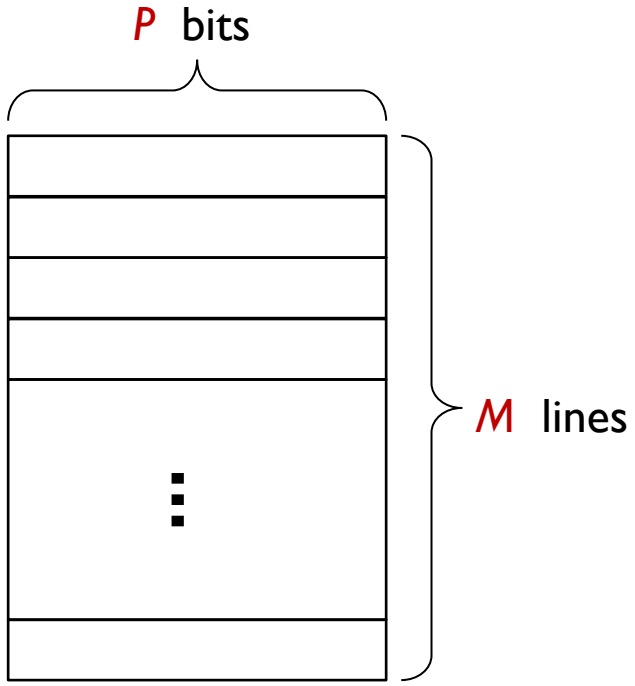
- Scaling of performance characteristics
  - Traffic on a write: proportional to number of sharers
    - Invalidation performed for each sharer individually
  - Latency of a write can be reduced by
    - Issuing invalidation to sharers in parallel

Flat, Memory-Based Schemes (cont'd)

- How does storage overhead scale?
  - Simplest representation: full bit vector
    - *i.e.* one presence bit per node
  - Directory storage overhead
    - Proportional to *P* * *M*

      P = # of processors (or nodes)

      M = # of blocks in memory
  - Does not scale well with P
    - Assume 64-byte cache line
    - Need a full bit vector per cache line
      - 64 nodes: 12.5% overhead
      - 256 nodes: 50% overhead
      - 1024 nodes: 200% overhead



Flat, Memory-Based Schemes (cont'd)

- Reducing storage overhead
  - Optimize full bit vector schemes
  - Limited pointer schemes
    - Reduce "width" of directory
  - Sparse directories
    - Reduce "height" of directory

Flat, Memory-Based Schemes (cont'd)

- The full bit vector schemes
  - Invalidation traffic is best, because sharing info is accurate

- Optimization for full bit vector schemes
  - Increase cache block size
    - Reduces storage overhead proportionally
    - Any problems in this approach?
  - Use multiprocessor nodes
    - Bit per multiprocessor node, not per processor
    - Still scales as P*M, but tolerates larger machines

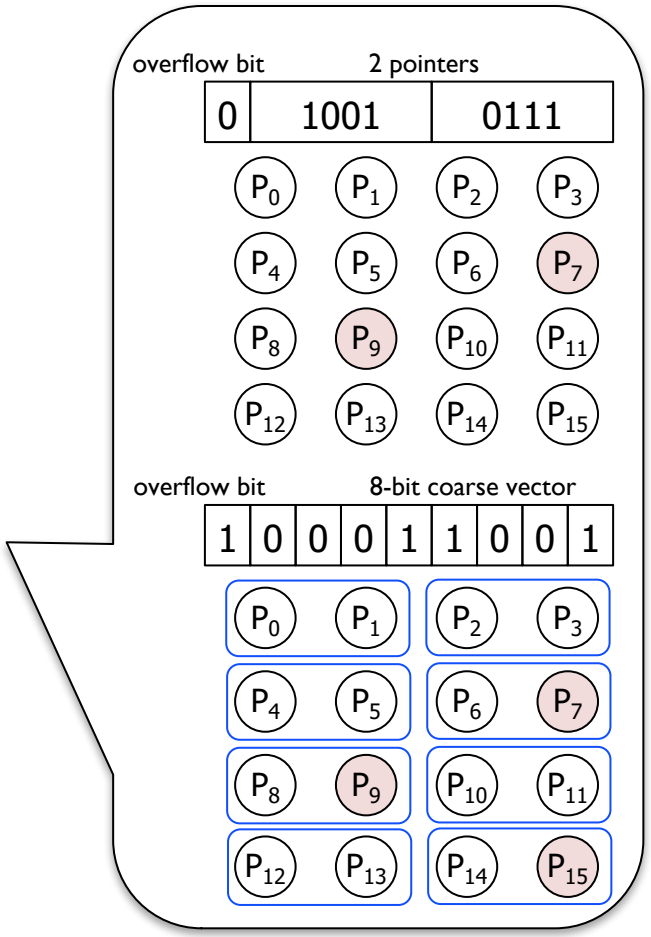
  - 128 byte line, 256 nodes with quad core processors
    - Overhead is 6.25% (256/4 bits per 128 byte = 1/16 overhead)

Flat, Memory-Based Schemes (cont'd)

- Limited pointer schemes
  - Observation:
    - Data will reside at a few caches at any time
    - A limited number of pointers per directory entry should be sufficient
  - Need overflow strategy
    - What if # of sharers exceeds # of given pointers
    - Many different schemes based on differing overflow strategies

Flat, Memory-Based Schemes (cont'd)

- Overflow schemes for Limited Pointers
  - Broadcast (Dir_iB)
    - Broadcast bit turned on upon overflow
  - No-broadcast (Dir_iNB)
    - On overflow, new sharer replaces one of the old one (and invalidate the old one)
  - Coarse Vector (Dir_iCV)
    - Change representation to a coarse vector
      - 1 bit per k nodes
    - On a write, invalidate all nodes that are corresponding to the bit

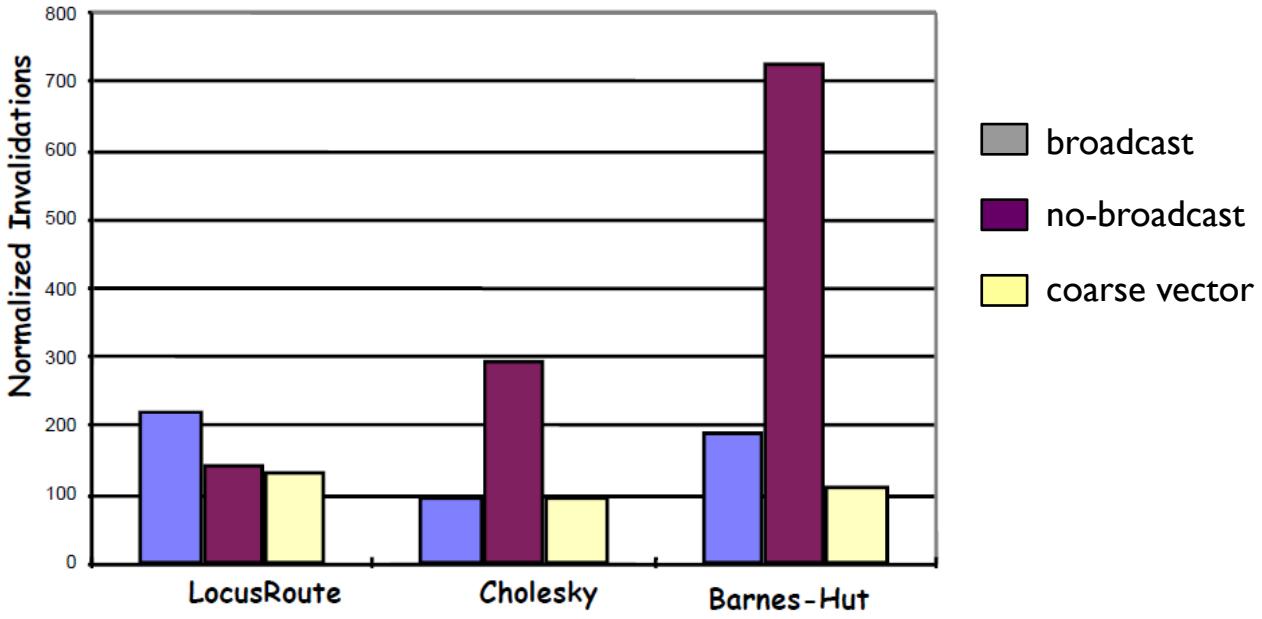


Flat, Memory-Based Schemes (cont'd)

- Overflow schemes (cont'd)
  - Software (Dir_iSW)
    - Trap to software, use any number of pointers (no precision loss)
      - MIT Alewife: 5 pointers, plus one bit for local node
    - Extra cost of interrupt processing on software
      - Processor overhead and occupancy
      - 84 cycles for 5 invalidations vs. 707 cycles for 6 invalidations
  - Dynamic Pointers (Dir_iDP)
    - Use pointers from a hardware free list in portion of memory
    - Manipulation done by hardware assist, not software
    - E.g., Stanford FLASH

Data on Invalidations

- Normalized number of invalidations
  - 64 processors, 4 pointers, normalized to full-bit-vector
  - Coarse vector shows quite robust performance
- General conclusion
  - Full bit vector is simple and good for moderate-scale machines
  - Several optimized schemes for large-scale machines, no clear winner

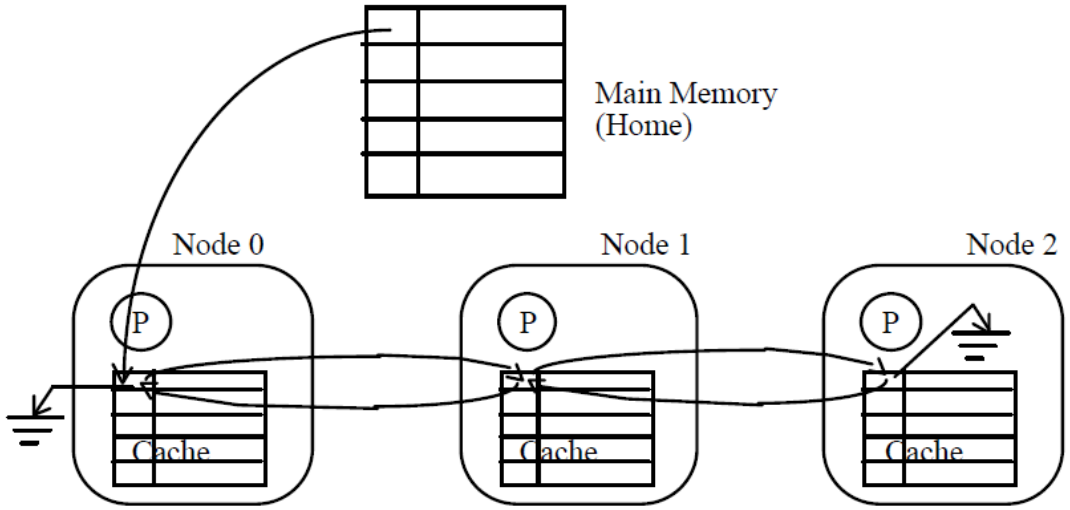


Flat, Memory-Based Schemes (cont'd)

- Reducing height: Sparse Directories
  - 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
    - No need for backup store,
      but send invalidations to all sharers when entry is replaced
    - One entry per line: no spatial locality
    - Different access patterns (from many procs, but filtered)
    - Allows use of SRAM, can be in critical path
    - Needs high associativity, and should be large enough
    - Can trade off width and height

Flat, Cache-Based Schemes

- Operations
  - 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 (communication needed)
  - On write: propagate chain of invalidations down the list
- Scalable Coherent Interface (SCI) IEEE standard
  - Doubly linked list



Flat, Cache-Based Schemes (cont'd)

- Scaling properties
  - Traffic on write: proportional to # of sharers
  - Latency on write: proportional to # of sharers
    - Don't know the identity of next sharer until reach current one
    - Assist processing at each node along the way
    - Even reads involve more than one other assist (home and the first sharer in the list to insert an entry)
  - Storage overhead: quite good scaling
    - Only one head pointer per memory block
    - Rest is all proportional to cache size (but need to store them in SRAM)
  - Other properties
    - Good: mature IEEE standard, fairness
    - Bad: complex

Directory Organization

- Flat Schemes
  - Issue: finding source of directory data – go to home, based on addr
  - Issue: finding out where the copies are
    - Memory-based: all info is in the directory at home
    - Cache-based: home has pointer to first element of distributed, linked list
  - Issue: communicating with those copies
    - Memory-based: point-to-point messages - can be multicast or overlapped
    - Cache-based: 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

Directory Protocol

- Directory approaches
  - Directory offer scalable coherence on general networks
    - No need for broadcast media
  - Many possibilities for organizing directory and managing protocols

- Hierarchical vs. flat directories
  - Hierarchical directories are not used much
    - High latency, many network transactions, and BW bottleneck near root
  - Both memory-based and cache-based flat schemes are popular
    - For memory-based, full-bit vector suffices for moderate scale machines

Summary

- Scalable coherence schemes for large machines
  - Hierarchical snoop-based cache coherence protocol
  - Directory-based cache coherence protocol
- Hierarchical snoop-based coherence
  - Inclusion property should be maintained
- Directory-based coherence
  - Flat, memory-based directory is dominant
    - Limited # of pointers: overflow strategy needed

References

- Chapter 6.3, 8.1~8.3, 8.10 in Parallel Computer Architecture: A Hardware/Software Approach, D. E. Culler, J. P. Sigh, A. Gupta, Morgan Kaufmann, 2002